Laboratorio 06: Arquitectura MVC con Node.js y Express

Introducción

En el desarrollo de aplicaciones web, una de las mayores preocupaciones es cómo organizar el código de forma que sea mantenible, escalable y fácil de entender. Aquí es donde el patrón **MVC** (Modelo-Vista-Controlador) juega un papel fundamental. Este patrón de diseño divide una aplicación en tres componentes principales, cada uno con responsabilidades claras: el **Modelo**, la **Vista** y el **Controlador**.

¿Qué es Express?

Express es un framework minimalista para Node.js que permite crear aplicaciones web y APIs de manera sencilla y eficiente. Es conocido por su flexibilidad, alto rendimiento y una gran comunidad de desarrolladores, lo que lo convierte en una herramienta popular para proyectos tanto pequeños como grandes.

¿Qué es el patrón MVC?

MVC (Modelo-Vista-Controlador) es un patrón de diseño de software que divide una aplicación en tres partes bien definidas:

- **Modelo**: Gestiona la lógica de datos y la interacción con la base de datos o la fuente de información.
- **Vista**: Se encarga de mostrar la información al usuario y define cómo se presenta la interfaz.
- **Controlador**: Actúa como intermediario entre el Modelo y la Vista. Recibe las solicitudes del usuario, las procesa utilizando el Modelo, y luego devuelve la información adecuada a la Vista.

Configurar el Proyecto

Para iniciar, utilizaremos el comando `npm init -y`, que nos permite inicializar rápidamente un proyecto en Node.js. **npm** (Node Package Manager) es el gestor de paquetes de Node.js, y el flag `-y` acepta automáticamente los valores predeterminados para la configuración del proyecto, generando el archivo `package.json` sin necesidad de responder las preguntas habituales, lo que agiliza el proceso inicial.

Además, instalaremos nuestro framework **Express y nuestro motor de plantillas para la renderizacion de las vistas**

```
npm init -y
npm install express
npm install pug
```

Estructura del proyecto

```
LABORATORIO-06/
│
├── node_modules/
├── src/
│   ├── controllers/
│   │   └── user.controller.js
│   ├── models/
│   │   └── user.model.js
│   ├── routes/
│   │   └── user.route.js
│   ├── services/
│   │   └── user.servie.js
│   ├── views/
├── app.js
├── .gitignore
├── package.json
├── package-lock.json
```

👀 Para esta práctica, se proporcionarán las vistas (vistas y estilos). Por favor, descarguen el archivo comprimido llamado `views` , extráiganlo y colóquenlo en la ubicación de su archivo `src`.

Creación de modelo

trabajaremos en el archivo `user.model.js`

1. Definición del Modelo `User`

Comencemos por definir el modelo `User`. Este modelo representa a un usuario en nuestra aplicación y contiene propiedades como `name`, `lastname` y `email`. La idea es que encapsule toda la información relevante de un usuario.

```
class User {
    constructor(name, lastname, email) {
        this.id = Date.now().toString();
        this.name = name;
        this.lastname = lastname;
        this.email = email;
    }
}
```

2. Definición de funciones para este modelo

Para gestionar la información de los usuarios en nuestra aplicación, hemos definido varias funciones en el modelo `User`. El arreglo `users` actúa como una base de datos en memoria que almacena instancias de la clase `User`. La función `save(user)` permite agregar un nuevo usuario al arreglo, tomando un objeto `user` como parámetro y utilizando el método `push` para almacenarlo ademas se retorna el usuario almacenado.

```
const users = [];

const save = (user) => {
    users.push(user);
```

```
    return user;
}

module.exports = {
    User,
    sav

};
```

Creación de servicios

trabajaremos en el archivo `user.service.js`

En esta sección, crearemos el apartado de **servicios**, que nos ayudará a manejar la lógica de negocios de nuestra aplicación. Estos servicios utilizarán los métodos de nuestro modelo existente, lo que nos permitirá almacenar y gestionar los datos del usuario de manera eficiente.

Implementación de servicios:

- **Servicio para agregar un nuevo usuario**: Este servicio se encargará de recibir la información del nuevo usuario y enviarla al modelo correspondiente para que sea almacenada en la base de datos. Además, devolverá el usuario ya almacenado.

```
const {save, User} = require('../models/user.model');

const addNewUser = (name, lastname, email)=>{
    const newUser = new User(name, lastname, email);
    return save(newUser);
}

module.exports = {
    addNewUser
}
```

Creación de controladores

trabajaremos en el archivo `user.controller.js`

En esta sección, definiremos los **controladores** de nuestra aplicación, que se encargarán de gestionar las solicitudes del cliente y de interactuar con los servicios para realizar las operaciones necesarias. Los controladores son responsables de procesar la lógica de negocio, gestionar las respuestas y renderizar las vistas.

Implementación de controladores:

- **Controlador para agregar un nuevo usuario**: Este controlador se encargará de manejar la creación de un nuevo usuario, renderizando la vista correspondiente y enviando los datos necesarios para su presentación.
- **Controlador para mostrar un formulario**: Este controlador renderizará un formulario donde los usuarios podrán ingresar su información.

```
const {addNewUser} = require('../services/user.servie');

const addUserController = (req, res)=>{
    try{
        const {name, lastname, email} = req.body;
        const user = addNewUser(name, lastname, email)
        res.render('home.pug', {user});
    }catch(e){
        console.log(e);
        res.status(500).send('Internal Server Error');
    }
}

const showForm = (req, res) => {
    res.render('form.pug');

}


module.exports={
    addUserController,
```

```
    showForm
}
```

Configuración de rutas

trabajaremos en el archivo `user.route.js`

En esta sección, nos enfocaremos en la creación de **rutas** dentro de nuestra aplicación Express. Las rutas son esenciales para gestionar las solicitudes HTTP y definir cómo la aplicación responde a diferentes endpoints. Utilizaremos controladores para manejar la lógica de negocio asociada con cada ruta.

En esta implementación, definiremos la estructura de las rutas que permitirá acceder a nuestras funciones, especificando la extensión que debe tener cada ruta y el método HTTP correspondiente. También se pueden añadir middlewares para realizar validaciones y otras funcionalidades necesarias, en este caso solo aremos uso de nuestro controlador.

```
const { addUserController, showForm} = require('../controller
const router = require('express').Router();
router.get('/', showForm);
router.post('/home', addUserController)

module.exports = router;
```

Configuración de la aplicacion

En esta sección, aprenderemos a configurar el archivo principal de nuestra aplicación Express, donde estableceremos middleware, vistas y rutas.

```
const express = require('express');
const userRoute = require('./routes/user.route');
const path = require('path');

const app = express();

// Establecemos Pug como motor de plantillas
```

```
app.set('view engine', 'pug');
// Especificamos la carpeta donde se encuentran las vistas
app.set('views', path.join(__dirname, 'src/views'));

// Middleware para parsear datos de formularios
app.use(express.urlencoded({ extended: true }));

// Usamos las rutas de usuario
app.use(userRoute);

// Configuración del puerto
const PORT = 3000;

// Iniciamos el servidor
app.listen(PORT, () => {
    console.log(`Server is running on port ${PORT}`);
});
```

Explicacion

Este es el punto de partida para construir una aplicación web utilizando Express. Proporciona un objeto que permite configurar middleware, definir rutas, gestionar errores y ejecutar el servidor, facilitando el desarrollo de aplicaciones web robustas y escalables.

```
const app = express();
```

Estas dos lineas configuran Pug como el motor de plantillas de tu aplicación Express y especifican dónde se encuentran los archivos de plantilla. Esto permite que la aplicación renderice vistas de forma dinámica y organizada, mejorando la experiencia de desarrollo y la calidad del código.

```
app.set('view engine', 'pug');
app.set('views', path.join(__dirname, 'arc/views'));
```

Esta linea nos permite manejar correctamente los datos de formularios en una aplicación Express. Permite que tu aplicación interprete los datos enviados por los usuarios, facilitando la interacción y la recolección de información en aplicaciones web. Sin este middleware, no podrías acceder a los datos de los formularios.

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

La linea `app.use(userRoute);` se utiliza para registrar las rutas definidas en el módulo `userRoute` dentro de la aplicación Express. Al hacerlo, todas las solicitudes que coincidan con las rutas especificadas en `userRoute` se gestionarán mediante los controladores y la lógica definida en ese módulo

```
// Usamos las rutas de usuario
app.use(userRoute);
```

La función `app.listen(PORT, () => { ... });` se utiliza para iniciar el servidor Express y hacer que escuche las solicitudes en el puerto especificado por la constante `PORT`, contiene un callback que se pasa como segundo argumento se ejecuta una vez que el servidor está en funcionamiento. En este caso, simplemente se imprime un mensaje en la consola que indica que el servidor está activo y en ejecución, junto con el número del puerto en el que está escuchando.

```
app.listen(PORT, () => {
    console.log(`Server is running on port ${PORT}`);
});
```

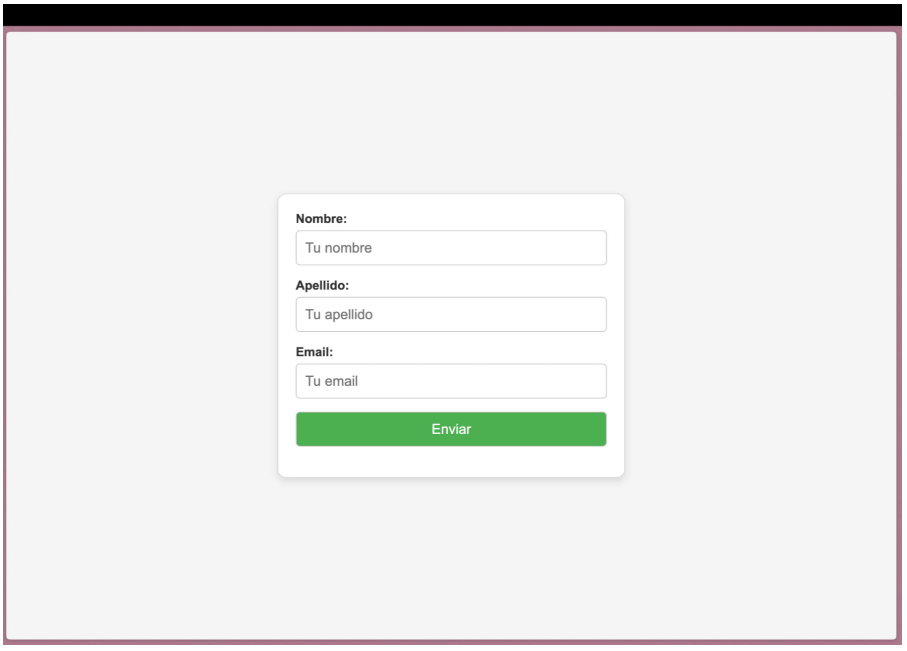
Ejecución

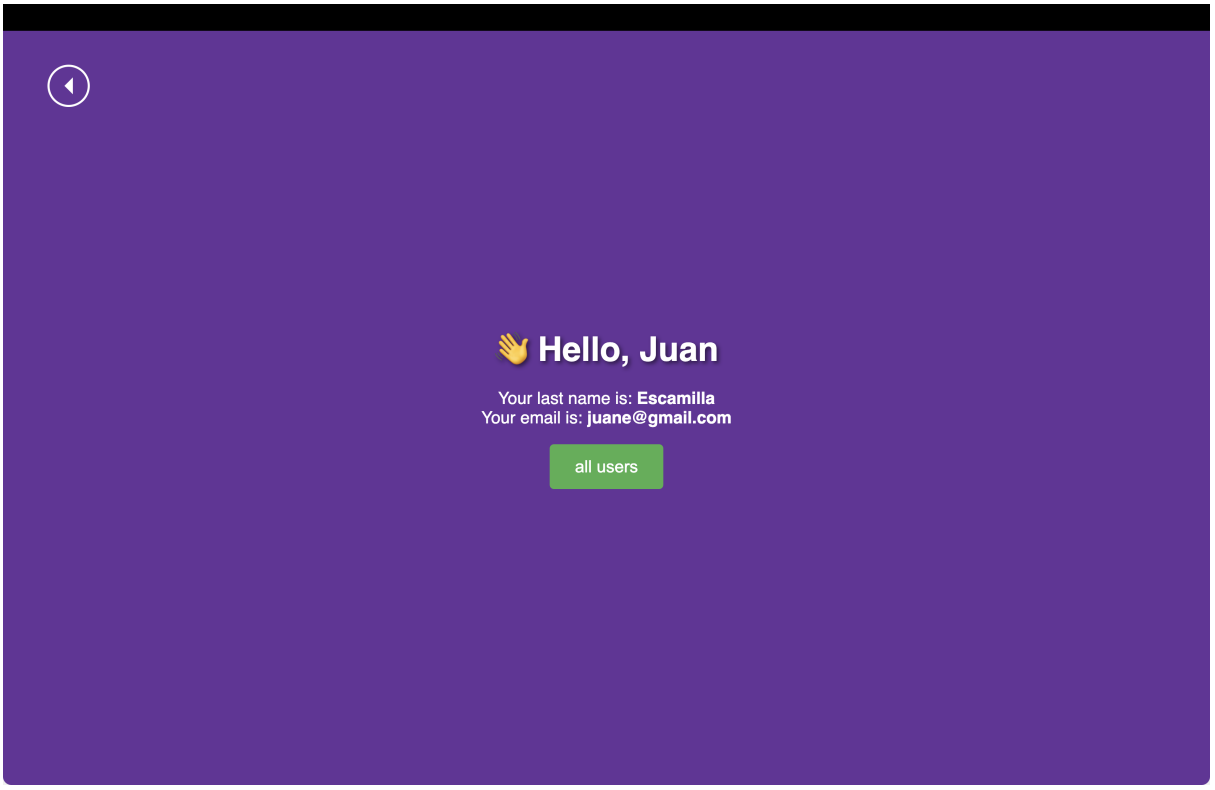
Para iniciar nuestro servidor, debemos ejecutar el siguiente comando, reemplazando el nombre del archivo principal, que en este caso es `app`:

```
node app.js
```

👀 Es importante tener en cuenta que los cambios realizados en el código no se aplicarán automáticamente, por lo que cada vez que se hagan modificaciones, será necesario reiniciar el servidor ejecutando nuevamente el comando anterior.

Resultado esperado.





Ejercicio de tarea

El botón destinado a visualizar todos los usuarios llevará al usuario a una nueva vista en la que se mostrarán todos los usuarios registrados en nuestra base de datos. Para implementar esta funcionalidad, se deberán seguir los siguientes pasos:

1. **Creación de Función en el Modelo**: Desarrollar una función en el modelo de datos que se encargue de recuperar y devolver la información de todos los usuarios registrados.
2. **Implementación de un Servicio**: Crear un servicio que se encargue de invocar la función del modelo y obtener los datos de los usuarios. Este servicio será responsable de comunicarse con la base de datos y devolver los datos necesarios.
3. **Desarrollo del Controlador**: Implementar un controlador que gestione la lógica de la nueva vista. Este controlador debe incluir un bloque `try-catch` para el manejo de errores, garantizando así un control adecuado sobre

cualquier posible fallo durante la recuperación de los datos. En caso de éxito, el controlador renderizará la nueva vista y enviará los datos de los usuarios a esta.

4. **Definición de la Ruta**: Configurar la ruta correspondiente a esta funcionalidad como `/allUsers`, que será la URL a la que se dirigirá el botón mencionado.

El resultado esperado de este proceso es que, al hacer clic en el botón para ver todos los usuarios, se cargue una nueva vista que muestre una lista completa de los usuarios registrados, con manejo adecuado de errores y una interfaz clara para el usuario.