«Talento Tech»

Desarrollo Web 4

Clase 07

Buenos Aires Ciudad

Buenos Aires aprende
Agencia de Habilidades para el Futuro

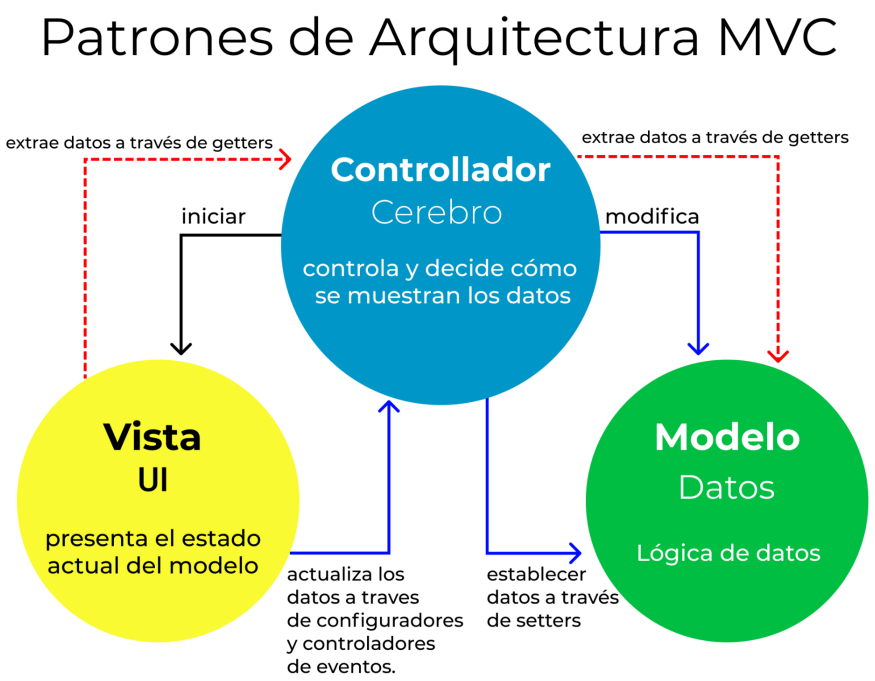
Clase N° 7 | MVC

Temario:

- Patrón de diseño MVC (Modelo, Vista, Controlador)

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Patrón de diseño MVC (Modelo, Vista, Controlador)



MVC, que significa **Modelo-Vista-Controlador**, es un patrón de diseño arquitectónico utilizado comúnmente en el desarrollo de software para organizar y estructurar el código de una manera que facilite la comprensión, mantenimiento y escalabilidad de una aplicación. Este patrón divide la aplicación en **tres componentes principales**, cada uno con una función específica:

Modelo

- Representa los datos y la lógica de negocio de la aplicación.
- Maneja la manipulación y gestión de datos, así como las reglas de negocio.
- No tiene conocimiento de la interfaz de usuario ni de cómo se presenta la información.

Vista

- Es responsable de la presentación de los datos al usuario y de la interacción del usuario.

- Muestra la información proveniente del Modelo.
- No realiza lógica de negocio ni manipula datos directamente.

Controlador

- Actúa como intermediario entre el Modelo y la Vista.
- Responde a las interacciones del usuario y actualiza el Modelo en consecuencia.
- También actualiza la Vista para reflejar los cambios en el Modelo.
- Encapsula la lógica de la aplicación y coordina las interacciones entre el Modelo y la Vista.

Flujo de Funcionamiento del MVC

- El usuario interactúa con la vista.
- El Controlador recibe estas interacciones y actúa en consecuencia.
- El Controlador actualiza el Modelo según sea necesario.
- La Vista se actualiza automáticamente para reflejar los cambios en el Modelo.
- Este ciclo se repite, proporcionando una separación clara de responsabilidades.

Ventajas de MVC

Separación de Responsabilidades
Cada componente tiene una función específica, lo que facilita el mantenimiento y la comprensión del código.

Reutilización de Código
Al dividir la aplicación en componentes, se puede reutilizar el código de manera eficiente.

Escalabilidad
Facilita la expansión y modificación de la aplicación sin afectar otras partes del sistema.

Facilita el Desarrollo en Equipo
Diferentes equipos pueden trabajar en paralelo en cada componente.

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Ejemplo aplicado a una aplicación web

MVC es un patrón ampliamente utilizado en el desarrollo de aplicaciones web y de software en general, proporcionando una estructura organizada y modular para los proyectos.

Modelo: Representa la estructura de datos y la lógica de negocios (por ejemplo, la base de datos y las operaciones CRUD).

Vista: Presenta la información al usuario (por ejemplo, la interfaz de usuario y las páginas web).

Controlador: Maneja las interacciones del usuario, actualiza el Modelo según sea necesario y actualiza la Vista (por ejemplo, controladores de rutas y lógica de manejo de solicitudes).

MVC usando Mongoose

Paso 1

```
import mongoose from "mongoose";
mongoose.connect("Acá tenes que colocar tu key de mongoose")
```

```
import express from "express";
import { postRoutes } from "./routes/post.js";
import mongoose from "mongoose";
const app = express ()
const PORT = 8080;
app.use(express.json())
app.use('/api/post', postRoutes)
mongoose.connect("Aca tenes que colocar tu key de mongoose")
app.listen(PORT, () =>{
    console.log(`Server listen on http://localhost:${PORT}/api/post`)
})
```

Paso 2

Para mayor seguridad podemos instalar la librería Dotenv. Esta librería nos permite manejar variables de entorno. Son muy importantes para el desarrollo de un servidor dado que manejamos mucha información que puede ser muy sensible y es bueno protegerla. Desde la consola, escribir el siguiente comando:

```
npm i dotenv
```

```
Hogar@DESKTOP-7D0F53V MINGW64 ~/Desktop/dsd-dw4 (master)
$ npm i dotenv

added 1 package, and audited 89 packages in 4s

9 packages are looking for funding
  run `npm fund` for details

found 0 vulnerabilities

Hogar@DESKTOP-7D0F53V MINGW64 ~/Desktop/dsd-dw4 (master)
$
```

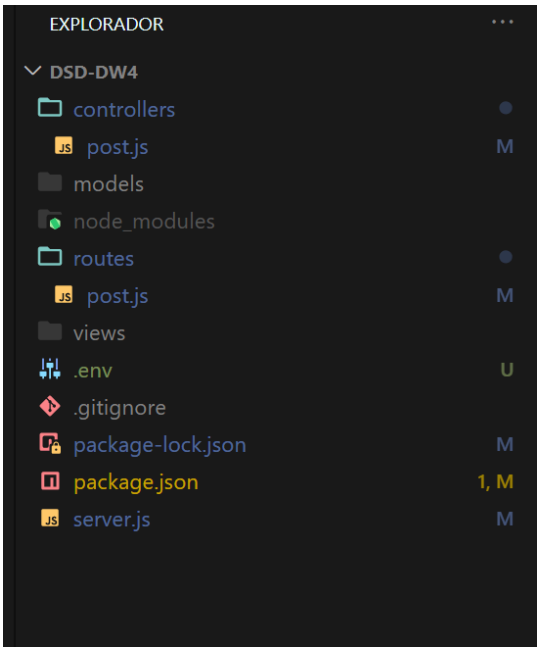
Paso 3

Importamos Dotenv en “server.js”

```
import express from "express";
import { postRoutes } from "./routes/post.js";
import mongoose from "mongoose";
import * as dotenv from 'dotenv'
dotenv.config()
```

Paso 4

Creamos un archivo “.env” a la misma altura que el archivo “server.js”



Paso 5

En el archivo ".env" vamos a escribir la contraseña de nuestra base de datos de Mongo Atlas de la siguiente manera:
Recorda que es importante que <username> lo reemplaces por tu nombre de usuario y <password> por la contraseña

De esta manera, el server.js quedaría así:

```
import express from "express";
import { postRoutes } from "./routes/post.js";
import mongoose from "mongoose";
import * as dotenv from 'dotenv'
dotenv.config()
const app = express ()
const PORT = 8080;
app.use(express.json())
app.use('/api/post', postRoutes)
async function main(){
```

```
    await mongoose.connect(process.env.DB);
}
main()
app.listen(PORT, () =>{
    console.log(`Server listen on http://localhost:${PORT}/api/post`)
})
```

Paso 6

Ahora tenemos que proceder a crear nuestro modelo que va a ser el molde para poder almacenar datos en nuestra db. Lo vamos a crear dentro de la carpeta models:

```
import mongoose from 'mongoose';
const {Schema} = mongoose;
const PostSchema = new Schema({
    title: String,
    author: String,
    body: String,
    date:{type:Date, default:Date.now},
    hidden: Boolean,
})
export const Post = mongoose.model('Posts', PostSchema)
```

Es muy importante que coloquemos los campos con su correspondiente tipado (tipo de dato) de esta forma podemos ya tener la base para poder crear un post. Solo tenemos que trabajar en el controlador.

Paso 7

Por último tenemos que crear una variable llamada POST la cual nos servirá para avisarle a mongo atlas que queremos crear una colección llamada “Posts” que va a tener el Schema PostSchema.

Primero lo que vamos a hacer es importar en nuestro controlador el Schema (esquema) que creamos en la carpeta Models:

```
import {PostSchema} from "../models/post"
```

Paso 8

Luego en nuestro post lo que vamos a hacer es hacer lo siguiente: para trabajar con Mongoose tenemos que convertir todas las funciones del controlador en asíncronas dado que la petición a las bases de datos siempre se realizan de ésta forma.

```
export const postPost = async (req, res) => {
   const body = req.body
   const newPost = await new Post(body)
   try {
      const savePost = await newPost.save()
      res.status(200).json({savePost});
   }catch (error){
      res.status(400).send(error)
   }
}
```

Lo que hicimos es capturar el body a través del primer parámetro req, luego creamos un nuevo post y le pasamos por parámetro lo que capturamos en el body.

Luego mediante un “try & catch" indicamos que sí está todo ok, que se guarde y que devuelva un mensaje 200 junto con toda la información en Json. Caso contrario, que nos devuelva un “Error 400”.

De esta forma, las, funciones Get post,Update Post y Delete, quedarán de la siguiente manera:

Get post

```
export const getPosts = async (req,res) =>{
   const getPosts = await postPost.find()
   res.json(getPosts)
}
```

```
export const getPost = async (req, res) =>{
    const id = req.params.id
    const findPost = await Post.findById({_id : id});
    if(!findPost){
        res.status(404).json({msg : "No se pudo encontrar el post"})
        return
    }
    res.json(findPost)
}
```

Update Post:

Acá se busca al post por id , si no lo encuentra manda un “Error 404” via Json y sino continua en base a lo que queramos cambiar va a actualizar lo que le pedimos a través del body o sino va a dejar lo que tenía por defecto en el caso que no queramos actualizarlo.

```
export const updatePost = async (req, res) =>{
    const id = req.params.id
    const findPost = await Post.findById({_id : id});
    if(!findPost){
        const error = new Error("Post no encontrado");
        return res.send(404).json({msg: error.message});
    }
    findPost.title = req.body.title || findPost.title;
    findPost.author = req.body.author || findPost.author;
    findPost.body = req.body.body || findPost.body;
    try{
        const savePost = await Post.save();
        res.json(savePost);
    } catch(error){
        console.log(error.message);
    }
}
```

Buenos Aires aprende
Agencia de Habilidades para el Futuro

Delete

```
export const updatePost = async (req, res) =>{
    const id = req.params.id
    const findPost = await Post.findById({_id : id});
    if(!findPost){
        const error = new Error("Post no encontrado");
        return res.send(404).json({msg: error.message});
    }
    try{
        await Post.findByIdAndDelete(id);
        res.json({msg: "findPost eliminado correctamente"});
    } catch(error){
        res.status(500).send(error)
    }
}
```

¿Te gustó el contenido visto hasta ahora?

¡Nos vemos en la próxima clase!


Buenos Aires
aprende
Buenos
Aires
Ciudad