**CICLO 4 PROGRAMACIÓN – DESARROLLO DE APLICACIONES MÓVILES**

Se realizará un trabajo con bases de datos NoSQL, es decir que se puede guardar información de otra forma, en este caso se utilizará ***MongoDB*** la página es:

<https://www.mongodb.com/>

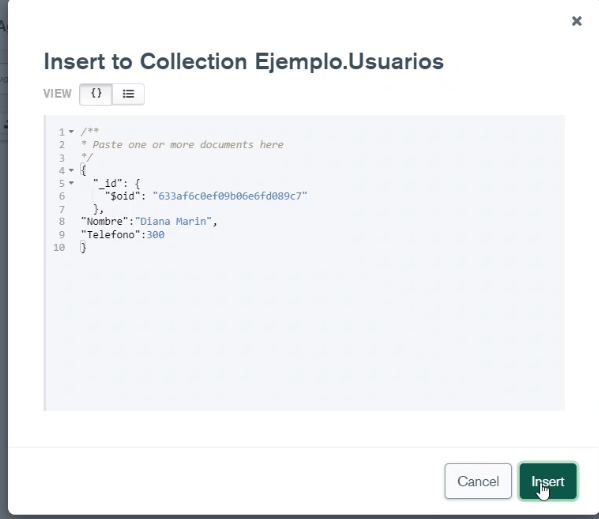
Se va a utilizar la nube y la interfaz de usuario de manera local.

1. Acceder a la nube, acceder de manera gratuita a mongodb, se puede acceder por wmedio de la cuenta de Google al ATLAS. (Es importante recordar que la página se actualiza, por lo tanto los vídeos que se encuentran en la página de la MISION no se encuentran actualizados).
2. Crear una organización – estructura jerárquica de carpetas, es por estándar de organización. (Colocar nombre a la carpeta), se crea la organización.
3. Crear un nuevo proyecto – se le da un nombre y se le da siguiente.
4. Construir la base de datos. – mongoDb le llama cluster (al espacio de almacenamiento)
5. Elegir el servidor (servicio de almacenamiento) – esta configuración no se toca, por cuestión de conexiones de internet internacional, lo mejor es conectarse a los servidores de Estados Unidos.
6. Solo se modifica el CLUSTER NAME, y se le da crear.
7. Dar un nombre de usuario y una contraseña.
8. Herramienta de manera local (SE HACE MÁS FÁCIL LA ADMINISTRACIÓN DE LA BASE DE DATOS)
9. **COMPASS,** se encuentra en products, se le da descargar, es un archivo ejecutable.
10. Instalar el programa, es solo dar siguiente hasta el final.

Debemos obtener una cadena de conexión, para que todo lo que se haga de manera local se refleje en la nube

1. Network Access
2. Add IP Address – si quiero que solo yo sea la persona que va a enviar datos a la bd, se tiene que indicar la dirección ip de mi equipo, para obtener esta información se usa el cmd, el proveedor del servicio entrega una ip del exterior (ipconfig /all), en este caso se busca *“mi ip pública”* en Google.
3. Permitir el acceso de cualquier parte, lo cual es lo mejor para poder trabajar desde diferentes puntos. *ALLOW ACCESS FROM ANYWHERE*
4. Database Access
5. Edit
6. Donde dice DATABASE USER PRIVILEGES – built-in-role \*\*ATLAS ADMIN\*\*
7. Actualizar
8. Buscar cadena de conexión para poder conectarme.
9. BD que se creo, darle conectar, en este caso lo hacemos por medio del mongo Compass. Esto genera una cadena de conexión, la cual se puede copiar y pegar en el mongo compass., **se debe modificar la contraseña al momento de pegar la cadena de conexión, para que se pueda conectar al servidor (quitar los parentesis).**
10. Crear una nueva base de datos, dar un nombre, ahora no se van a manejar tablas, de ahora en adelante serán **colecciones**
11. Darle click a la colección y darle – add Data, no confundir el file con el document, document hace analogía a insertar un registro en la tabla usuarios. Esto entrega un archivo JSON con un id hexadecimal, esto se conoce como el OBJECTID.

1. La fecha y hora exacta de creación del documento, conocido también como timestamp (4 bytes) 2. Un número aleatorio (5 bytes) 3. Un contador incremental aleatorio que inicia en otro valor aleatorio



**05 Octubre 2022**

**Manejo de versionamiento – Creación de un repositorio.**

Es mejor hacer un repositorio para hacer las entregas semanales, por lo tanto todo se va a manejar desde GitHub, es una herramienta útil para el desarrollo de muchos proyectos.

Un repositorio es una herramienta en la nube que me permite tener un espacio de almacenamiento donde se tienen los documentos en el equipo y en la nube (servicio de almacenamiento), la herramienta se conoce como **Git,** por medio de unos comandos en consola se envía la información sin conectarme necesariamente a la nube.

A través de comandos se envía información en la nube (aplicación ya instalada). Para acceder a la consola:

1. Se puede hacer por la consola de Windows o la consola específica del Git
2. Tecla de Windows y buscar ***GitBash.***
3. En la consola de Gitbash se pueden ejecutar los comandos, este por defecto se ubica en un espacio del equipo, generalmente se ubica dentro del usuario.
4. Esta consola es sensible a mayúsculas y minúsculas en los comandos
5. Si se usa Gitbash, se puede abrir desde otra carpeta, por lo tanto, se le pueda dar click derecho y se envía la consola desde esa carpeta ***GitBash here***.

GitHub

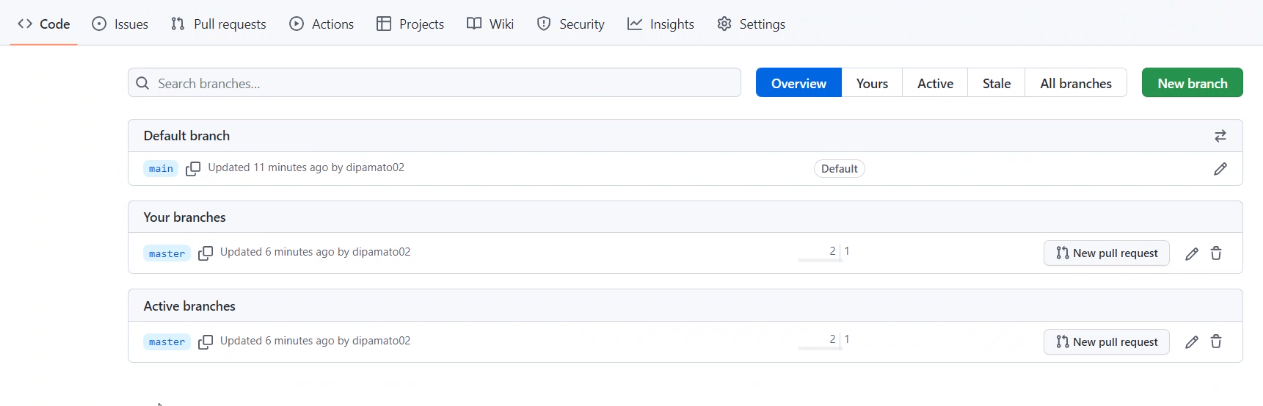
Es la herramienta que nos permite almacenar información en la nube y se debe registrar en github.com, parar realizar el almacenamiento se debe crear un nuevo repositorio con su respectivo nombre. Si el archivo es privado, se puede compartir el repositorio, sin embargo, cada vez que alguien nuevo vaya entrar se me pedirá permiso.

En este punto la subida del código y de la información se encuentra en el archivo de videos del mismo ciclo. En este caso se actualizó la versión del readme y esto muestra una serie de versiones en el GitHub, con un número. Revisar grabación a las 8:45.

Si se quiere hacer un envío de programación

1. Abrir la consola del Gitbash desde la carpeta.
2. No se necesita enviar uno por uno.
3. git init
4. El comando ls muestra los archivos que están en la carpeta
5. Git add . (con el espacio), ya se prepara para subir la carpeta
6. Git commit -m “otros archivos” (aquí se pone nombre para realizar la carga)
7. Git Branch -m main
8. Git remote add origin
9. Ir al github a la parte de #code# y buscar la url del archvio
10. Git remote add origin https://url
11. Git push -u origin main
12. En este caso se mostró un error en el main (fetch first – liberar la rama)
13. Git Branch -m master
14. Git push -u origin master

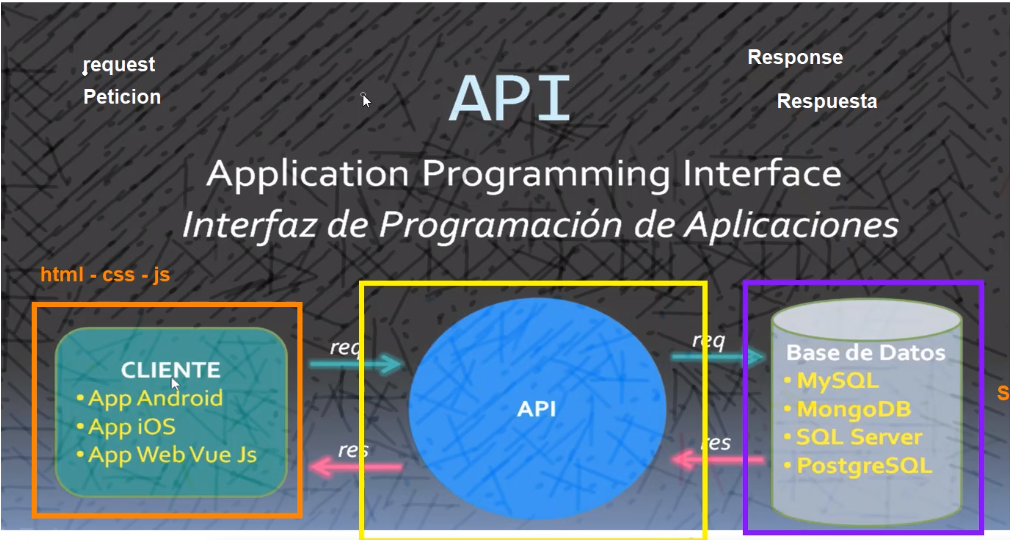
Unificar con el github, se revisaron las ramas dentro de la pg de GitHub.

El manejo del repositorio se va a hacer durante todo el ciclo, en este caso se va a trabajar en esto más adelante.

***APLICACIONES RESTFUL***

Diferencia entre las aplicaciones (buscar), las api son una interfaz de programación de aplicaciones.

API – Application Programmming Interface. – se utilizar un lenguaje de programación para enviar información al cliente (react, angular, etc) y por el lado del servidor se pueden usar bases de datos relacionales o no relacionales. Programar una estructura para unificar ambos elementos.



Nosotros vamos a consumir una api que es de doble autenticación, esto va a estar en la parte de codificación.

Cliente, html + css + javascript, en el ciclo 3 y ciclo 2 se realizó el trabajo en bases de datos relacionales.

Investigar sobre el Middleware – proveerle una respues al cliente, donde este ultimo también puede enviar información.

En este caso el backend se divide en dos partes, la que va dirigida al servidor (almacenamiento) y la construcción de la aplicación (consumo de la aplicación).

REST

Transferencia de estado representacional, la api debe cumplir con las siguientes condiciones

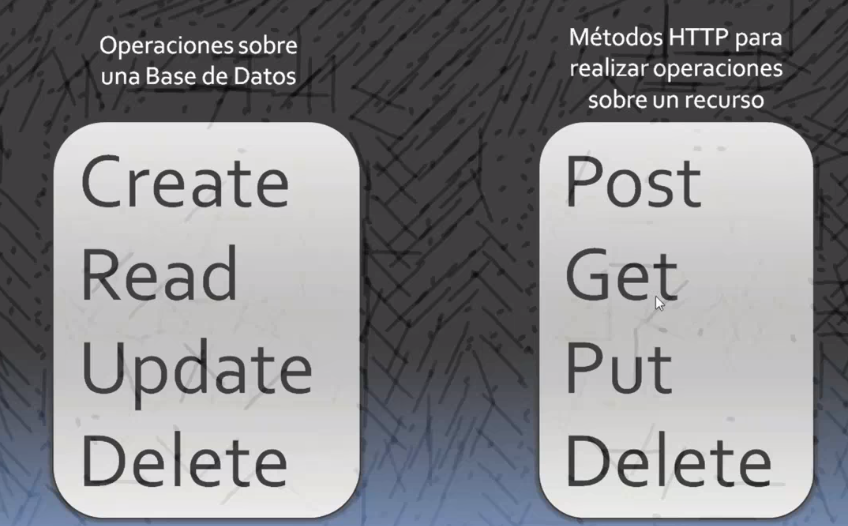
1. Uso de una interfaz uniforma: se debe garantizar que al generar una API, la url básica no debe cambiar y por lo tanto ya no sería restful, utilizan los métodos get, post, put y delete (no tiene nada que ver con los métodos de las bases de datos), aunque se parecen un poco
2. Modelo cliente – servidor: garantizar que la interfaz gráfica es independiente de toda la lógica de programación.
3. Operaciones sin estado: todas las operaciones que se ejecutan en la aplicación se deben ejecutar desde el navegador. El resto de operaciones a nivel lógico deben volver a hacer una consulta al servidor
4. Almacenamiento en Caché
5. Sistema de capas: hace referencia a que yo puedo usar diferentes capas de diferentes aplicaciones a través de microservicios.
6. Código baja demanda: garantizar que el servidor, cuando este devuelve valores, estos deben manejar formato xml o json.

Si no tiene estas 6 características, ya no sería una aplicación resful.

**Octubre 7 de 2022**

**Request**

Solicitudes de http, se hacen por medio de métodos



Los métodos del http se relacionan con los métodos de las bases de datos.

**Codificación en node.js. – Columna vertebral. – es un lenguaje parecido a phyton.**

Se complementa por módulos. Permite trabajar por el lado del cliente (html, css y javascript) y del servidor. Cuando se habla de framework, para el cliente se usa angular, esto es básicamente para el frontend. Para este caso los más utilizados son angular y react.

El backend se compone por las conexiones a la base de datos y a la forma en la que se estructura el api, en este caso se va a utilizar LOOPBACK, también existen otros como el adonis, etc.

Para ver si node está instalado, se ejecuta el CMD y se escribe la palabra “*node*”, es una estructura de codificación muy básica. Sin embargo, el trabajo en este cmd no queda guardado, por lo tanto se va a trabajar en visual studio code (es una herramienta similar al bloc de notas), no es un entorno de desarrollo especializado.

Cuando se trabaja desde la terminal se puede seleccionar el gitbash para enviar información al repositorio. Esto se puede realizar por defecto en el desplegable al lado del powershell

pc1hwe

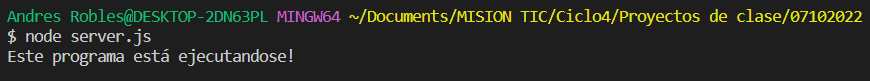
1. Abrir la carpeta del proyecto en visual studio code, por medio de la consola del gitbash, este va a ser un proyecto de node, por lo tanto se tiene que generar un archivo de configuración.
2. Aquí hay un asistente de configuración “***npm***” este equivale al dotnet del .NET, a la profe le aparecía el main y el master, esto surge ya que se había hecho envío al github. Para estructurar el proyecto se tiene que hacer uso del npm (node package manager)
3. Proceso de generación npm init (esta instrucción genera un paquete de configuración). El -y evita bastantes pasos (no solicita información detallada).

Npm init -y

En este caso dejamos el init solo para realizar la inicialización, se necesita un nombre (no será visto por el usuario o por el cliente). El único elemento que va a ser importante es *EL NOMBRE DE ENTRADA – (Nombre para el archivo de tipo main siempre va con un .js).* **Debe ser consecuente con el archivo que se va a generar.**

Los requerimientos siguientes a la entrada no son relevantes.

1. Se genera un paquete JSON que es el inicio de la configuración. Este paquete asigna una identificación diciendo que este va a ser un archivo node js.
2. Generar el main (ESTO ES OBLIGATORIO SIEMPRE). El nombre va a llevar el mismo nombre que se le dio a la entrada. Se puede generar con VSC o con el comando touch server.js
3. Ya se está en capacidad de correr la aplicación, aunque aún no hay información.
4. Para correr el servidor se hacer un *node server.js (lo que está subrayado corresponde al nombre que se le dio al servidor).* El js significa que se encuentra en lenguaje Javascript.
5. Console.log() muestra un mensaje en la consola de la aplicación.



Como se sigue trabajando en VSC se tiene que seguir guardando cada vez que se haga un cambio. Node tiene un módulo que permite hacer la ejecución en tiempo real.

NODEMON

Nodemon, es un complemento de node.

1. Cada vez que se va a utilizar un módulo (funcionalidad adicional) se debe hacer uso del **NPM.**

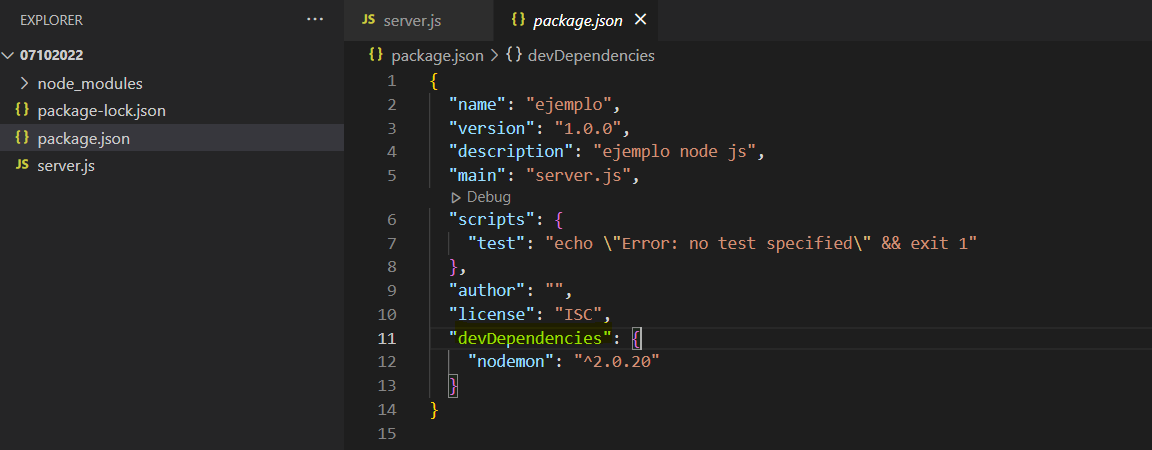
Npm i (se puede usar solo la i o la palabra completa install) nodemon

La pantalla se puede limpiar con el comando **clear.**

En node se tiene la posibilidad de instalar modulos mientras se está en el desarrollo (codificación), pero que no es necesario ejecutar en la producción (despliegue de información), hay que tener cuidado con los paquetes que se van a enviar a la producción.

-D o -P son banderas para saber si van al desarrollo o la producción

Esto genera una instalación de manera local, únicamente para este proyecto. Los módulos que se van instalando hacen referencia a dependencias.



La imagen muestra que el paquete fue instadalo en el dev, es decir en el estado de desarrollo. Los que aparecen solo con Dependencies, es para un entorno global.

* En este caso en particular, cuando se hace la instalación del paquete y se intenta correr el programa con el comando nodemon server.js, aparece, bash: nodemon: command not found, esto pasa por una configuración del equipo. Cuando se ejecuta en la consola de poweshell muestra el error cmdlet.

npm i -g nodemon (esto coloca la instalación de manera global)

El cmdlet suele ser un error en el path que se encuentra en las propiedades avanzadas del equipo, en este hay unos comandos de ejecución – traducción, si estos archivos no están actualizados algunas veces, estas generan error.

Esto se puede automatizar colocando la instrucción del nodemon dentro del archivo json.



Para correr la aplicación se tendrán entonces dos opciones:

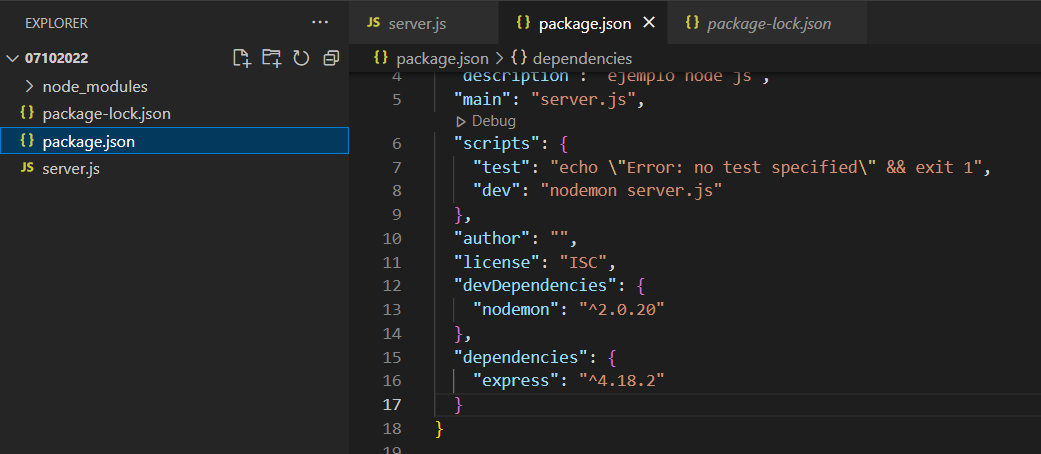
1. Nodemon server.js
2. Npm run dev

Esta es una aplicación en consola y aquí lo importante es generar un servidor, para hacer esto hay que instalar un paquete específico, este es llamado el módulo express (ejecuta como un servidor) y permite ser conectado por medio de un navegador.

Cuando se instalan paquetes, es importante detener el servidor con ctrl + c, express es muy similar a una librería y permite cambiar una aplicación nodejs a un servidor.

npm i express –save (se usa esta instrucción para guardar la información de manera global)

Dentro del archivo de configuración de JSON se va a mostrar la dependencia (desarrollo y producción)



**CONVERTIR UNA APLICACIÓN DE TIPO CONSOLA A UNA APLICACIÓN DE SERVIDOR**

1. Instalar el paquete express.
2. Importar el paquete al archivo que se quiere convertir, la forma de importar de los paquetes se realiza de la siguiente manera.

Voy a almacenar dentro de una constante

Const: Valor que nunca va a cambiar durante la codificación

Let: Espacio en memoria que puede cambiar a medida que avanza el proyecto (es más grande que el var), este puede cambiar de acuerdo al bloque de código que se esté utilizando.

Var: Es similar al let, a programación a gran escala se escogen entre el let y el var por el espacio de almacenamiento.

En este caso se va a utilizar la ***const*** debido a que esto va a estar en todo el desarrollo.

Const express = require (“express”); // Importación del paquete express.

Express es un objeto

La palabra ***requiere*** trae el módulo desde otra carpeta y permite su uso.

1. El express se entiende como una clase específica, en este caso se sigue trabajando bajo el POO. Esta clase suministra muchos métodos para convertir la aplicación en un servidor.

Cada vez que yo creo un objeto hay que instanciarlo. Esto hace referencia a iniciar para poder utilizar.

Const app=new express (); -- va a tener la posibilidad de acceder a todos los métodos del express.

1. Uso del método listen

App.listen(), yo le debo sumistrar los datos desde donde se ejecutaraá el servidor.

app.listen(3000, function(){

    console.log("Servidor ejecutandose desde el puerto 3000")

})

Hay dos formas para crear las funciones en el lenguaje de codificación de javascript. La primera que usa la palabra clave *function* y la segunda que hace uso de una función flecha ()=>{}

app.listen(3000, ()=> {

    console.log("Servidor ejecutándose desde el puerto 3000")

})

En este momento no hemos creado ningún recurso, este se creo por defecto en el localhost en el puerto 3000

1. Indicar la ruta.

Para entender cómo funciona la ruta se hace un ejemplo con un app.get y una respuesta.

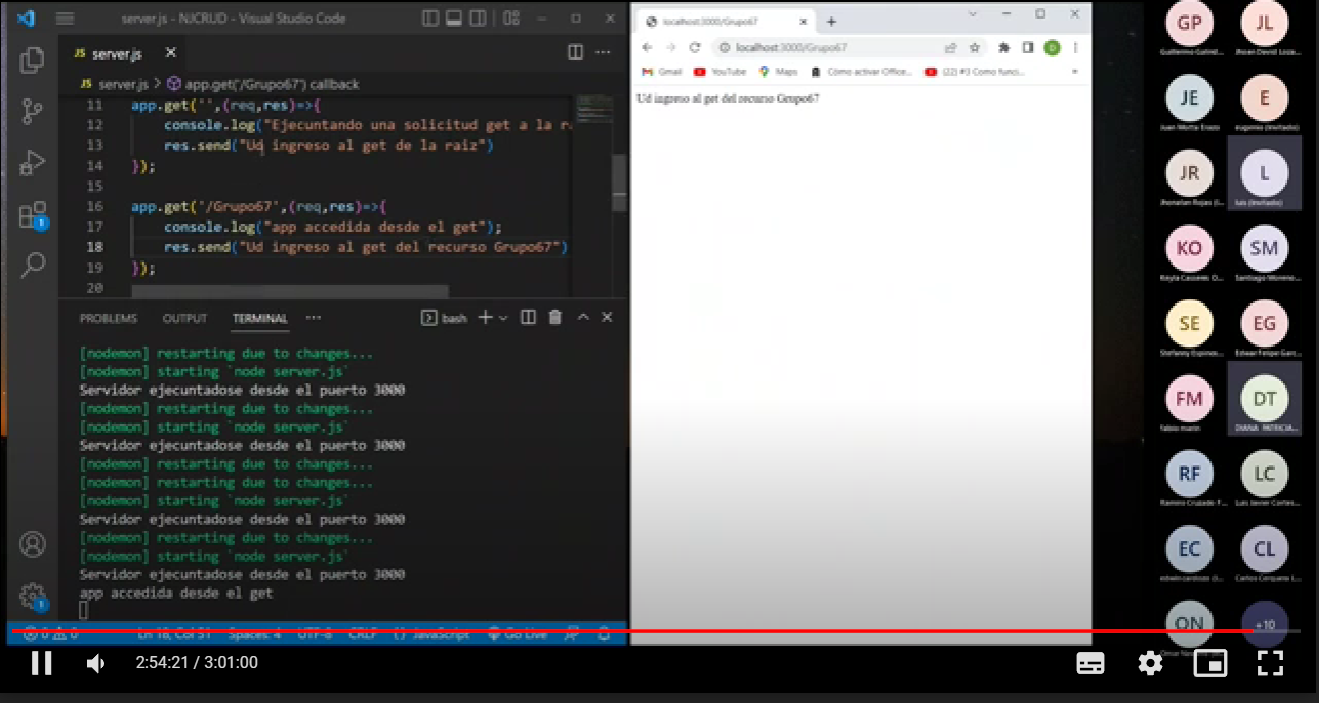
app.get('', (req,res)=>{

    console.log("Ejecutandose una solución get a la raíz de la app")

    res.send("Usted ingresó al get de la raíz")

})

En este caso se utiliza ‘ ’ refiriéndose al localhost, se entiende por defecto que este código se alojará en el localhost. Las palabras req (require) y res (response) son un estándar de codificación que se suele usar para lo que se necesita y lo que se envía. El localhost reconoce el mensaje sin necesidad de hacer una conversión debido a que esto está en javascript.



De ahora en adelante las páginas que se presente en el localhost no se llamarán páginas, en cambio se nombraran como recursos.

app.get('/Grupo67', (req,res)=>{

    console.log("app accedida desde el get")

    res.send("Usted ingresó al get del recurso grupo67")

})

**10 de Octubre 2022**

**Interacción por el postman**

**Crear un workspace**

**Instanciación de mongoose**

Esto ya se realizó en los vídeos anteriormente. Este paquete no necesita una instanciación (esto depende del paquete, por lo tanto es importante estudiar el paquete antes de su uso).

1. Se conecta a mogoose por medio de una constante

Const mongoose=require(“mongoose”);

1. Usar la función connect de mongoose

Mongoose.connect(“Cadena de conexión”)

La cadena de conexión se obtiene de mongodb, en mi caso, se usa el MongoDb drivers.

mongodb+srv://Grupo67:<password>@clusterprueba.iv8ld1u.mongodb.net/?retryWrites=true&w=majority

Una vez se tiene la cadena de conexión se cambia la contraseña y se coloca el nombre de la base de datos, como se ve en el punto siguiente.

1. Cambiar el nombre a la base de datos, entre el / y el ?

mongoose.connect("mongodb+srv://Grupo67:Grupo67@clusterprueba.iv8ld1u.mongodb.net/Grupo67?retryWrites=true&w=majority")

***Es importante recordar que la contraseña va sin los paréntesis.***

1. Generar una estructura como lo fueron las clases en java y las entidades en .Net o C#, en este caso se nombra como *MODELO en javascript*.
2. Creación de la carpeta ***modelos***.

La estructura va a cambiar con respecto a lo trabajado anteriormente ya que mongodb no es una base de datos relacional, por lo tanto no se va a tener un formato de tablas, en este caso se va a hablar de colecciones, donde cada uno de los registros se va a llamar documentos, los cuales tendrán datos de tipo JSON.

Para almacenar los datos se hace a través de modelos por medio de schemas.

El **mapeo** de las bases de datos se va a hacer por medio de **mongoose**. Las relaciones son muy fáciles de modelar.

1. Creación de modulos deporte y atleta.

Deporte.js y atleta.js.

Mongoose nos va a ayudar a construir esto, por lo tanto es necesario hacer una importación.

1. Creación del schema para el mapeo de la base de datos.

Const deporteSchema=new mongoose.Schema

Esto nos ayuda a mandar la información a nuestra base de datos, como es un recurso de tipo json se deben usar los {}

Para enviar información se debe consultar la API de mongoose. En este caso se pueden tener datos sencillos como el string o un compuesto con condiciones.

const deporteSchema= new mongoose.Schema({

    nombre:{type:String, required:true},

    numeroJugadores: Number,

    pais: {type:String, requiered:true}

})

Así queda de manera general.

1. Convertir en un modelo, el cual por sintaxis debe tener el mismo nombre de la clase.

const Deporte=mongoose.model('Deporte', deporteSchema, 'Deportes')

El modelo es el que se comunica y por ende permite la comunicación de la base de datos. Este model solicita 3 parámetros. Cómo se va a llamar el documento, el schema para mandar los datos en JSON y cómo se va a llamar esa colección

1. Dar permisos para la exportación

module.exports=Deporte;

1. Llamar el módulo desde el servidor (server.js) por medio de una constante

Const deporteSchema = require()

Dar permito de acceso ./ (entra a la raíz) / modelos / deporte), en este caso esto no es un módulo, cómo es una carpeta se realiza por medio de la siguiente instrucción

const deporteSchema=require('./modelos/deporte')

1. En la clase pasada, por medio del express se logró comunicar con el get de una manera básica, el express usa el método router para construir la aplicación, nos va a permitir generar las rutas, esto también se hace desde el server.js

const router=express.Router();

Para usar los datos por medio del postman, se obtendrán por medio del body los cuales van a ser capturados en el formato estándar (urlencoded), en el body es donde se van a construir todo lo relacionado con los formularios. En este caso se debe preparar el servidor por medio de el comando:

app.use(express.urlencoded({extended:true}))

Los datos los va a codificar y enviar a través de express en un formato de tipo JSON. Los datos los va a enviar a través de express por un formato JSON.

app.use(express.json());

1. Cuando se utiliza este móduli hay que dale permiso para trabajar con el router

app.use(router);

**Métodos CRUD**

El router es la ruta que le damos al usuario externo para que consulte nuestras funcionalidades.

Probar si está funcionando por medio del navegador por medio del router con la instrucción send que se usó en la página web. Este se puede probar a través del navegador

router.get ('', (req,res)=>{

    res.send("Bienvenido a mi primera API!")

})

Crear a través del método post, para hacer esto es importante generar una nueva ruta. En este caso sí se va a utilizar el req que se encuentra en la función flecha.

Router.post(‘/deporte’, (req,res)=>{

let nuevoDeporte = new deporteSchema({

nombre:req.body.nombre,

numeroJugadores:req.body.jugadores,

país: req.body.pais

});

nuevoDeporte.save((err,datos)=>{

if(err){

console.log(“No se pudo almacenar el dato”)

}

Res.send(“Dato almacenado de manera correcta!!!”)

}

})

Como se le envía un nuevo deporte, este se debe enviar por medio de un archivo JSON, como se puede ver anteriormente

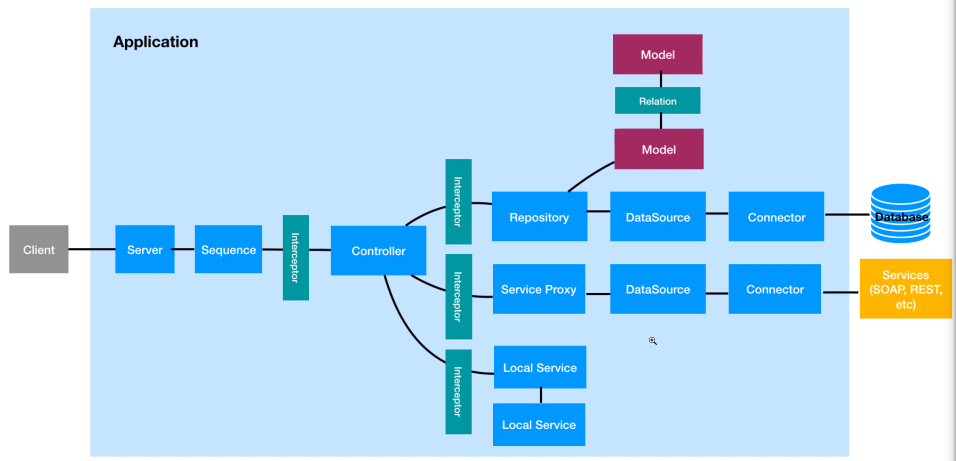
Guardar por medio del nuevoDeporte.save, como se ve en el código anterior.

**Callback:** Una función que se va a ejecutar después de que guarde la información. Generalmente puede traer un error o los datos que fueron almacenados.

**12 de Octubre de 2022**

**Lógica de negocio y trabajo con el backend, conexión a la base de datos – LOOPBACK**

Datasource (permite gestionar la conexión a la base de datos) – model (mapeo, tablas a clases dentro del loopback) – relaciones – repositorios (gestionan los datos a través de Loopback) – servicios –controlador (llama a recursos internos) – middleware (interceptor, filtro de autenticación, cuando se hace una solicitud se tiene que brindar permisos de acceso)– servidor y la secuencia , controlador (lógica de negocio). Todo lo anterior viene previamente establecido.



CRUD



1.

Instalación, toda la información se encuentra en loopback.io con todos los pasos a seguir.

npm install -g @loopback/cli

La instalación debe realizar en una carpeta específica, ya que esto es lo mejor para el proceso. Esto ya tiene algunas funciones que se habían instalado anteriormente. Yo lo había hecho anteriormente, por lo tanto creé una nueva carpeta para tener el archivo de forma ordenada.

Loopback nos va a ayudar con el proceso de administración de las bases de datos y a poner en funcionamiento del servidor. Ya tiene preestablecida toda la estructura, toda la estructura interna tiene construida la base de la aplicación. En este caso se va a trabajar con loopback 4.

Loopback, empieza con la database (en este caso mongoDB), con el conector y la DataSource. Posteriormente se realizan los modelos y se trabaja en las relaciones (un pedido tiene que tener un receptor de manera obligatorio), aunque no es una base de datos relacional, hay algunas relaciones que sí se deben notar de forma intrínseca.

Nosotros haremos un proceso de autenticación por medio del local Service a través de twilio.

**API en loopback**

1. Ejecutar el comando lb4 app
2. Dar un nombre y una descripción del proyecto
3. Crear un archivo raíz, generalmente se llama Backend**Pedidos,** lo que se encuentra en negrilla es algo especial para este proyecto.
4. Dar un nombre de la clase de aplicación: **app**
5. Ejecutar todas las características (estas se pueden retirar manteniendo la barra espaciadora oprimida y dando click), la aplicación da una serie de ayudas por si se quiere modificar algo dentro de los instaladores.
6. Se genera un proyecto por defecto

Es importante recordar que se generan unas carpetas

1. Node\_modules 🡪 son algunos paquetes que se instalan de forma rápida
2. Public 🡪 entorno básico de tipo html
3. Archivos ignore, son formas de almacenar la información. Por ejemplo, el git ignora toda la carpeta de node modules, ya que se considera una información no necesaria, esto es importante ya que hay información que no tenemos que versionar de manera obligatoria.

La carpeta principal de trabajo va a ser la src, en ella vienen una estructura arquitectónica con algunas carpetas (controladores, repositorios, modelos), todos en archivos de tipo lectura.

Para correr la aplicación, el ejecutable va a estar dentro de la carpeta BackendPedidos

*Cuando corrí el npm start me falló, en dado caso de que esto vuelva a pasar se puede ejecutar el código*npm cache clean --force

1. Cd BackendPedidos
2. Npm start
3. Mirar dónde se está ejecutando. (localhost puerto 3000)
4. Al ejecutar se genera una primera parte básica del html (entorno gráfico que se puede ver en el public)
5. Se generan dos archivos explorer (en este archivo se puede var un método get inicial) y json.

Seguir con las instrucciones que se encuentran en loopback.io

lb4 datasource

construir el datasource ya que en la carpeta src (source) no se encuentra.

1. Nombre de Datasource: mongodb
2. Conector para realizar la conexión (es algo parecido al mongoose): con el uso de las flechas se busca MongoDB (StrongLoop).
3. Se tiene la cadena de conexión? No
4. Dejar todo lo demás vacíos
5. Yes al final.

Conectar con mongoDb, por lo tanto se tiene que acceder a mongo desde el navegador para obtener la cadena de conexión, esta se debe pegar en el mongodb.datasource.ts. (que se encuentra en la carpeta llamada datasources) en el **url,** se debe cambiar la contraseña y quitar los paréntesis y cambiar el nombre.

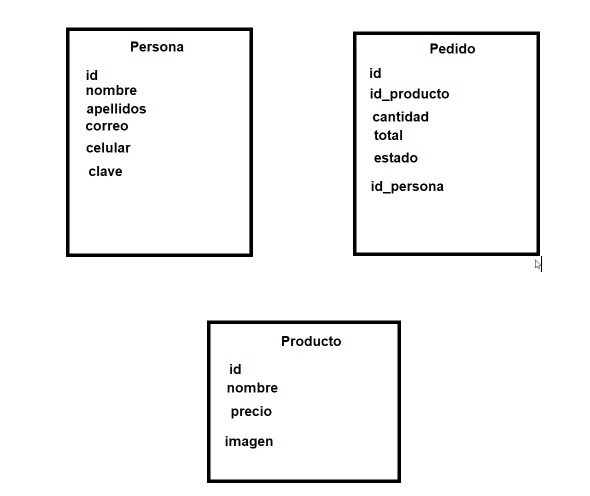
  url: 'mongodb+srv://Grupo67:Grupo67@intentoclase.jh4h4xp.mongodb.net/Grupo67?retryWrites=true&w=majority',

Dentro de la carpeta src (source), se encuentran el archivo index, application, sequence (middleware), repositorios y componentes.

**Generar modelos “Diagrama de clases”**

Se va a construir de una forma muy elemental, en este caso se tendrán tres clases fundamentales, persona, pedido y producto (ver pantallazo).

En pedido se pide el id del producto para acceder a producto y saber qué producto se va a tomar, al igual se necesita el id persona (esto se verá más adelante).



Estas dos van a estar relacionadas en el futuro



Construir un modelo

1. lb4 model.
2. dar un nombre, en este caso **persona.**
3. Si se genera como entidad (va a ser para mapear hacia la base de datos). Si selecciono modelo, puede ser de manera temporal y no se va a mapear a la base de datos. En este caso se usa **entity.**
4. **No,** a las entidades de formato libre.
5. Dar atributos. Id 🡪string (los id son asignados por mongo de manera aleatoria), en este caso se tiene que seleccionar al propiedad id, YES, va a ser generado automáticamente, YES.

Va a preguntar por las siguientes propiedades, que son las que se encuentran en la clase persona. Nombre, apellidos, correo, celular, clave. Todas las propiedades nos preguntan si son necesarias, en este caso todas fueron necesarias.

1. Cuando se termina, se deja en vacío y se da enter
2. Si se necesita agregar otra cualidad o característica se puede hacer desde el modelo que fue creado, esto se puede hacer en el src en el modelo específico.

Este modelo crea un entity, model y property.

Crear los modelos para las siguientes clases y los repositorios.

Lb4 repositorio.

**14 de Octubre de 2022**

En esta clase se va a generar hasta las relaciones

Generar el otro modelo, Pedido y producto, entity, la cantidad y el total sí son de tipo numérico ya que es posible que se genere una operación en el futuro.

Las relaciones van a ser realizadas en el futuro.

De tener un error en la creación del modelo, este puede ser corregido en el nuevo módulo creado o se puede eliminar el modelo y crear uno nuevo.

**Generar repositorios**

Este es el encargado de realizar el mapeo a la base de datos (ya se deben tener los modelos creados), en dado caso que se agregue un nuevo modelo, se debe crear otro repositorio.

1. Lb4repository
2. Fuente de la base de datos: MongodbDatasource (es posible que se tengan varias bases de datos, para eso está la flecha)
3. Por cada una de las entidades se generaba un nuevo repositorio (ciclo anterior Rarbitro), se debe hacer igual
4. Seleccionar todos con el espacio y las flechas. Dar enter
5. Listo se crearon los repositorios.

Los repositorios tendrán la misma estructura sintáctica. 10:15

En este caso al revisar la estructura del repositorio creado se puede ver que este usa o importa ciertas carpetas.

Cuando se hacía el Dbset <Persona> Personas, se generaba una tabla, en este caso el repositorio hace esto. Con mongo no se crea una migración, mongodb hace el mapeo mucho más rápido.

**Generar relaciones**

Es necesario tener en cuenta el modelo de clases realizado.

1. Lb4 relation
2. Hay 4 tipos de relaciones

**belongsTo:** es cuando un dato pertenece a otra entidad (un pedido pertenece a una persona)

**hasMany:** cuando una entidad tiene mucho de otros

**hasManyYhrough:** es para realizar la relación de muchos a muchos donde surge una relación intermedia

**hasOne:** relación de uno a uno

En este caso vamos a usar el belongsTo y el hasMany.

Como el pedido pertenece a una persona, se selecciona el belongsTo (en clase se usa el hasOne, de quedar algo mal, cambiar el modelo a hasOne.) y se selecciona el modelo de origen (Pedido) y el modelo destino (Persona). En otras palabras un pedido pertenece a una persona.

1. Definir el nombre para la llave foránea, para este caso en particular se deja el que estaba por defecto (personaid).
2. Definir el nombre de la relación (persona).
3. Desea agregar información de la clase persona, como esa es la idea con la creación de esta relación se le da YES

Como se está creando una relación, en este caso NO SE CREA UN ARCHIVO DENTRO DEL PROYECTO, por lo tanto se modifica el *modelo* que se había creado anteriormente. PedidoModel ahora tendrá un nuevo campo.

Se crea la nueva relación, en este caso inversa, ***una persona tiene muchos (hasMany) pedidos.*** Cuando se crea una relación se crea un controlador, esto se puede ver desde la carpeta de ***controladores,*** en los controladores se pueden ver una serie de métodos (post, get, etc.)

En este caso el nombre de la llave foránea y de la relación se dejan como lo sugiere loopback. De esta forma ya se habrían creado los datasource, los modelos, los repositorios y las relaciones del modelo.

Cuando se crea una relación se actualiza el repositorio y de haber creado la conexión con información de dos modelos, se crea un controlador.

**17 de Octubre 2022**

Desde los métodos CRUD creados por el sistema, no se puede hacer todo lo que queremos realizar.

Creación de las relaciones con el producto y de pedido con persona, de pedido con persona (esto se hizo con los vídeos).

Creación de la relación de producto y pedido, HasMany

*Es importante seguir la estructura que se encuentra en la imagen de la clase del 12 de Octubre.*

**Generar los controladores**

1. lb4 controller
2. cada uno de los modelos debe tener el CRUD (crear, eliminar, actualización), en este caso se debe repetir la instrucción las veces que sea necesario, crear los controladores que sean necesarios de acuerdo a la cantidad de modelos o entidades del proyecto,
3. Dar el nombre de la clase que se va a generar.
4. Por ahora, no se va a generar un controlador vacío.
5. Qué modelo se va a utilizar (mismo modelo del paso 3).
6. Cuál es el repositorio que se creó para este modelo.
7. Cuál es la propiedad id
8. De qué tipo es la propiedad id
9. Se omite el id? **La base de datos lo crea, por lo tanto se usa YES para omitir.**
10. Cuál es la ruta de acceso. (URI BÁSICA – localhost:3000) se le da en YES.

Cuando se revisa el controlador ya existe el router, por lo tanto ya se encuentra un @ que nos muestra que ya se ha creado el router (esto se encuentra en el método post).

En este caso se crean todos los métodos CRUD. En clase se revisaron las funcionalidades de cada método creado en este controlador. Se ejecuta el lb4 controller

El método post tienen dos solicitudes, desde el navegador (solicitud inicial), **acceder al servidor,** respuesta de tipo 200.

***Npm start***

En este caso NO SE REALIZAN MIGRACIONES para mostrar información a la base datos, no son necesarias. Ya se puede ir al explorer, en este caso se hace el intento con el método post para persona, modificando la información del JSON y se le da EJECUTAR, **Ya se puede ejecutar el mongodbCompass y se va a ver que se creó la base de datos pedidos67 y tiene el dato que se envió.**

**Hasta aquí va el sprint 1 del proyecto + los modelos que se van a implemetar, por medio de modelos, correo con los roles del equipo y las invitaciones a cada uno de los repositorios. Desarrollo de los mockups para las vistas del frontend y el módulo de seguridad.**

**Documento, repositorio en el github (imagen ya sea en foto, pdf, etc, con el modelo de base de datos) dipamato02, tablero de Kanban.**

Twilio, nos da muchas limitaciones a nivel gratuito, solo permite enviar mensajes a solo un número de teléfono. Sendgrid, generar un correo a nivel de equipo.

**Conectar a sendgrid y twilio.**

**Con twilio se pudo crear una cuenta enviando un mensaje a otro celular, el número de teléfono es +13854062978, esto se creó desde la cuenta de Gmail afrs…**

1. Crear una cuenta de sendgrid
2. Logearse, y usar phyton para integrar el servicio.
3. Si no funciona sendgrid, intentar por medio del celular o otra dirección ip.

Para el final de la semana la aplicación tendrá construido hasta el momento del uso de microservicios de mensajes de texto y de envío de correos electrónicos. Estas son twilio (no se permite una gama alta de números telefónicos) y Sendgrid para el envío de correos electrónicos. Esto se encuentra en los vídeos del módulo.

Sendgrid API KEY – generación de un correo genérico.

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

1. Ir a web api y seleccionar que el servicio se va a seleccionar a través de phyton.
2. La codificación se encuentra en el momento en que selecciono phyton.

**Estructuración de twilio.**

Después de realizar la debida suscripción, se puede ver en los docs de usuario la forma para conectarse usando phyton. Se utilizará phyton para generar un **servidor** ya que se tiene facilidad en la codificación y no va a entrar en uso con las otras aplicaciones que se van a generar.

Para acceder a la consola de phyton se usa el anaconda navigator y se lanza el spider. En twillio se va a acceder a los documentos de desarrollador que se encuentran en twilio, tanto en twilio como en sendgrid, NO SE DEBE PONER INFORMACIÓN DE LA CUENTA, **como esto no se pone se deben codificar unas variables de entorno que conecten con la cuenta de twilio y sendgrid.**

1. Generar una carpeta dentro del proyecto de formación.
2. Guardar el proyecto dentro de la carpeta que se acaba de crear, usando el .py, ya que se trata de un archivo de phyton. Este va a ser VariablesEntorno.
3. Para generar las variables de entorno se tiene que llamar una librería.

Import **os**

Este paquete hace referencia al sistema operativo, esto para no tener que generar unas variables desde el sistema

1. Utilizar la clase environ, esta clase es la que permite guardar dentro de una variable el dato que yo quiero almacenar

Os.environ[“Prueba”]=”Este es un dato almacenado en una VE”

1. Crear un nuevo archivo de phyton en Spyder, posteriormente se guarda en la carpeta con el nombre de servidor. (SIMILAR AL EXPRESS – CREACIÓN DE UN SERVIDOR)

Un servidor es algo que permite ejecutar algo dentro del servidor

1. Importar el flask (esto ya está instalado), posteriormente se va a llamar el paquete Flask

from flask import Flask

1. Inicializar el Flask por medio de la variable app, esta tiene una estructura sintáctica para realizar la inicialización de la variable que usa un doble guión al piso al inicio y al final.

app=Flask(\_\_name\_\_)

1. Ahora hay que definir la ruta, en este caso la raíz.

@app.route(‘/’)

1. Definir la primera función. Traer la variable que quedó en las variables de entorno.

def uno ():

variable = os.environ.get("Prueba")

return variable

No olvidar que se debe importar el os.

1. Crear un if que identifica si el \_\_name\_\_ puede ser el main.

if \_\_name\_\_=='\_\_main\_\_':

app.run()

1. Se le da run a la aplicación y ya nos va a generar una conexión en el localhost en el puerto 5000
2. En este caso nos genera un error ya que la variable no se encuentra almacenada en el equipo.
3. Para que esto funcione también hay que poner a correr el archivo creado previamente de variables de entorno.

La funcionalidad de las variables de entorno es hacer funcionar el sendgrid y el twilio. Para hacer esta configuración, es copiar y pegar la información del twilio.

import os

from twilio.rest import Client

# Find your Account SID and Auth Token at twilio.com/console

# and set the environment variables. See http://twil.io/secure

account\_sid = os.environ['TWILIO\_ACCOUNT\_SID']

auth\_token = os.environ['TWILIO\_AUTH\_TOKEN']

client = Client(account\_sid, auth\_token)

message = client.messages \

.create(

body="Join Earth's mightiest heroes. Like Kevin Bacon.",

from\_='+15017122661',

to='+15558675310'

)

print(message.sid)

antes de hacer la implementación se debe crear una nueva ruta y seguir los siguientes pasos.

1. Crear la nueva ruta

@app.route(‘/sms’)

1. Crear una segunda función

Def dos():

1. Pegar el código de twilio (ver arriba) e identarlo.
2. La siguiente parte del código va donde se hacer la importación de las librerías.

from twilio.rest import Client

1. Construir las variables de entorno que pide twilio account\_sid y auth\_token
2. Copiar los identificadores que se encuentran en la consola de twilio a las variables de entorno y correr la parte correspondiente a las variables de entorno.

os.environ['TWILIO\_ACCOUNT\_SID']="ACc2f477a5c684b7022edc77aabc6a3190"

os.environ['TWILIO\_AUTH\_TOKEN']="\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\* "

1. Ahora se va a modificar el cuerpo del mensaje para hacer una personalización, para esto se va a hacer un try (es importante recordar que el try no viene con catch en py, en este caso viene con un except)

def dos():

try:

account\_sid = os.environ['TWILIO\_ACCOUNT\_SID']

auth\_token = os.environ['TWILIO\_AUTH\_TOKEN']

client = Client(account\_sid, auth\_token)

message = client.messages \

.create(

body="Join Earth's mightiest heroes. Like Kevin Bacon.",

from\_='+15017122661',

to='+15558675310'

)

print(message.sid)

return "El mensaje ha sido enviado exitosamente!"

except Exception as e:

return "No se pudo enviar el mensaje!"

1. Aún no se ha modificado el mensaje que va a ver el cliente, este se modifica como se muestra a continuación.

def dos():

try:

account\_sid = os.environ['TWILIO\_ACCOUNT\_SID']

auth\_token = os.environ['TWILIO\_AUTH\_TOKEN']

client = Client(account\_sid, auth\_token)

contenido = request.args.get("mensaje")

destino = request.args.get("telefono")

message = client.messages \

.create(

body=contenido,

from\_='+15017122661',

to=destino

)

print(message.sid)

return "El mensaje ha sido enviado exitosamente!"

except Exception as e:

return "No se pudo enviar el mensaje!"

1. Falta instalar el twilio (es una librería llamada anteriormente), para hacer esto hay que ir a anaconda y se envía un powershell, desde esta consola se realiza la instalación del paquete de twilio solicitado.

conda install -c conda-forge twilio

Para poder modificar el mensaje se tuvo que importar antes desde el flask la librería request

from flask import request

Con el postman, se prueba el get localhost:5000/sms

El request que se vio anteriormente permite capturar la información que se ve en el campo mensaje y el campo teléfono.

El twilio no se encuentra funcionando y esto puede ser porque falta la instalación del flask

conda install -c conda-forge flask-cors

**Octubre 19 de 2022**

En el git solo deberá aparecer la parte del backend para esta semana, por lo tanto del loopback + el diagrama de modelos.

**Estructuración del sendgrid**

Ir a email api y seleccionar integration guide y usar la API.

1. Ir al SPIDER, usar los dos documentos que se vieron en la parta anterior.
2. Generar otra ruta dentro del servidor de spider

@app.route(‘/e-mail’)

Def mail ():

1. Seguir los pasos del sendgrip (copiar el código).

message = Mail(

from\_email='from\_email@example.com',

to\_emails='to@example.com',

subject='Sending with Twilio SendGrid is Fun',

html\_content='<strong>and easy to do anywhere, even with Python</strong>')

try:

sg = SendGridAPIClient(os.environ.get('SENDGRID\_API\_KEY'))

response = sg.send(message)

print(response.status\_code)

print(response.body)

print(response.headers)

except Exception as e:

print(e.message)

Cuando se copia el código, es importante separar las importaciones que vienen incluidas.

1. Importar el paquete OS y los paquetes de sendgrip que se encuentran en el código (el os ya estaba instalado).
2. Hacer la instalación del paquete a través de anaconda, para esto se tienen que enviar una consola desde anaconda (powershell)

conda install -c conda-forge sendgrid.

1. Identar el código que fue copiado desde el sendgrid.
2. La función Mail viene incluida dentro de las librerías que fueron importadas, a continuación se tiene que modificar el correo “from email”.
3. Ir al archivo de las variables de entorno y colocar el nombre que se está usando en el main de phyton, aquí se usará la api key.
4. De no haberla guardado, se crea una desde el sendgrip (en la sección de crear una api en el paso 2) y asignar la variable con =
5. Crear el return después del último print

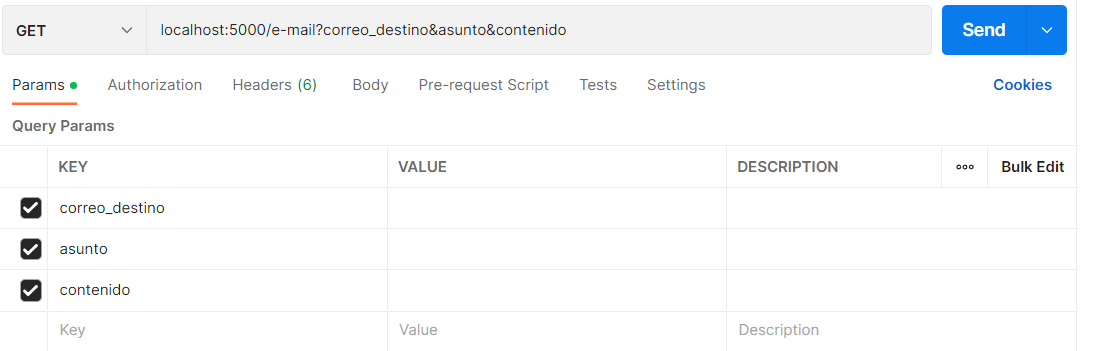
Return “correo enviado exitosamente”

1. En la excepción también se crea un return en caso de error

Return “Error! El correo no fue enviado ”.

Personalizar el mensaje que se va a enviar al cliente

1. Crear las variables necesarias para perzonalizar la información que va a ser enviada, esto quiere decir que se puede modificar el destinatario.
2. Ejecutar el postman.
3. Colocar la URL – localhost:5000/e-mail
4. Modificar el formulario con los datos que se van a pedir



1. Crear la variable *destino* por medio del uso de la clase request (esta se había importado anteriormente cuando se estaba haciendo uso de twilio)

destino = request.args.get(“correo\_destino”)

asunto = request.args.get(“asunto”)

mensaje=request.args.get(“contenido”)

1. Modificar la variable *mensaje,* con las variables destino, asunto y mensaje, según sea necesario.
2. Correr o ejecutar la aplicación.
3. Modificar el VALUE del postman.

El método get también permite hacer envíos, sin embargo, todo lo que se ha copiado queda enviado por la URL, en el futuro se va a hacer el cambio al método post ya que el método get para este tipo de casos no resulta seguro.

El spyder muestra una serie de datos que sendgrid envía

202 (significa que ha sido exitoso)

b''

Server: nginx

Date: Mon, 24 Oct 2022 02:46:52 GMT

Content-Length: 0

Connection: close

X-Message-Id: 5ZpX2QLqS1-ey7Eacx\_FUA

Access-Control-Allow-Origin: https://sendgrid.api-docs.io

Access-Control-Allow-Methods: POST

Access-Control-Allow-Headers: Authorization, Content-Type, On-behalf-of, x-sg-elas-acl

Access-Control-Max-Age: 600

X-No-CORS-Reason: https://sendgrid.com/docs/Classroom/Basics/API/cors.html

Strict-Transport-Security: max-age=600; includeSubDomains

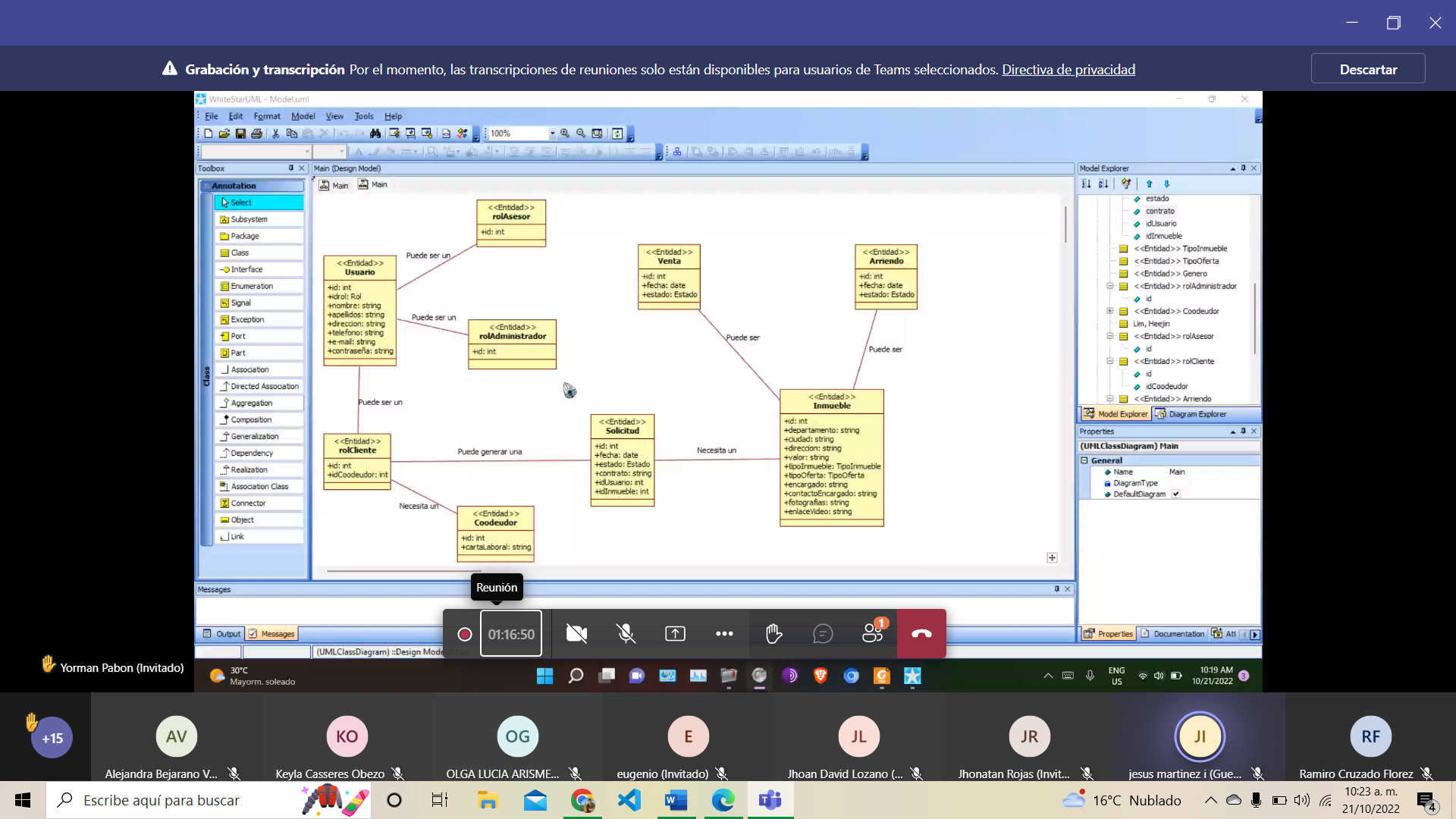
Este servidor va a ser consumido como un servicio, sin embargo hacen falta una serie de cosas en el backend para permitir el desarrollo total de la aplicación, esto va a estar relacionado con la seguridad de la aplicación.

Ahora se va a hacer la implementación por medio del LOOPBACK. Para hacer el consumo se tienen que hacer unas modificaciones dentro del controlador de persona (persona.controller.ts). No se puede permitir la creación de una persona hasta que yo valide que esos datos son correctos, por lo tanto, antes de hacer el uso del microservicio es importante tener la seguridad lista.

**Diagrama de modelos.**

Ninja mock 🡪 ayuda a generar unos mockups.

**21 de Octubre de 2022**



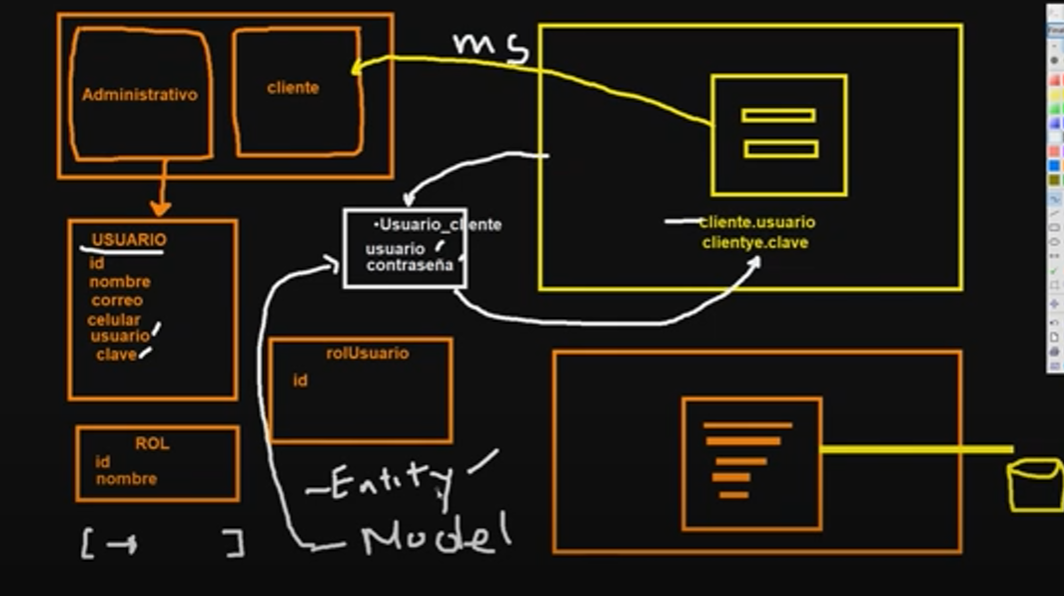
En este ciclo se va a combinar tanto el administrativo como el cliente, esto debido a que se está creando un módulo administrativo en el cual se gestionan roles, en este caso es mejor trabajar con una relación de muchos a muchos, esto genera una tabla adicional que se llama rol usuario.

El cliente debe ir aparte de todo lo administrativo ya que no va a tener permisos, él no puede hacer cambios dentro de la aplicación. Uno de los cambios es que el usuario no va a colocar la contraseña, yo la asignaré para ejercer la doble verificación de una mejor manera.

La imagen que se encuentra en la siguiente página muestra la forma de mejorar en algunos milisegundos, esto se parece al personaview que se creó con el antiguo profesor (muestra únicamente los datos o la información importante).

**Mongodb no genera herencias (esto es una herencia), por lo tanto se tiene que generar una tabla o una entidad para cada uno de los archivos.**

Es diferente un rol a una persona, el rol únicamente habla de permisos administrativos, a un cliente no se le puede entregar ningún permiso.



Siguiendo el ejemplo que se está creando en clase, el modelo de persona se asemeja al cliente en las aplicaciones REST que se deben crear, lo primero que se debe hacer es cambiar el requerido que se tenía en la clave por un false, ya que esta clave no la entregará el usuario y por el contrario será generada de manera automática por código.

  @property({

    type: 'string',

    required: false,

  })

  clave?: string;

Como se puede observar en el código se deja de hacer obligatorio el uso de la clave y se reafirma con el ? en la clave, para crear un objeto de tipo persona.

1. Cuando se va a generar el objeto (por medio del método post en el controlador).

Recordando las clases del ciclo pasado, se accedía al método post por un public, en este caso las api rest siempre son publicas, por lo tanto esto no va a ser necesario al momento de desarrollar el controlador.

 @post('/personas') // hace referencia al app.route

Como no se tiene la parte grafica en el loopback se encuentra la simulación con el response hasta antes del async.

  @response(200, {

    description: 'Persona model instance',

    content: {'application/json': {schema: getModelSchemaRef(Persona)}},

  })

Cuando se está usando el loopback se encuentra una función async, esta es una función asíncrona, esta se va ejecutando a medida que se va corriendo la aplicación (es decir sin haber creado toda la lógica de la programación).

  async create(

es análogo al public persona create(Persona persona)

Cuando se crean los controladores, estos automáticamente generan los métodos CRUD, sin embargo ninguno de estos métodos sirve para el proceso de autenticación.

Como se va a hacer un proceso de autenticación que no tiene o no está relacionado con los métodos CRUD, por lo tanto se van a crear unos métodos propios que se van a necesitar para toda la lógica de programación, lo primero que se va a generar es un método post

Este método POST vendrá desde el proceso de autenticación (login con el usuario y la contraseña) se accede a la raíz y se le va a enviar una respuesta por medio de el prompt identificar-persona, en este caso se tiene que enviar una respuesta automática

@post('/identificar-persona',{

  })

La respuesta automática se genera por medio del responses y se le da un parámetro 200, también se está creando una función asíncrona en la cual se va a solicitar las credenciales a la persona que se está logeando en la pg.

  @post('/identificar-persona',{

    responses: {

      '200':{

        description:"Identificacion de las personas"

      }

    }

  })

  async identificar (

    @requestBody() credenciales: //A través del formulario de login va a incluir usuario y contraseña

  ):Promise<Object>

En este momento la función no se encuentra terminada ya que antes de hacerla se debe crear un modelo desde la consola por medio de la función lb4 model que no va a estar relacionado con la base de datos (no va a enviar ningún tipo de información)

1. Lb4 model
2. Nombre de la clase modelo: Credenciales
3. Como no se necesita en la base de datos se crea de tipo **MODEL**
4. **NO** se asignan propiedades adicionales
5. **Asignar las propiedades,** en este caso va a ir USUARIO Y PASSWORD
6. **NO SE VAN A TENER IDS ya que no van relacionadas con la base de datos, pero las dos van a ser necesarias.**
7. COMO ESTO NO VA A IR MODELADO HACIA LA BASE DE DATOS NO ES NECESARIO CREAR LOS REPOSITORIOS NI CONTROLADORES (**HACE PARTE DE LA LÓGICA DE NEGOCIO**)

  @post('/identificar-persona',{

    responses: {

      '200':{

        description:"Identificacion de las personas"

      }

    }

  })

  async identificar (

    @requestBody() credenciales:Credenciales //A través del formulario de login va a incluir usuario y contraseña

):Promise<Object>

Credenciales

Hace referencia al modelo que se acaba de crear, esto crea una referencia en persona.controller.ts. Esto nunca se incluyó, pero loopback hace el proceso. En la promesa va a retornar un objeto.

Después del promise se crea una variable que se llame persona y se le asigna un await (no ejecuta las instrucciones hasta que reciba una respuesta), esto se hace para realizar la búsqueda por medio del método que se encuentra en el código en la parte de abajo, posteriormente se usa la palabra clave *where* para pedir el correo y la constraseña

let persona = await this.personaRepository.findOne((

where: {

correo:credenciales.usuario,

clave:credenciales.password

}

));

Return persona

El return es solo para mostrar gráficamente, el object va a ser cambiado por el tipo persona.

  @post('/Login',{

    responses:{

      '200':{

        description:"Identificacion de las personas"

      }

    }

  })

  async identificar(

    @requestBody() credenciales:Credenciales

  ):Promise<Persona | null>{

    let clavecifrada=this.servicioautenticacion.EncriptarPassword(credenciales.password);

    let persona = await this.personaRepository.findOne({

      where:{

        correo:credenciales.usuario,

        clave:clavecifrada

      }

    });

    return persona;

Así se ve finalmente, la clavecifrada y el correo fueron tratados en la siguiente clase 24 de Octubre.

Lo siguiente que se tiene que hacer es

1. Npm start
2. Ir al explorer y ver personaController – post identificar persona
3. Aquí solicita una persona y una contraseña, se crea un ejemplo de persona primero para que esto funcione.
4. Se coloca el correo (correspondiente al usuario) y la contraseña que se enviaron anteriormente.
5. Debe mostrar la información

Se va a utilizar el password generator y otro paquete para cifrar la clave

**24 de Octubre de 2022**

En la clase anterior se había trabajado en el método post para hacer el registro de persona. En nuestro caso, no vamos a pedir una contraseña al usuario, esta va a ser enviada por correo usando la verificación del correo electrónico.

Antes de enviar los datos de persona a la base de datos es necesario consumir un microservicio que genere las contraseñas, además estas deben ser cifradas ya que si alguien tiene acceso a las bases de datos tendría acceso a las contraseñas.

Para cifrar y enviar la contraseña se debe modificar el create de la persona (desde la parte gráfica), antes de que se envíe la información a la base de datos se le debe enviar la información al usuario, esto implica hacer el consumo de microservicios para cifrar la contraseña de los usuarios.

Los servicios se generan de la siguiente forma, estos no se generan de manera automática ya que algunas aplicaciones no necesitan servicios.

Hay tres tipos de consumir un servicio.

1. Proxy: como los de Google, consume datos de otra aplicación, incluso puede estar almacenado en la base de datos del tercero. Este es de tipo remoto
2. Local Service: vinculado al contexto de la aplicación, que usa información de otros lugares
3. Local Service: se consume de manera local, pero este va a consumir elementos que yo ya tenga en la base de datos

Hay que hacer el desarrollo de los servicios en la aplicación.

1. Ingresar a backendPedidios
2. Lb4 service
3. Hay que elegir el servicio que se va a implementar (ver los ejemplos anteriores), en este caso se va a usar la que yo mismo creo (2da opción).
4. Nombre del servicio: Autenticacion

Una vez se crea, ya se va a ver una capa llamada services, se crea un archivo que se puede modificar para implementar el servicio de autenticación.

Instalar dos paquetes específicos, el primero es el que genera la clave de manera automática donde no se conozca la contraseña. Este servicio también tiene su respectiva documentación.

<https://npm.io/package/generate-password>

npm install generate-password –save

Extraer el paquete en una variable, dentro de los servicios (autenticación.service.ts)

const generador = require(“generate-password”)

Crear los métodos para generar ese servicio

GenerarPassword (){

let password = generador.generate({

length: 8,

numbers: true

});

return password;

}

Ya teniendo la contraseña se debe pensar en la encriptación.

**Encriptación**

Como se debe cifrar la información, se hará uso de otro servicio llamado crypto-js

npm i crypto-js –save

Extraer el paquete en una variable

const crypto = require(“crypto-js”)

Crear los métodos para generar ese servicio, a este le debe entrar una contraseña en este caso se va a llamar password.

EncriptarPassword(password: string){

let passwordE = cryptoJS.MD5(password);

return passwordE

}

El paquete de crypto tiene muchos métodos de encriptación, esto se puede revisar en los documentos del cryptojs. Nosotros vamos a usar el md5 que se encuentra en esta función.

Esto se acaba de hacer de manera local, pero esto necesita ser consumido dentro del controlador de persona.

Antes de usar el archivo se tiene que hacer la importación, ver el paso a paso.

**Consumo del servicio desde el controlador de persona**

1. Importar el servicio de autenticación

**Import AutenticacionService**

1. Al igual que los repositorios, este servicio debe estar inicializado, en este caso se usa el controlador de persona y se agrega el

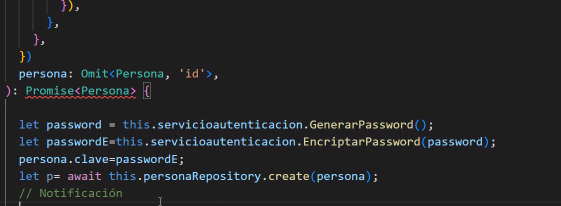
**@service (AutenticacionService)**

**Public servicioautenticacion:AutenticacionService**

dentro de la clase personacontroller.

Se está instanciando el objeto, en este caso se pidió hacer el import de otra clase del loopback

1. Después del promise persona



let password = this.servicioautenticacion.GenerarPassword ();

let passwordE = this.servicioautenticacion.EncriptarPassword (password);

persona.clave=passwordE;

let p= await this.personaRepository.create(persona);

1. Una vez ya se ha asignado la contraseña a la persona se debe incluir el proceso de notificación para enviar un correo con la contraseña y con la información necesaria, es necesario abrir anaconda y spider.

Como se va a hacer el proceso de notificación y esto se encuentra en spider y Python, hay que usar uso de un microservicio que se llama fet

let destino = p.correo;

let asunto = ‘Registro en la APP - Pedidos’

let contenido =`Hola, ${p.nombre}, su nombre de usuario es: ${p.correo} y su contraseña

     de accesso a nuestra app es: ${password}`;

Es necesario pensar en esto de forma dinámica, se le va a extraer a p (tiene toda la información de una persona) el correo, se manda un mensaje y que el contenido sea interactivo con la información que fue enviada.

String template, esto se hace con las comillas sencillas inversas como se ve en la variable let contenido. Esto cambia el uso de las concatenaciones y hace más fácil el proceso de codificación,

**EL STRING TEMPLATE SE HACE USANDO EL CÓDIGO ASCII USANDO EL ALT + 96**

Se usan , el símbolo $ y {}, de la siguiente manera

=`Hola, ${p.nombre}, su nombre de usuario es: ${p.correo} y su contraseña

     de accesso a nuestra app es: ${password}`;

Esto es un poco más dinámico y se usa unas comillas invertidas (grabación 8:57)



1. Consumir el servidor que fue creado en phyton, para esto se usa el fetch (hay dos formas de hacer la instalación).

npm install node-fetch

npm install node-fetch@2

al igual que las otras instalaciones este debe ser llamado por una variable, el segundo es el que funciona.

const fetch=require(“fetch”)

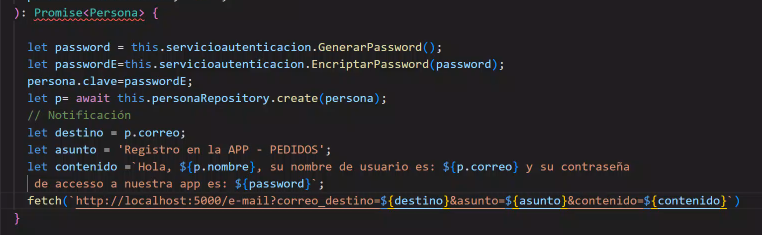
posteriormente se debe enviar la url al fetch

fetch (‘http://localhost:5000/e-mail?correo\_destino=${destino}&asunto=${asunto}&contenido=${contenido}’)

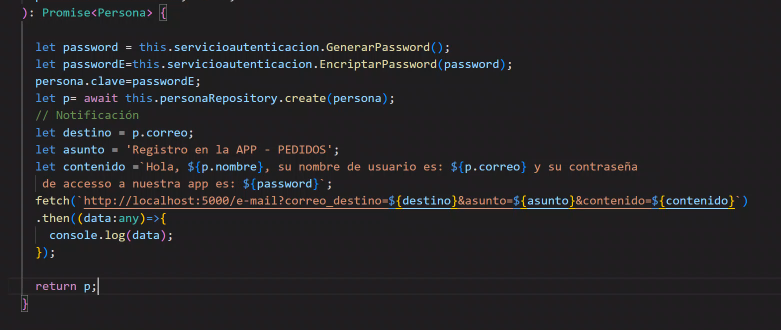
fetch(`http://localhost:5000/e-mail?correo\_destino=${destino}&asunto=${asunto}&contenido=${contenido}`)

El fetch va a llamar la información con la que se ejecuta la autenticación en Python, en este caso se había llamado a las variables como correo\_destino, destino y asunto y se les envía las constantes que se habían creado en la aplicación.

Se hace por un string template.



El método fetch, me permite consumir el servidor que se usó para enviar el correo, es parecido a un get.



El método then manda la respuesta cuando se ejecute el método, se complementa con una función flecha. El data:any corresponde a la evaluación de cualquier tipo de dato que sea enviado. El código completo se encuentra en la parte de abajo

    let password = this.servicioautenticacion.GenerarPassword();

    let passwordE=this.servicioautenticacion.EncriptarPassword(password);

    persona.clave=passwordE;

    let p= await this.personaRepository.create(persona);

    // Notificación

    let destino = p.correo;

    let asunto = 'Registro en la APP - PEDIDOS';

    let contenido =`Hola, ${p.nombre}, su nombre de usuario es: ${p.correo} y su contraseña

     de accesso a nuestra app es: ${password}`;

    fetch(`http://localhost:5000/e-mail?correo\_destino=${destino}&asunto=${asunto}&contenido=${contenido}`)

    .then((data:any)=>{

      console.log(data);

    });

    return p;

  }

console.log(data); Esto muesta el dato que fue encontrado.

**EL STRING TEMPLATE SE HACE USANDO EL CÓDIGO ASCII USANDO EL ALT + 96**

Generalmente al ejecutar se genera un error, esto se hace adrede para mostrar que falta correr el servidor phyton.

Cuando se desarrolla una api, únicamente se piensa en la lógica de negocio.

Una vez ya se tiene la información lista, se corre la aplicación con el npm start y se va al explorer para generar el método post en persona controller, se elimina la clave y el pedido. Se hace la verificación con un correo real.

En clase se generó un error debido a que el Python no estaba corriendo, siempre es importante correr el servidor local.

En este caso cuando se genera el registro la clave se genera de manera automática, pero no se va a ver dentro de la aplicación debido al crypto js. Si se revisa el correo que se usó se va a ver la contraseña que fue generada de manera automática.

Se va al post/login y se debería tener la opción de ver como si ya se hubiera ingresado, sin embargo, esto no es posible aún debido a que la contraseña ha sido encriptada anteriormente, en el futuro cuando se haga el CRUD, toca enviar la contraseña cifrada debido a que esta es la que está guardada la contraseña cifrada.

Para arreglar esto se crea una variable y se compara con las credenciales. Esto se hace dentro de la función de identificar que se creó anteriormente.

  @post('/Login',{

    responses:{

      '200':{

        description:"Identificacion de las personas"

      }

    }

  })

  async identificar(

    @requestBody() credenciales:Credenciales

  ):Promise<Persona | null>{

    let clavecifrada=this.servicioautenticacion.EncriptarPassword(credenciales.password);

    let persona = await this.personaRepository.findOne({

      where:{

        correo:credenciales.usuario,

        clave:clavecifrada

      }

    });

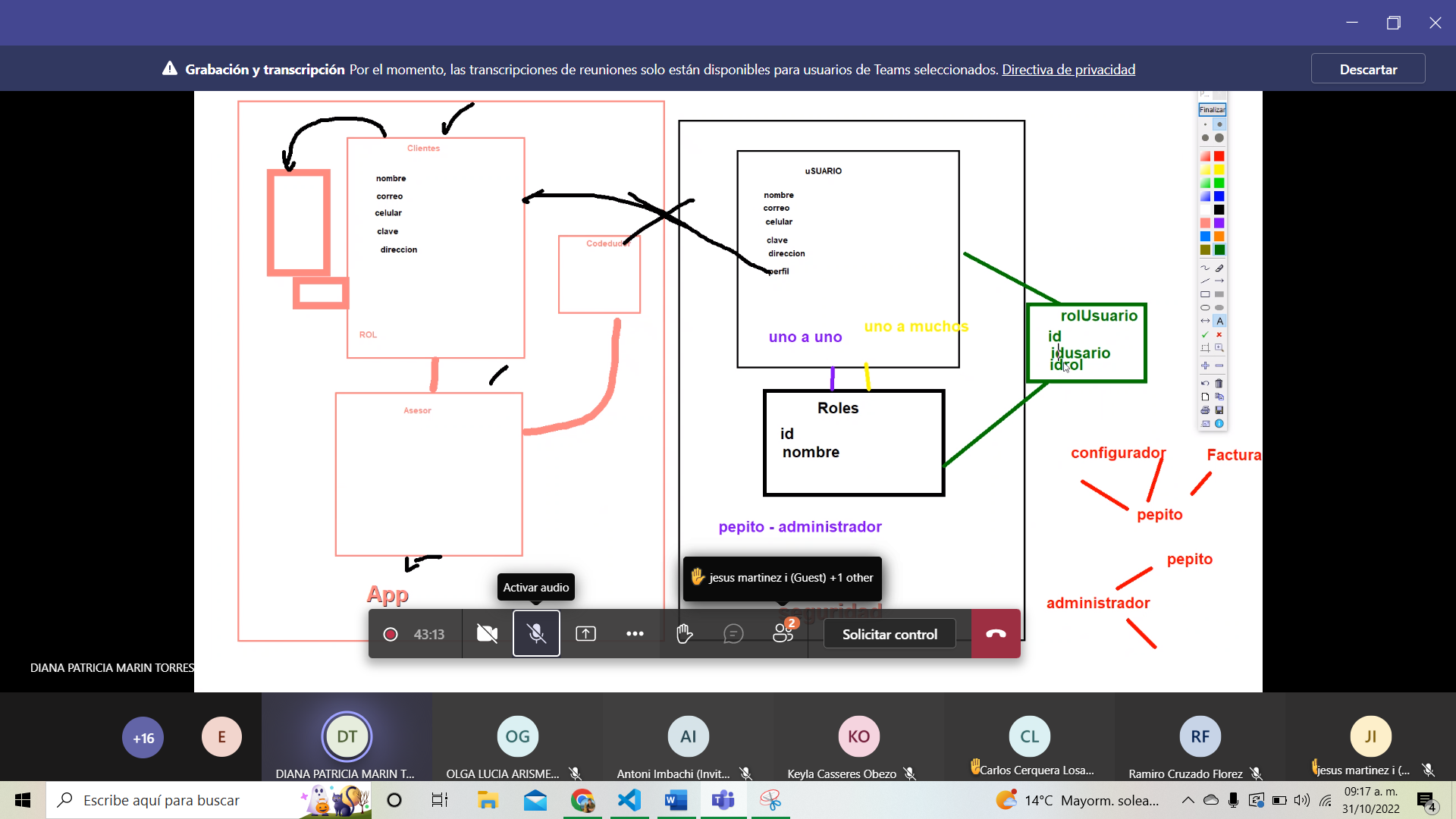
    return persona;

  }

Si se ejecuta la persona login va a funcionar con la clave que se envió por correo.

**26 de Octubre de 2022**

**31 de Octubre de 2022**



En la clase anterior se había generado el modelo de USUARIO, en este caso se va a colocar el perfil y se va a dejar de manera opcional. 9:20 am

De la clase persona se elimina el rol, el usuario va a tener la misma información de persona + el perfil.

Generar el controlador del usuario y del rol

Lb4 model rol usuario (solo va a tener el id por el momento)

Id

Sting, yes yes

Lb4 repository (rolusuario)

Lb4 relation

Hasmanythrough 🡪 relación de muchos a muchos a través de

Modelo de origen: persona

Destino: roles

Modelo intermedio: rolUsuario

Llave foránea: personaid

Rolesid

Roles

Yes

De manera automática se generan los atributos

Lb4 controller para usuario

Lb4 controller para rol

Lb4 para RolUsuario

El controlador de usuario va a estar encargado de recibir la información desde el frontend

En usuario se debe traer toda la información de la clave, ya que este va a ser el acceso a todos los usuarios del sistema.

Porque la relación de muchos a muchos se hizo de persona con rol y no con usuario. (Quedó mal porque la relación sí era con usuraios.

**02 de Octubre de 2022**

**Backend:** Token y verificación + estrategias.

**Frontend:** A nivel de código, un documento en el repositorio con extensión html con el formulario de registro, el de login y el de recuperación de contraseña.

Bootstrap, para el uso del html.

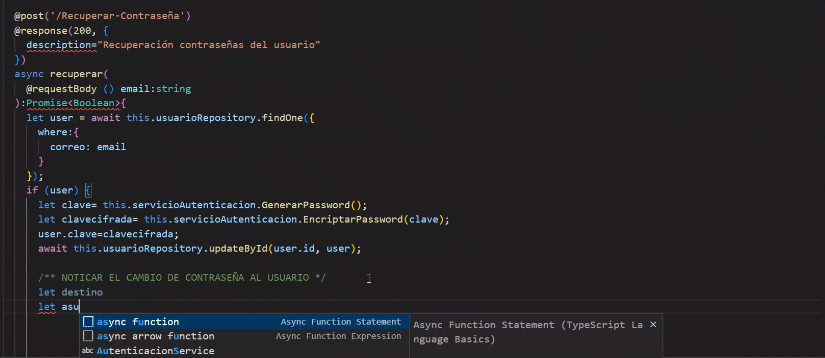
**Recuperación de contraseña**

Una vez que se conecte el usuario y quiera cambiar de contraseña, se tiene que capturar la información del usuario, esto siempre se hace por medio del método **post.**

Como se hace generalmente en otras pgs web, el usuario debe tener acceso al usuario (e-mail), por lo tanto, se hace un método asincrónico en el cual se solicite el usuario en string, posteriormente, se hace una respuesta o promise de tipo booleano, si es de tipo *true* se enviará la información correspondiente al correo del usuario.

Para revisar si el usuario está registrado se debe hacer la consulta a la base de datos y por ende, se ha de usar el método await, se debe acceder al repositorio de usuario y buscar la información que está mandando el usuario como email o como información de login, por lo tanto, se usa el método findOne donde (where) el correo corresponde al email. Se hace un if para verificar si el usuario existe o no, si existe se le envía la información al usuario. Como la contraseña se genera automáticamente en este if se le va a enviar el método generar password y posteriormente el método de encriptación.

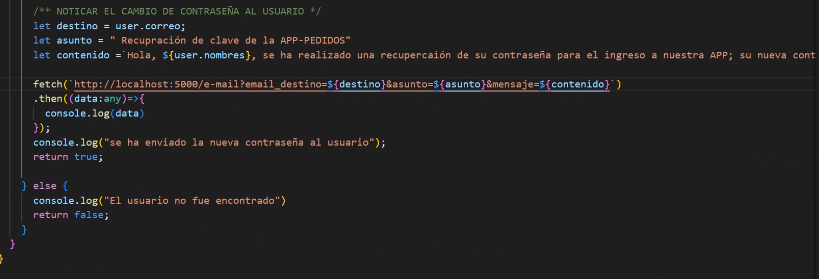
Como esta nueva información se tiene que enviar nuevamente a la base de datos, se crea otro método await para actualizar la información.



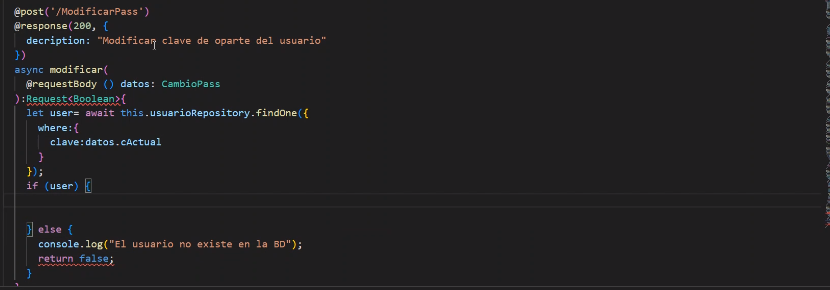
En el else que no se ve, se manda un mensaje por consola que muestra al usuario el error generado.

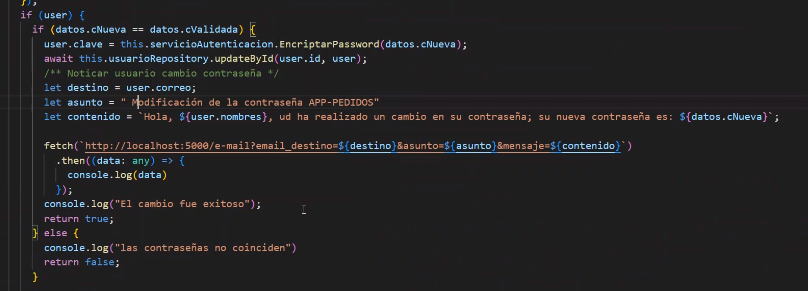
**Proceso de notificación al usuario**

El proceso de notificación consiste en enviar la información importante para el usuario por medio del correo electrónico, esto se hizo anteriormente y es importante tener el fetch instalado.



Generalmente cuando se hace el proceso de olvido de contraseña se permite generar una nueva contraseña, en este caso, se crea un nuevo método que se llama modificar contraseña, para esto debe entrar la constraseña actual, la contraseña nueva y la verificación, se puede pedir en el request o se puede crear un modelo (lb4 model – cambio pass) que se llama contraseña (que es como el tipo View que se vio en ciclos anteriores), como no va a estar mapeada a la base de datos a este modelo no se le genera un id y solo se envían tres datos correspondientes a las contraseñas solicitadas. dypjhe





Se corre la aplicación y se muestra va a modificar contraseña. En el caso de clase no funcionó por la encriptación.

Revisar grabación.

**4 de octubre de 2022**

Se hizo la implementación de las estrategias, se crea una estrategia llamada estrategiaAdmin.ts, y se hace otra para la estrategiaConfig.ts

Para asignar un rol, se tiene que tomar el id que aparece en mongodb del usuario y el id del rol, ir a la tabla rol usuario y asginar el rol y el id de la persona.