

>> back End Inicial << orig

TI Final → ABM

• Clase 1: Git

git → sistema de versionado

github → sitio que se basa en git

* Comandos básicos (terminal)

✓ cd + nombre carpeta → para moverse dentro de la estructura

✓ cd .. → nos lleva al nivel arriba

✓ ls → lista los archivos de la carpeta

✓ pwd → muestra la dirección completa donde estamos

✓ mkdir + nombre carpeta → crea una carpeta

Clave ssh → forma de autenticación

* Comandos básicos (github)

✓ `git clone` → para clonar un repositorio

✓ `git status` → avisa si hay cambios para subir

✓ `git add +` → agrega los cambios

Nombre archivo o `git add .` para agregar todos los archivos

✓ `git commit` → genera una nueva versión q/ el código actualizado

↳ `-m "Mensaje q/ el commit"`

✓ `git push` → sube los cambios al repositorio local

Clase 2 : Git Branches e intro a Node

* Más comandos de Git

✓ `git pull` → trae la ultima versión del repositorio remoto

`merge` → `automática` : lo soluciona git
↳ `manual` : lo tenemos q/ resolver nos

`branches` → linea temporal paralela al main

`git branch + nombre-del-branch`

✓ pull request → para informar a una branch master q' queremos mergear nuestros cambios

/ Para clonar:

- git clone + url del repo (ssh)

✓ Como crear una branch:

- git branch + nombre-branch → git branch coby-branch

✓ Para cambiar a una branch

- git checkout # nombre-branch → git checkout coby-branch

✓ Como hacer una pull request (desde github)

→ ir a pull requests

→ new pull request

• .gitignore → es un txt "gitignore"

dependencies

/ node_modules

NODE

Es un entorno de ejecución de JS.

Nos permite correr JS en el servidor.

single-thread : procesa 1 comando a la vez

asíncrono: Puede hacer otras tareas pero 1 a la vez
mientras le falte otra.

Otras Funcionalidades:

- * acceso al sistema de archivos

- * hacer HTTP requests

Para poder correr un archivo: node + archivo.js

Variables: almacenar y hacer referencia a otro valor

Se declaran con let → no permite redeclarar

var → se puede redeclarar su valor

const → no permite redeclarar

Siempre usar const pero si necesitamos cambiar el
valor podemos usar let

Qué puede ser?

números

"strings"

booleanos (true/false)

arrays → [1, 2, "tres"]

Objetos → const = {

full-name: "Constanza",

age: 36,

residence: "Añagro"

}



✓ Las variables pueden pasarse:

• Por valor: siempre representa el valor q' se le asigna en ese momento

• Por referencia: los valores q' le asignamos pueden modificarse

• Clase 3 : Condicionales y Bucles

✓ Condicionales

Para querer reaccionar ante una entrada a lo q reciba del usuario?

(A) if else : scope / AMBITO

GLOBAL : Funciona en todo el código.

(B) else if else

LOCAL : Funciona dentro de bloq de código solo

(A) if (condition) { se ejecuta este código}

else { se ejecute este código, m no cumple c/lo contrario}

(B)

if (Condition 1) { se ejecuta este código}

else if (Condition 2) { " " " " " m no cumple condition 1 }

else { se ejecute m no cumple ninguna }

Operadores de control → Para poder hacer condiciones
Son usados para controlar las condiciones dentro de la condición

$= =$ → igual a

\neq → distinto q'

PRUEBA EL VALOR

y además el tipo

$= = =$ → iguales } PRUEBA EL VALOR

$\neq \neq$ → distintos }

$<$ → si es menor } q' otro

$>$ → si es mayor } q' otro

\leq → menor e igual } q' otro

\geq → mayor e igual } q' otro

Oператоры логики

• operator & (and) → $U \& V$ → si ambas condiciones se cumplen.

• operator || (or) → si cumple una, es verdadero
si no cumple ninguna, es falso.

> Mayor que

✓ **Bucle**, → forma de ejecutar 1 acción vs veces a través del código.

• **For**: se repite hasta q la condición establecida se evalúa como false.

For (**inicio** ; **condición** ; **incremento**)
let i = 0 ; i < 10 ; i++)

• **While**: ejecuta sus sentencias mientras la condición sea cierta. → andan al lado

while (**condición**) { **sentencias** }

let i = -5

while (i < 10) { console.log (i) ; i++ }

• **for** (**in**) : iterar sobre los propios de un objeto

for (**variable** in **objeto**) { **sentencias** }

x = 2

for (let i = 0; i < 10; i++)

console.log (i)

lo que hace el q. 1

Funciones : Es un bloq' de código q' realiza 1 tarea

declarar : función nombrefunción (pará1, pará2, etc) {
 instrucciones }

return : finaliza la ejecución de una función

```
function sayHi(name) {  
    return "Mi nombre es " + name}
```

- ejecución : ~~sayHi~~ sayHi ("Loli")

- ámbito / scope: variables
 → global : pueden accederse desde cualquier lugar
 → local : no pueden ser accedidos fuera de la función

- expresiones : → anónima : `function () => {}`

→ const dollar = `function`

- Closures : func q' maneja variables independientes.
 se acuerda el ámbito donde se creó.

Ej. 1

```
function vowelCounter(arr) {  
    let count = 0  
    const VOCALES → declaración del  
    → contador  
    for (let i = 0; i < arr.length; i++) {  
        if (arr[i] === "a" || "e" || "i" || "o" || "u")  
            ↓ same c/cada vocal {  
                count += 1  
            }  
    }  
    return count  
}
```

• **Metodos** → funciones q realizan acciones
están asociados a un contexto en particular

/ tipos de datos:

(a) strings : son base de datos

/ string.length → propiedades

/ métodos:

• toUpper Case () : otra forma strings.toUpperCase()

(b) Arrays:

/ Bucle:

• nombre, forEach (function(elemento, indice) {
 console.log (elemento, indice); }) ;

• mjsP

→ Crea un nuevo array

OPTIONAL)

const newFruitas = frutas.map(function (elemento, indice) {
 return "Soy una " + elemento; }) ;

console.log (newFruitas);

✓ Añadir 1 elemento al final de un Array

const frutas = ["Naranja", "Banana"];

frutas.push ('Naranja');

• Eliminar elemento del final de un Array

frutas.pop ("Mango")

• Eliminar elemento del inicio

frutas.shift ("Mandarina")

• Añadir elemento al inicio

frutas.unshift ("Mandarina")

• Encontrar el indice de 1 elemento

frutas.indexOf ("Banana")

• Eliminar un elemento c/el indice

frutas.slice ("Banana", 1)

Puede usarlo para agregar elementos: frutas.slice(1, 0, "Naranja")

Posic donde se inserta ↑

Elemento elem
se borra ↓

nuevo elemento

agregar

• Copiar un Array

const newFruitas = frutas.slice();

• Convertir un Arra y en string

frutas.join(",")

④ Métodos de Strings

• Lee 1 lista de elementos

nombreStringable.split(" ")

• Devuelve el string en Mayúsculas

nombreVariable.toUpperCase();

• Devuelve el string en Minúsculas

nombreVariable.toLowerCase();

• Busca y reemplaza

Otro nombre Variable .replace ("Mi nombre es", "Soy")

• concat () Adjunta texto en una sola cadena de texto

• nth -ceil ()

① $y = \text{numero de dia}$

```

if ( $y == \text{numero}$ ) {
    console.log ("Lunes")
} else if ( $y == 1$ ) { console.log ("martes") }
else if ( $y == 2$ ) { console.log ("miércoles") }
else if ( $y == 3$ ) { console.log ("jueves") }
else if ( $y == 4$ ) { console.log ("viernes") }
else if ( $y == 5$ ) { console.log ("sábado") }
else if ( $y == 6$ ) { console.log ("domingo") }
else { console.log ("no ingresaste un numero") }

```

let

② ~~month~~ month = mes

~~if month == 1 { console.log ("Enero") }~~

```

if (month == "Enero" || month == "Marzo" || month == "Mayo" || month == "Julio" || month == "Agosto" || month == "Octubre" || month == "Diciembre") { console.log (month " tiene 31 dias") }

else if (month == "28" || month == "29" || month == "Febrero") { console.log (month " puede tener 28 o 29 dias") }

else if (month == "Abril" || month == "Junio" || month == "Septiembre" || month == "Noviembre") { console.log ("month " tiene 30 dias") }

```

Atencion

let num = numero

```
if (num < 0) {console.log("Es un numero +")}
else if (num > 0) {console.log("Es un numero -")}
else if (ingresaste num == 0) {console.log("Ingresaste 0")}
else {ingresaste mal la cosa"}
```

const notaNumeros =

```
if (notaNumber == "a" || notaNumber == "e" || notaNumber == "i" || notaNumber == "o" || notaNumber == "u") {console.log("Es una vocal")}
else if {console.log("Ingresaste una consonante")}
else {console.log("Ingresaste una vocal un numero seguido")}
```

const x =

y = 5

```
if (x > y) {console.log("x es mayor que y")}
else if (y > x) {console.log("y es mayor que x")}
```

NODE

* **modulos nativos** : paquetes de funcionalidades q' vienen c/node.

→ HTTP: api request (express)

* **modulos externos**: paquetes externos, se instalan

NPM: node package manager

* **NodeMon** : es servicio q' vigila los cambios en nuestro archivo. : open install -g nodemon

Se ejecuta con "nodemon + Nombre archivo.js"

* **cómo compartir información entre archivos**

2 formas

(a) **module.exports** es un objeto nos permite exportar funciones, variables y acc desde otro archivo

(b) **require** nos permite usar modulos (nativos, 3º party o propios)

a) module.exports

Any module → native de node : `module.exports = {
 person: {
 name: "Coco",
 age: 33, }},`

b) require → `const person = require("./file.js");
console.log(person);`

Javascript, utilizar módulos nativos

```
const fs = require("fs")  
const data = require("./person.js");  
  
fs.writeFile("name.txt", data.person.name, () => {  
  console.log("File created");  
});
```

Iniciáizar un proyecto

`npm init` → crea un package.json file

`npm init -y` → stepo de npm init acelera todo x defecto

```
const fs = require("fs")  
const randomData = require("./newfile.js") → saca us el module.exports
```

```
fs.writeFile("person.txt", "sara", () => {  
  console.log("Todo jóyd");});
```

Express: Sirve para crear web APPs y APIs

API (application programming interface)

frameworks y librerías → fuentes de código abierto
q' facilita el desarrollo

API (application programming interface) es una capa
q' nos permite conectar una PW/WEB APP q' nuestra base
de datos. La información se expone en formato JSON (js
object notation). Controla como se maneja la informacion de una
DB

HTTP: protocolo de comunicacion entre dos partes

→ métodos HTTP: convenciones de comunicación

* **get**: para pedir información (excepto: INF. SENSIBLE)

* **post**: para enviar información sensible (viajan ocultos
en el body de la request)

* **put**: para modificar información (= q' post)

* **delete**: para borrar información

→ Códigos de estado / status codes

: se usa para comunicar el
estado de los HTTP requests

- 10 x : informativas
- 20 x : éxito
- 30 x : redirecciones
- 40 x : errores del cliente
- 50 x : " del servidor

→ Rutas / endpoint: punto de entrada de nuestra app

* Estática

www.misitio.com

* Dinámica

Middleware: cualquier cosa q' ayuda para facilitar la comunicación entre el cliente y el servidor

En este caso: expresión similar de forma

en lo q' le pedimos a node q' realice (GET) Requests

objetos

* Req: (request) → Pedido de información

* Metodos:

Req: query

Req: params

* Res: (Response) → Respuesta / Entregar info

* Metodos: Res: send

Res: json

Res: status

6333616
HOJA N° 8
FECHA

const request = require('express');

const router = express.Router();

const app = express();

router.get('handle', (req, res) => {
 código a ejecutar
});

Para empezar a hacer un proyecto en express:

[npm express-generator nombre-projecto → no-view]



✓ Una vez creado, instalar los dependencias: npm install

✓ Para correr "node app": hay q ir al package.json y cambiar en "start" node x nodemon y desq "nodemon start"

[app.js] → el + importante. Es donde se hacen los llamados

index.html → se borra

Rutas

↳ **Dinámicas**: aceptan una posición variable en su estructura.

Con un solo endpoint podemos manejar múltiples casos.

res.send ('Hola usuario, \${req.params.name}');

Visible

```
router.get('/users/:name', function (req, res, next) {  
  res.send('Hola usuario: ${req.params.name}');  
});
```

- **Req. Params** → nos da acceso a la posición variable de una ruta dinámica. se obtiene desde el servidor

Parámetro adicional

? clave=valor & clave2=valor2

- **Req. Query** → nos da acceso a los parámetros que se envían a través de la url a la ruta. vienen desde el cliente

```
router.get('/delpromo/:name', function (req, res, next) {
```

console.log('req.params.', 'params');

 " ("req.query.", "query");

res.send(`\$ req.params.variable`); notar el punto)

[873]

- [874]

- [875]

routes / index.js → se tiene que estar el cor. mta.

↳ module exports = router;

Controllers → Poo en JS

↳ Clases: es un template para crear objs p' encapsular variables y logica para ser utilizadas dentro del controlador de la misma clase

declaro clase:

```
class NombreClase {  
    constructor (name, age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```

```
upUp () {  
    return this.age++  
}
```

```
}
```

↳ instanciar una clase: crear un objeto

```
const gato = new Animal ("Kerigan", 8)
```

↓

Palabra reservada

Herencia: class Cat extends Animal {

```
    constructor (name) {  
        super (name);  
    }  
}
```

```
speak () {  
    console.log(`' ${this.name} ' means: ' ${this.speak} '`)  
}  
}  
}
```

```
let Gato = new Cat("Alyx");  
Gato.speak();
```

✓ Animal → Clase
✓ Cat → subclase (clase que hereda de la clase base)
✓ Gato → Obj

Controllers → 1 capa extra para no tener toda nuestra lógica en el cuerpo de la ruta y tener un código + limpio
Se almacena toda la lógica de negocio de nuestra aplicación, luego se referencia

```
router.get('/', function(req, res, next) {  
    // referencia controller  
    res.send('sssss'))});
```

✓ Clases → Prog. orientada a objetos (P.O.O)

Es un paradigma que permite crear obj.

{ usar clase tipo recta implementar }

{ recta, etc }

- declarar Clase :

```
class NombreClase {  
    constructor (name, age) {  
        this.name = name;  
        this.age = age;  
    }  
}
```

```
ageUp () {  
    return this.age++  
}  
}  
}
```

Propiedades intrínsecas
no de clase

Método. Función asociada a la clase

- instanciar 1 clase; crea obj q la herita de ese clase
y le q le pasaron parámetros para crearse.

```
[const Gato = new NombreClase ("herman", 8)]
```

- ✓ Herencia : Cuando una clase hereda de otra recibe los props de su clase madre. Se le puede agregar props específicos.

Objetos

```

• class NuestraSubClase extends NombreClase {
    Constructor (name) {
        super (name);
    }
    speak () {
        console.log(`It's ${this.name} meows.`);
    }
}

```

↳ Prop de la clase madre

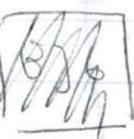
```

let Gato = new Cat ('Aby');
Gato.speak();

```

Animal → clase
 Cat → subclase
 Gato → objeto instanciado en Cat

This ⇒ hace referencia al Objeto de ese clase
 (nuestraSubClase)



Sintaxis Controllers:

- ① nueva carpeta "Controllers"
- ② abrir archivo controllers.js y declarar una clase:

```
class UserController {
```

```
getUsr (req, res) {
```

```
  console.log (this);
```

```
  res.send (`Hello ${req.params.name}`);
```

```
  module.exports = UserController;
```

NOTA

(3) Why a index.js w imports (& el archivo all router)

const userController = require ("./ ruta a userController.js") → Close

(4) creando la instancia

const userInstance = new userController(); → obj.

(5) referenciamos

router.get ("/users/:id", function (req, res, next) {
 userInstance.getOne (req, res);
});

module.exports = router;



HOJA N°
FECHA

APIs externas, ASYNC/AWAIT, Axios y spread operator

APIs se comunican entre ellos y se consumen con protocolo HTTP.
Lo q se consumen son endpoints externos.

- Axios: librería para hacer HTTP requests

- Usando Axios

```
const axios = require("axios") → ya está en el package.json  
como dependencia
```

```
class GithubUser {  
  getUsr (req, res) {  
    const data = axios.get(`https://api.github.com/users/${req.params.username}`)
```

```
    console.log(data);
```

```
    res.send(`No se logró obtener el request`)
```

```
}
```

```
}
```

```
module.exports = GithubUser;
```

Async/Await w/ Axios

async → adelante de la función

await → adelante de axios.get

Spread operator

• spread syntax → sirve para extender estructuras de datos iterables



Inmutabilidad en JS

• Crea un nuevo estado de algo. Una nueva versión con el cambio aplicado sin modificar el original.

const a = [1, 2, 3]

const b = a; → va a copiar x referencia

(copia el array de a x referencia)

= espacio de memoria (Aumenta el →)

espacio

strings ↴

valor se asigna
x valor

numeros ↴

array ↴
objetos ↴

valor se asigna a
referencia

const no se modifica el contenido del valor (ab 1 array
en 1 string)

spread operator genera un nuevo espacio en memoria
sin modificar el original

const a = [1, 2, 3];

const b = [...a, 5] → genera una copia de a
y la modifica agregando
5 pero a queda igual

Modelado de datos → tener un objeto con los "atributos"
de informacion segun nuestras necesidades

const axios = require("axios");

class UserController {

async getUsr (req, res) {

const data = await axios.get ("https://api.github.com/users/IMDoe")
*);

console.log (data);

const modeledData = {

Nombre: data.data.name,

Apellido: data.data.last_name;

};

res.json (modeledData); } }

[EJ 1] - [EJ 2] - [EJ 3]

Falta [EJ 0] y [EJ 4]

Centro de la mano → L 166. J 1 1690
máximo: 1690 mm

Centro de la mano → L 166. J 1 1690
máximo: 1690 mm

(2005) solapas + espesores
L 166. J 1 1690 mm

(dash) polimero

L = desplazamiento

polimero + tab + tab + madera

dash + dash + dash + dash + dash

Headers → INDICACIONES INFORMATIVAS q' puede trae nuestro request

Se reciben x el BE y podemos utilizarlas para generar condiciones

✓ Método Put

= q' Post pero c/ put

Postman → se usa aquí no hay Front

TP Final: ABM → ~~funciones de control o de negocio~~
mapear

post > esconde la info en el bodyオリジン API
put > ~~funciones de negocio~~ API

body → obj. JS

clave: valor

el lugar donde enviamos la información

Postman:

crea colecciónes de Request

- headers: es la info de la request

* desde Postman

- headers:

• more autogenerated headers

key : value

* desde el código

- ~~post~~

console.log (req.headers)

log. headers

Método delete

/ Se hacen deseños posteriores al browser solo
acepta "get"

/ es con req. Personas que no te quieren hacer dinámica

Mongo → databases

↳ database: es un conjunto de datos pertenecientes a 1 contexto y almacenados sistemáticamente para su posterior uso.

* Relacionales: almacena INF en tablas y luego crea relac entre ellos en base a Identificación (MySQL)

* NO Relacionales: guardan la info en un único registro (Mongo)

↳ mongo hasta h

→ Mongo

→ mongod -- dbpath=/Users/cohjar/Downloads/db

3. Estructura de datos

HOJA N°

FECHA

- ✓ Comandos:
- use nombre_database → para usar
 - db → me muestra la database donde estoy
 - db.dropDatabase() → elimina la db
 - show dbs → muestra las bases de datos existentes
 - db.createCollection('name') → para crear colecciones dentro de la base de datos
- db
mascotas
 |
 +--> gatos
 |
 +--> perros
 |
 +--> hamsters } collections
- db.name.drop() → borra la colección
 - db.name.insert({}) → agrega un registro nuevo
 - show collections → muestra las colecciones creadas
 - db.users.insert({ name: "Juan", nickname: "Gallo", edad: 27 })

✓ Quedan

- db.name.find() → devuelve un array c/ todos los registros de la collection name

Lobos3t Robot Robot 3 t

↓ Interfaz gráfica para manejar las bases de datos locales y externas

hay q tener corriendo mongo

lnea | ecrito | clona o clina base de datos locales

Mongoose: wrapper de mongo

↓ librería q se utiliza para conectar mongo en la Node.js

se instala en el proyecto q utilice mongo

comando npm install -- save mongoose

```
const mongoose = require ("mongoose");
```

```
mongoose.connect ("mongodb://localhost: 27017 / users", {
```

useNewUrlParser: true

useUnifiedTopology: true

```
});
```

se hace en bin/www.js

arranca

se ejecuta antes del http server

Models

- Un modelo de datos es donde vamos a escribir el plan de nuestra collection (q' field van a existir, q' tipo de dato van a contener)
- en mongoose se escriben ~~los~~ el modo de un "schema"
- Modelos date → darle un formato de acuerdo a los q' necesitamos

"schema" → Método de mongoose

testSchema = mongoose.Schema (

```
{ prop1: {
    type: String,
  },
```

```
anotherProp1: {
    type: Number,
    required: true
}
```

}

)

} contiene las
propiedades q' el
tipo de lo q' q'
quieras q'
tenga el obj. Model

module.exports = mongoose.model ('Test', testSchema);

- el nombre del model:
- ✓ en singular
 - ✓ en mayúscula la primera letra
 - ✓ es un string

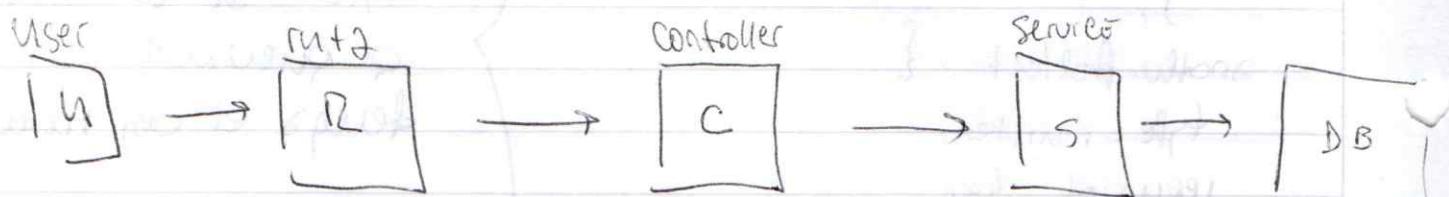
otros Modelos:

- /required: → si lo pones es obligatoria
- /default: → para poner un valor x defecto en caso de no tener la información
- /enum: → para tener nuestros propios tipos de datos.



Services → tiene métodos

- ↳ son el espacio de código donde viven todos los func q' interactúan directamente c/ la base de datos



```

class Service {
  getUsers() {
    const query = User.find();
    return query;
  }
}
  
```

module.exports = User.Service;

- En el controller → usamos el servicio

```
class UserController {
```

```
constructor(userService) {
```

```
    this.userService = userService
```

```
}
```

```
async getUsers(req, res) {
```

```
    const users = await this.userService.getUsers()
```

```
    return res.json(users)
```

```
    // establecer en la respuesta el tipo de contenido
```

```
}
```

```
}
```

- En el router → startus 1. añadir el servicio

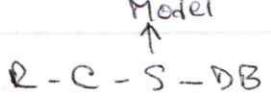
```
const userController = require('./userController');
```

```
const userService = require('./userService');
```

```
const userInstance = new Usercontroller(new userService);
```

Métodos Mongoose

- Model . find (filter) → Devuelve + o - mas c/ los otros
del el modelo
 - Model .findOne (filter) → Devuelve + unico obj en base
al rdo del filtro
 - findOne (filter) . exec ()
 - Model .findOneAndUpdate (filter, data) → Modifica
findOneAndUpdate . exec ()
unico obj en base
al rdo del filtro c/
el valor de data
 - Model . deleteOne (filter) → Borra una unica entrada q'
cumpla c/ la condicón del filtro
 - Model . save (data) → Crea + entraña en muestra db
c/ la dato provisto
- ```
addUser (data) {
 const newUser = new User (data);
 return newUser . save();
}
```



Ej 1, 1 y 2

## Patrones de diseño

### • Modelo (UserModel)

- ① requiere de mongoose: const mongoose = require("mongoose")
- ② el schema: const userSchema = mongoose.Schema {

```

 name: {
 type: string,
 required: true
 }
}

```

- ③ Model Prof. dicho:

```
module.exports = mongoose.model("User", userSchema)
```

### • User Service

- ① require el modelo. const User = require ("userModel")

```
class UserService {
```

```
 addUser (user) { console.log (user) }
```

```
module.exports = UserService
```

Se instancia en el controller en router

② cambio en route

30-2-2022

const userService = require('userService')

const userInstance = new userModel((new userService))

③ cambio en Controller

class userController {

AddUser

constructor (userService) {

this.userService = userService

}

addUser (req, res) {

console.log (req.body);

el servicio → this.userService.addUser (req.body)

res.send ('ok');

module.exports = userController

④ const User el modelo

const User = require ('userModel');

NOTA

```
const class userService {
 add user (user) {
 const newUser = new User (user)
 }
}
```

```
return newUser . save ()
}
}
```

```
module.exports = userService
```

Ej. API Usuarios

## ② Post a la db

Crea el endpoint en router:

```
router.post (" /add ", function (req , res , next) { })
```

Crea el controller:

```
class UserController {
 addUser (req , res) {
```

```
 console.log (req . body) //Acc. A un req
```

```
 res . send ("OK")
```

```
}
```

```
}
```

```
module.exports = UserController
```

Instancio el Controller en router:

```
✓ const userController = require("./UserController")
✓ const UserInstance = new UserController()
```

Claro → Tenemos de un obj

Obj → Conjunto de variables y metodos q' funcionan dentro de un mismo scope

```
router.post('/add', function(req, res, next) {
 UserInstance.addUser(req, res)}
```

Vamos a postman q lo probarmos

Para hacer el ingreso de la dato a la db, necesitamos el service pero primero hay q modelar la data q' quiero mapear c/ el Model:

```
const mongoose = require("mongoose")
```

```
const UserSchema = mongoose.schema({
```

name: {

type: string,

required: true

}

)

```
module.exports = mongoose.model("User", UserSchema)
```

- En userService, llamamos al modelo. Aca requerimos solo lo de el modelo necesario

```
const User = require("path of userModel")
```

```
class UserService {
```

```
addUser (name → lo q' recup. en modelo → user) {
```

```
console.log (user)
```

```
}
```

```
module.exports = UserService;
```

Deslina, para instanciarlo, se hacen 2 cambios:

- ① const userService = require("path of userService")  
const userInstance = new UserController (new userService())  
en routes

- ② class UserController {  
constructor (userService) {  
this.userService = userService  
}}

```
add User (req, res) {
```

```
console.log ("on")
```

```
this.userService.addUser (req.body) → llamada al servicio
```

```
res.send ("on")
```

```
}
```

Para conectarse a la base de datos desde el servicio

```
class UserService
```

```
 addUser (user) {
```

```
 const newUser = new User (new user)
```

```
 newUser.save ()
```

```
}
```

```
}
```

```
module.exports = UserService;
```

Si queremos ver q' responde le db al guardar una ent.

en el controller:

```
const response = this.userService.addUser (req.body)
```

tenemos q' usar async → en el metodo

await → en el promise

2.33.31

## Postman

(repositorio)

HOJA N°

FECHA

## ✓ Métodos HTTP

- **post**: crea info en memoria
- **put**: modifica información
- **delete**: borra información

## ✓ Código de estado

Para concatenar métodos.

res.status (código de estado) .json (sacaza)

### ① Método Post (sin controller)

```
router.post('/inscribir', function (req, res, next) {
```

```
const {body} = req; // req.body
```

```
console.log(body);
```

```
res.send('request realizada con éxito');
```

```
});
```

```
module.exports = routes;
```

Método Post (cf controller)

/ route.post (" / user ", function (req, res, next) {

UserInstance.postUser (req, res);

});

/ controller

postUser (req, res) {

const { body } = req;

console.log (body);

res.status (200). send (" usuário agregado com sucesso")

/ cf axios (FE)

const response = await axios.post ('/ user ', {

firstName: " Fred "

lastName: " Phastone "

)

console.log (response)