

UNIVERSIDAD DE SAN CARLOS DE GUATEMALA
CENTRO UNIVERSITARIO DE OCCIDENTE

MANEJO IMPLEMENTACIÓN DE ARCHIVOS

ING. PEDRO DOMING

AUX. FERNANDO OCAÑA

MANUAL DE USUARIO

MARCOS ANDRÉS AGUARE BRAVO

20183269

INTRODUCCIÓN

Este proyecto tiene como objetivo crear una tienda virtual donde diferentes usuarios puedan tener acceso a vender sus productos, así como también una administración que pueda monitorear los productos que se puedan vender en la página. Obteniendo la página un porcentaje de las ganancias funcionando como intermediario.

REQUISITOS DEL SISTEMA

1. HARDWARE

Procesador: Intel Core i3 o equivalente

Memoria RAM: 8GB

Almacenamiento: 50GB

Conexión a Internet estable

2. SOFTWARE

Sistema operativo: Windows 10 o Linux

Navegador web actualizado

Servidor Web con Express

Base de datos MongoDB

Sistema de control de versiones GIT

Estos son los requisitos mínimos y algunos de los programas utilizados para crear el programa.

ARQUITECTURA DEL SISTEMA

Para la realización de este proyecto se utilizaron las siguientes tecnologías:

BACKEND

1. NodeJS
2. Express como servidor
3. Mongoose como esquemas para BD
4. MongoDB para Base de datos
5. Babel como precompilador de NodeJS
6. Multer manejo de archivos

FRONTED

1. Angular
2. Bootstrap para Interfaz gráfica
3. Bootstrap Icons para iconos de GUI
4. SplideJS para carousel de imágenes y productos

Para un fácil manejo del CRUD en nuestra base de datos nos auxiliamos de mongoose para realizar esquemas que nos permitan establecer una estructura de las entidades.

```
1  import { Schema, model } from "mongoose";
2
3  const productSchema = new Schema(
4    {
5      name: String,
6      user_seller: {
7        type: Schema.Types.ObjectId,
8        ref: "User",
9      },
10     description: String,
11     image: String,
12     price: Number,
13     stock: Number,
14     allowed: {
15       type: Number,
16       default: 1,
17     },
18     category: String,
19   },
20   {
21     timestamps: true,
22     versionKey: false,
23   }
24 );
25
26 export default model("Product", productSchema);
```

Como este modelo de producto, que nos permite referenciar objetos como lo es "user_seller" que es el usuario que venderá dicho producto dentro de la plataforma.

Para las diferentes consultas como mencionamos anteriormente, hacemos uso de Express. Que permite realizar las típicas peticiones de un navegador, en los cuales tratamos de manejar los errores que puedan ocurrir en las distintas operaciones.

```

1  router.post("/login", async (req, res) => {
2    try {
3      const user = User(req.body);
4      const find = await User.findOne({ username: user.username });
5      if (find) {
6        if (find.password === user.password) {
7          res.status(200).send(find);
8        } else {
9          const alert = createMessage(
10             "Contraseña incorrecta",
11             "La contraseña es incorrecta"
12           );
13          res.status(200).send(alert);
14        }
15      } else {
16        const alert = createMessage(
17          "Usuario no encontrado",
18          "El usuario no se ha encontrado"
19        );
20        res.status(200).send(alert);
21      }
22    } catch (error) {
23      const al = createAlert(error);
24      res.status(400).send(al);
25    }
26  });

```

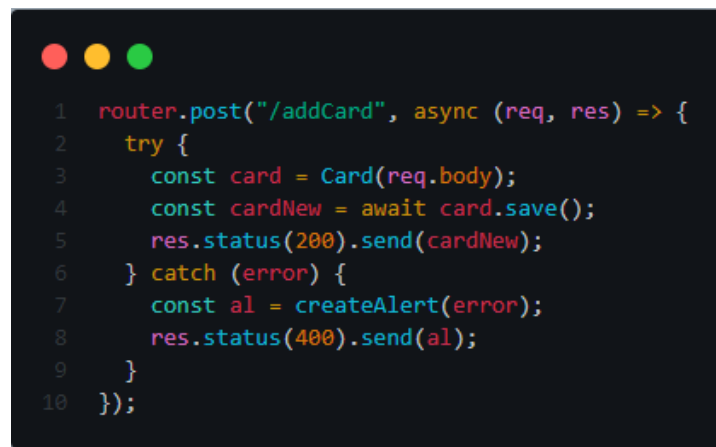
Este método nos sirve para validar las credenciales de un usuario cuando inicia sesión, por lo que se procede a buscar el usuario y que la contraseña coincida con la escrita en el formulario de inicio de sesión. De ser incorrecta la información se responde con un un objeto de tipo alerta.

```

1  class Alert {
2    constructor(title, message, type, show, code) {
3      this.title = title;
4      this.message = message;
5      this.type = type;
6      this.show = show;
7      this.code = code;
8    }
9  }
10
11  export default Alert;

```

Este objeto de tipo alerta como podemos observar, tiene un título, mensaje, tipo, la opción de si se debe mostrar y el código del error. Por lo que es información para que pueda ser visualizada por el usuario final y pueda entender lo que está ocurriendo dentro de la aplicación.

A screenshot of a code editor with a dark background and three colored window control buttons (red, yellow, green) in the top left corner. The code is written in a light-colored font and shows a function for adding a card. It includes a try-catch block to handle errors, creating an alert object and sending it back to the client on a 400 status code.

```
1 router.post("/addCard", async (req, res) => {  
2   try {  
3     const card = Card(req.body);  
4     const cardNew = await card.save();  
5     res.status(200).send(cardNew);  
6   } catch (error) {  
7     const al = createAlert(error);  
8     res.status(400).send(al);  
9   }  
10 });
```

Toda la funcionalidad de la aplicación está dada por estas funciones, que lo que hacen es crear, actualizar, eliminar y obtener la información de los diferentes esquemas de la base de datos.

Luego para el manejo de imágenes y archivos se hace uso de multer, que lo que hace es guardar la imagen en una carpeta específica para que luego pueda ser consultada y vista desde la interfaz gráfica por el usuario final.

De esta manera es como configuramos multer, indicando la carpeta en la que se almacenará la información, así como también se le asigna un límite al tamaño de las imágenes o archivos que es de 2MB. Si se cumple que el archivo es menor a 2MB se procede a asignar un nombre con un identificador único con UUID versión 4. Evitando la duplicidad de archivos y garantizando la consistencia de los datos. Así

mismo se limita a que el archivo recibido sea de tipo imagen como lo son jpeg, jpg, png y gif.

```
1  const storage = multer.diskStorage({
2    destination: path.join(__dirname, "public/img/products"),
3    filename: (req, file, cb) => {
4      cb(null, uuid() + path.extname(file.originalname).toLocaleLowerCase());
5    },
6  });
7
8  const upload = multer({
9    storage: storage,
10   dest: path.join(__dirname, "public/img/products"),
11   limits: { fileSize: 2000000 },
12   fileFilter: (req, file, cb) => {
13     const filetypes = /jpeg|jpg|png|gif/;
14     const mimetype = filetypes.test(file.mimetype);
15     const extname = filetypes.test(path.extname(file.originalname));
16     if (mimetype && extname) {
17       return cb(null, true);
18     }
19     cb("Error: El archivo debe ser una imagen valida");
20   },
21 }).single("image");
```

Para la configuración de la base de datos tenemos la configuración de conexión, en la cuál indicamos la dirección ip en la cuál está nuestro DBMS de MongoDB que es el 127.0.0.1 y el puerto 27017 en el que está funcionando, así como también especificamos el nombre de la base de datos que es "ecommerce". En caso de que no se pueda establecer la conexión se lanza un mensaje con el error.

```
1  import { connect } from "mongoose";
2
3  (async () => {
4    try {
5      const db = await connect("mongodb://127.0.0.1:27017/ecommerce", {
6        useNewUrlParser: true,
7        useUnifiedTopology: true,
8      });
9      console.log("Database is connected to:", db.connection.name);
10     } catch (error) {
11       console.log(error);
12     }
13   })();
```

EJECUCIÓN

Para poder iniciar nuestra aplicación "Backend" o nuestro servidor de respuesta para la interfaz gráfica debemos estar dentro de la carpeta "/Backend" y en una terminal ejecutar el comando "npm start". De esta manera se iniciará nuestra aplicación. En dado caso que se descargue del repositorio, deberá ejecutarse primero el comando "npm i" para instalar todas las dependencias.

```
marco@AGUARE-WIN MINGW64 /d/USAC/1er.2023/ARCHIVOS/LAB/E-commerce/Backend  
$ npm start
```

Para poder iniciar nuestra aplicación "Fronted" o nuestra interfaz gráfica debemos estar en nuestra carpeta "/Fronted" y en la terminal de comandos ejecutar "ng serve" ya que es una aplicación de angular. Si de igual manera se descargo el proyecto desde el repositorio se deberá ejecutar primero "npm i" para instalar las dependencias.

```
marco@AGUARE-WIN MINGW64 /d/USAC/1er.2023/ARCHIVOS/LAB/E-commerce/Fronted  
$ ng serve
```