MANUAL EXTENDIDO

Aplicación Virtual On

Por Adrián González Morales

ÍNDICE

INTRODUCCIÓN

El proyecto usará tanto programación de BackEnd como FrontEnd, en la cual crearemos tanto la vista html de la página, así como la programación de sus funcionalidades. A su vez, inicializar un servidor NodeJS que haga funcionar la página en local, para poder probarla y verla.

Necesitaremos instalar NodeJS para el servidor y MySQL (En mi caso usaré **XAMPP**) para la base de datos. Y MySQL Workbench si queremos trabajar de forma gráfica con la base de datos. En mi caso lo he usado para ver la información que se iba guardando y crear el diagrama de la base de datos desde ahí con sus herramientas.

Para instalar los repositorios necesarios iniciaremos el proyecto con el comando:

npm init --yes

Esto creará el archivo de configuración. En este podremos ver el nombre de la aplicación, la versión, su descripción, el archivo principal y los diferentes scripts. Lo modifico para que se adapte a mi aplicación.

Necesitaremos también estos módulos, que serán los que usaremos durante el proyecto:

- **Express**: es el framework que usaremos para este proyecto.
- **Express-handlebars**: extensión para usar el motor de plantillas de handlebars dentro de express.
- **Express-session**: extensión para poder crear sesiones para los distintos usuarios.
- **Mysql**: módulo para conectarnos a la base de datos.
- **Express-mysql-session**: para guardar las sesiones del usuario dentro de la base de datos
- **Morgan**: para poder mostrar por consola las peticiones http que van llegando al servidor.
- **Bcryptjs**: módulo para cifrar las contraseñas.
- **Passport**: módulo para autentificar a los usuarios.
- **Passport-local**: autentificar desde nuestra base de datos
- **Timeago.js:** Convertirá las fechas en un formato: hace 3 minutos, hace 4 horas, etc.
- **Connect-flash:** mostrar mensajes entre distintas vistas.
- **Express-validator:** validar datos que meta el usuario

BASE DE DATOS

Para poder guardar los datos de los usuarios necesitaremos una base de datos. Importo el archivo de contenido de la base de datos aquí. Nos quedaría algo así. Contaremos con dos usuarios, el usuario común con permisos para crear, borrar sus propias reservas y ver los eventos. Y un administrador que será capaz de crear, borrar y editar reservas y eventos.

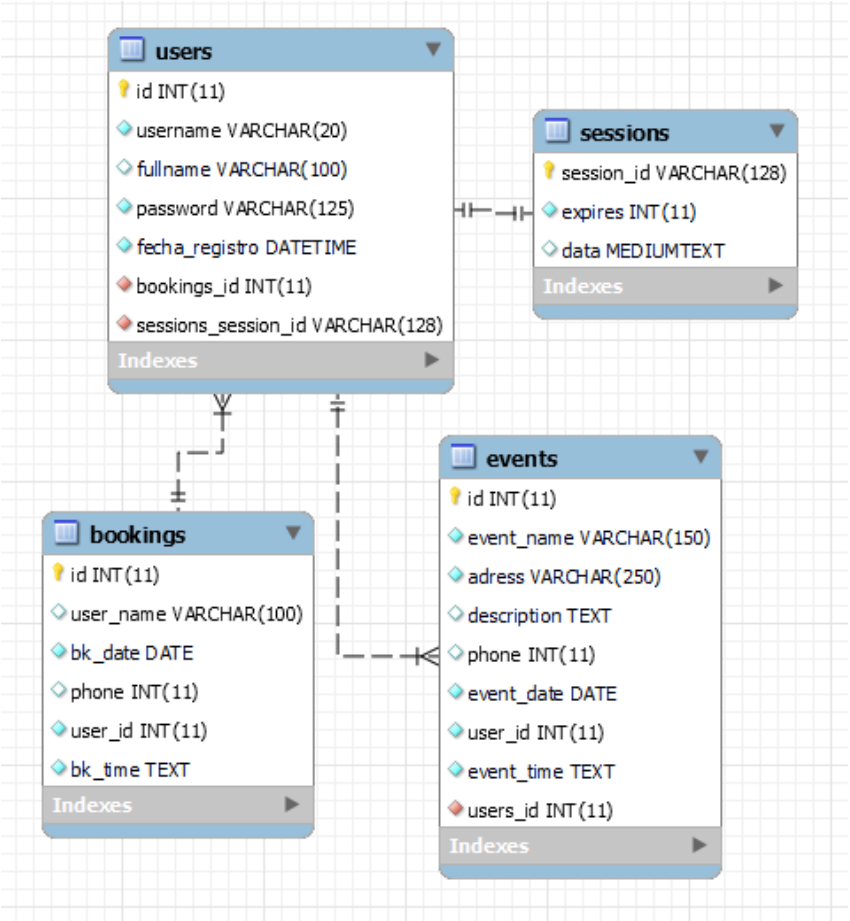
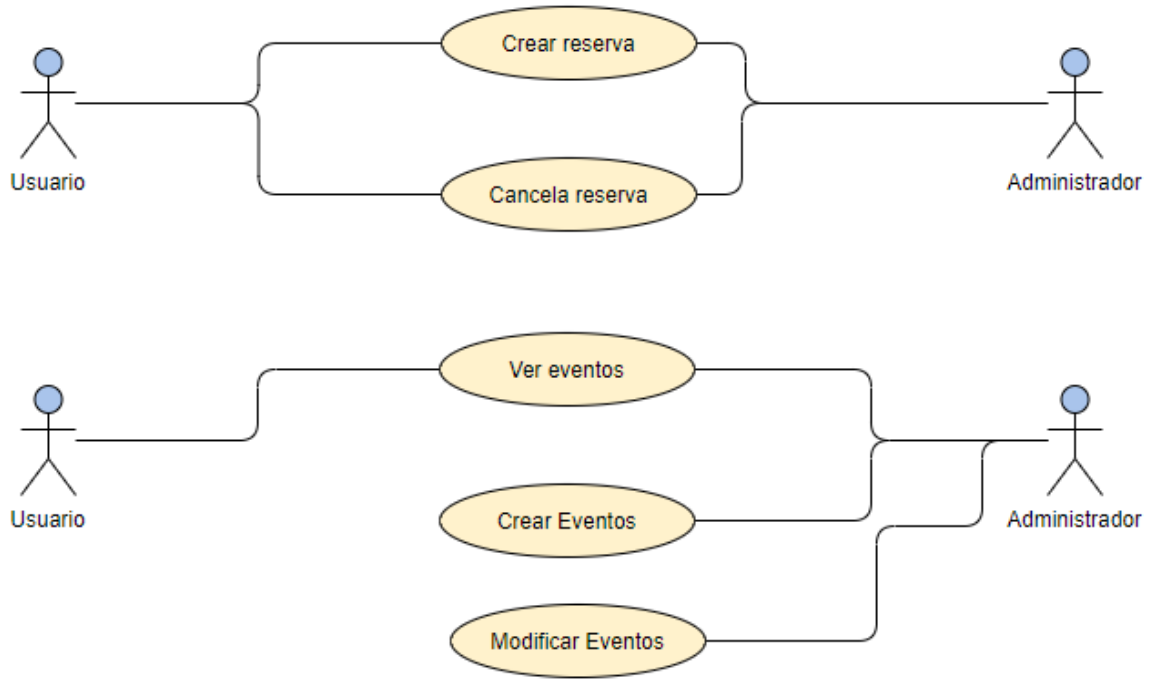


Diagrama de uso:



El archivo de la base de datos queda así:

```
-- Comandos en mysql para crear la Base de Datos
CREATE DATABASE web;
USE web;
-- Tabla USERS
CREATE TABLE users (
	id INT NOT NULL AUTO_INCREMENT,
	user VARCHAR(20) NOT NULL,
	fullname VARCHAR(100),
	password VARCHAR(30) NOT NULL,
	fecha_registro DATETIME NOT NULL DEFAULT current_timestamp,
	PRIMARY KEY(id)
);

-- Table EVENTOS
CREATE TABLE events (
	id INT NOT NULL AUTO_INCREMENT,
	event_name VARCHAR(150) NOT NULL,
	adress VARCHAR(250) NOT NULL,
	description TEXT,
	phone INT,
	event_date DATETIME NOT NULL,
	event_time TEXT NOT NULL,
	user_id INT NOT NULL,
	PRIMARY KEY(id)
);

-- Tabla RESERVAS
CREATE TABLE bookings (
	id INT NOT NULL AUTO_INCREMENT,
	user_name VARCHAR(100),
	bk_date DATETIME NOT NULL,
	bk_time TEXT NOT NULL,
	phone INT,
	user_id INT NOT NULL,
	PRIMARY KEY(id)
);
```

Una vez tenemos la base de datos,creo el servidor. Con los pasos anteriores de inicialización del proyecto, se nos crearon 2 archivos **“package-losck.json” y “package.json”** Aquí dentro se encontrará la configuración y las dependencias del proyecto:

```
{
    "name": "VirtualOn",
    "version": "1.0.0",
    "description": "",
    "main": "index.js",
    Depuración de ▷
    "scripts": {
        "start": "nodemon src/index.js"
    },
    "keywords": [],
    "author": "",
    "license": "ISC",
    "dependencies": {
        "bcryptjs": "^2.4.3",
        "connect-flash": "^0.1.1",
        "express": "^4.17.1",
        "express-handlebars": "^5.3.1",
        "express-mysql-session": "^2.1.5",
        "express-session": "^1.17.1",
        "express-validator": "^6.10.1",
        "morgan": "^1.10.0",
        "mysql": "^2.18.1",
        "passport": "^0.4.1",
        "passport-local": "^1.0.0",
        "timeago.js": "^4.0.2"
    },
    "devDependencies": {
        "nodemon": "^2.0.7"
    }
}
```

Podemos ver todas las dependencias antes nombradas, además de una extra que nos será de mucha ayuda en la creación de aplicaciones BackEnd **“nodemon dev”**. También podemos ver cómo selecciona nuestro script principal, en este caso **“index.js”**.

1. El archivo principal de todos los scripts 'index.js'.

Requerimos todos los módulos que tendremos que importamos al proyecto, incluyendo los datos para la conexión a la base de datos:

```
// Reequirimientos
const express = require('express');
const morgan = require('morgan');
const exphbs = require('express-handlebars');
const path = require('path');
const flash = require('connect-flash');
const session = require('express-session');
const MySQLStore = require('express-mysql-session');
const passport = require('passport');

const { database } = require('./keys');
```

Inicializamos express, el cual será la base para nuestro proyecto:

```
// Inicializaciones
const app = express();
require('./lib/passport');
```

Necesitaremos configurar el servidor. Lo iniciaremos en algún puerto que se encuentre libre con **"process.env.PORT"** o en caso contrario el **puerto 4000**. También estableceremos la carpeta donde estarán nuestras vistas html (en este caso las convertiremos en archivos handlebars) con el método **"path.join"**. Y también la configuración del motor de plantillas **"Handlebars"** y para que la extensión de los archivos no sea tan larga la configuraremos como **".hbs"**. Estableceremos también el Layout o vista principal, que será el archivo **"main.hbs"**, también el directorio de las demás vistas que se encontrarán en la carpeta **"layouts"** la cual usaremos para el login y el registro. Además, añadiremos también las vistas parciales en las que colocaremos la barra de navegación y los mensajes del servidor al usuario en la carpeta **"partials"**. En el caso de los helpers, contarán con las funciones necesarias para handlebars. Y por último lo enlazaremos con el archivo **"/lib/handlebars"** y lo estableceremos como el motor de vistas de nuestro proyecto:

```
// Configuracion
// Si hay un puerto libre, lo toma, sino tomará el puerto 4000
app.set('port', process.env.PORT || 4000);

// Establecer donde está la carpeta de las Vistas '/view/
app.set('views', path.join(__dirname, 'views'));

// Configuración de las plantillas de Handlebars
app.engine('.hbs', exphbs({ // Cambio de la extensión de .handlebars a .hbs
    defaultLayout: 'main',
    layoutsDir: path.join(app.get('views'), 'layouts'),
    partialsDir: path.join(app.get('views'), 'partials'),
    extname: '.hbs',
    helpers: require('./lib/handlebars')
}))
app.set('view engine', '.hbs');
```

- **HANDLEBARS.JS**

Lo usaremos para configurar la conversión de los timestamp con la biblioteca **timeago.js.** Con esto haremos que, si queremos, la fecha sea más legible para el usuario común. En vez de creado el día 5 de mayo a las 14:00, se verá como "Hace 15 días".

Creo el objeto helpers y lo exporto, porque será el que usemos en otras vistas de handlebars para colocar este formato de tiempo.

```
const { format } = require('timeago.js');

const helpers = {};

helpers.timeago = (timestamp) => {
    return format(timestamp);
}

module.exports = helpers;
```

- **MIDDLEWARES**:

Los middlewares son funciones que se ejecutan cada vez que un usuario ejecuta una petición o que una aplicación cliente manda una petición al servidor. El que nosotros hemos instalado es **"morgan"**, el parámetro que usaremos es dev, para que nos detalle por consola, los datos necesarios para los desarrolladores.

"express.urlencoded" aceptará las solicitudes entrantes desde los formularios de los usuarios, con la extensión que solo aceptará datos sencillos como strings, enteros, etc.

"express.json" para enviar y recibir archivos json.

```
// Middlewares
app.use(session({
    secret: 'nimnimsession',
    resave: false,
    saveUninitialized: false,
    store: new MySQLStore(database)
}));
app.use(flash());
app.use(morgan('dev'));
// Metodo para acptar desde los formularios los datos que nos pasan los usuarios en formatos sencillos (Strings, etc)
app.use(express.urlencoded({ extended: false }));
// Metodo para extender el tipo de variables que podemos recibir por el usuario
app.use(express.json());
app.use(passport.initialize());
app.use(passport.session());
```

- **VARIABLES GLOBALES**

Usaremos un middleware para declarar las variables y que puedan ser accedidas desde nuestra aplicación. Definiremos las variables de los dos mensajes, **"success"** para los mensajes de aceptación y de color verde, **"message"** para los mensajes de error y **"user"** para:

```
// Variables Globales
app.use((req, res, next) => {
    app.locals.success = req.flash('success'); // Variable del mensaje de Success
    app.locals.message = req.flash('message'); // Variable del mensaje de Error
    app.locals.user = req.user;
    next();
})
```

- **RUTAS**

La ruta principal en la carpeta **"/routes"** donde se ejecutará el **"index.js"**, pero no hará falta ponerlo ya que index.js es el valor predeterminado. Y otra ruta **"authentication"** donde irán el resto de funciones de nuestra aplicación.

```
// Rutas
app.use(require('./routes'));
app.use(require('./routes/authentication'));
```

- **PUBLIC**

Carpeta pública donde colocaremos **las imágenes, los vídeos, los archivos css y fuentes.** Establecemos la ruta de estos archivos en estático.

```
// Archivos Públicos
app.use(express.static(path.join(__dirname, 'public')));
```

- **SERVIDOR**

Por último, selecciona el puerto en el que se abrirá el servidor.

```
// Servidor
app.listen(app.get('port'), () => {
    console.log('PORT: ', app.get('port'));
});
```

2.ROUTES/INDEX

Aquí se mostrará la ruta principal, en la que se iniciará la página que abriremos por primera vez, y se mostrará la vista de bienvenida:

```
// Definición de la ruta principal al iniciar el servidor
const express = require('express');
const router = express.Router();

// Renderiza la vista index principal en la ruta '/'
router.get('/', (req, res) => {
    res.render('index');
});

// Lo exportamos
module.exports = router;
```

3.AUTHENTICATION.JS

Aquí están los métodos principales de nuestro programa, registro, login, etc. A su vez, también programaremos los cambios de pantalla que sucederán con cada acción además de los cambios entre las mismas.

```
// Script de la redireccion de las rutas
const express = require('express');
const router = express.Router();
const passport = require('passport');
const { isLoggedIn } = require('../lib/auth');
const { isNotLoggedIn } = require('../lib/auth');
const pool = require('../database');
```

Primero importamos los módulos necesarios que usaremos a continuación.
Empezamos creando las funciones que nos llevarán a cada ruta dentro de nuestra página, con la función **"router"** de Express:

```
router.get('/home', isLoggedIn, (req, res) => {
    res.render('home');
});
```

Esta función lo que hará será, obtener la ruta que hay en **"/home"** y en Home estará el renderizado de la vita **"home"**. Además de esto implementaremos la función **"isLoggedIn"** que explicaré en el apartado "isLoggedIn". Entonces, cuando se active la ruta **"localhost:4000/home"** por ejemplo, el resultado será el render de la vista Home:

Esta pantalla implementará la barra de tareas que habíamos implantado anteriormente con las plantillas de Handlebars y que saldrán en todas las pantallas. También, una vista de el twitter de la empresa, su red social principal y más importante implementada por medio de la API que nos proporciona twitter. Podemos conseguirla entrando en esta página y seleccionando simplemente el usuario de la cuenta que queremos mostrar: https://publish.twitter.com/#

Y por último añadí unos iconos de cada una de las redes sociales que tienen disponibles para acceder a ellas rápidamente, con un simple click.

Como he dicho que la página tenía un login, pero antes deberá tener un registro para que el usuario pueda loguearse. Lo estableceremos en la ruta **"/signup"**, con **"get"** mostraremos la vista y **"post"** lo usaremos para luego añadir los valores a la base de datos. Añadiremos la función **"isNotLoggedIn"** que hará lo contrario que la anterior, esta explicación la podemos encontrar en el apartado "is Not Logged In".

```
router.get('/signup', isNotLoggedIn, (req, res) => {
    res.render('auth/signup');
});

router.post('/signup', isNotLoggedIn, passport.authenticate('local.signup', {
    successRedirect: '/profile',
    failureRedirect: '/signup',
    failureFlash: true
}));
```

En la función **post** tendremos además la autenticación de passport en la que indicaremos que:

- con **successRedirect**: cuando la autentificación sea correcta nos redirigirá a nuestro perfil automáticamente.
- Con **failureRedirect**: nos quedaremos en la misma pestaña de signup.
- Y en caso de fallar se activará el **failureFlash**, que en nuestro caso lo pondremos como true y se activará la función de mensajes que declaramos en las Variables Globales.

Las función de signup está más explicada en detalle en el apartado "Signup".

Continuamos con la ruta **"/signin"** la cual tendrá la misma función que el signup pero con el logueo del usuario, sin registro.

Con get renderiza la vista **"auth/signin"** y cuando se ejecute la función de logueo dentro de la vista pasará a ejecutarse la función **post** que decidirá si las credenciales del usuario son correctas, en caso de serlo lo redireccionará a su perfil. Y de igual forma que la función Signup en la función **post** tendremos también la autenticación de passport en la que indicaremos que:

- con **successRedirect**: cuando la autentificación sea correcta nos redirigirá a nuestro perfil automáticamente.
- Con **failureRedirect**: nos quedaremos en la misma pestaña de signin.
- Y en caso de fallar se activará el **failureFlash**, que en nuestro caso lo pondremos como true y se activará la función de mensajes que declaramos en las Variables Globales. En el mensaje se indicará el error para que intentemos loguearnos de nuevo.

```
router.get('/signin', isNotLoggedIn, (req, res) => {
    res.render('auth/signin');

});

router.post('/signin', isNotLoggedIn, (req, res, next) => {

    passport.authenticate('local.signin', {
        successRedirect: '/profile',
        failureRedirect: '/signin',
        failureFlash: true
    })(req, res, next);
});
```

Para la ruta **"/profile"** ya que no usaremos ala autentificación se desarrollará toda la función en esta misma. con **get** en caso de que el id de usuario de inicio de sesión sea = 1, nos detectará como administradores y nos mandará a la vista **"/admin"**.

En caso contrario, renderiza la vista profile y en profile, hará una query asíncrona que obtendrá todos los datos de las reservas de ese usuario desde la base de datos, solo las que contengan su mismo id, ordenadas ascendentemente y nos mostrará en la consola los datos, para poder verificarlos. Con **post** no se ejecutará nada, porque no añadiremos nada a la base de datos, así que no será necesario.

```
router.get('/profile', isLoggedIn, async(req, res) => {
    if (req.user.id == 1) {
        res.redirect('admin')
    } else {
        //const bookings = await pool.query('SELECT * FROM bookings WHERE user_id = ?', [req.user.id]);
        const bookings = await pool.query("SELECT id, user_name, DATE_FORMAT(bk_date, '%d-%m-%Y') AS bk_date, phone, user_id, bk_time " +
            "FROM bookings  WHERE user_id = ? ORDER BY bk_date ASC", [req.user.id]);
        console.log(bookings);
        res.render('profile', { bookings });
    }
});
```

Para la la ruta **"/logout"** no tendremos vista ya que no será necesaria, porque nos deslogueará y nos mandará directamente a la vista de signin en caso de que queramos loguearnos de nuevo.

```
router.get('/logout', isLoggedIn, (req, res) => {
    req.logOut();
    res.redirect('/signin');
});
```

En la ruta **"/bookings"** será donde el usuario haga las reservas y se guarden directamente en la base de datos. Con **get** mostraremos la vista correspondiente.

Con **post** preparamos para enviar datos a la base de datos. creamos una variable que obtendrá los datos que recogeremos de la vista (con req.body), la fecha de la reserva (bk_date), el teléfono de contacto (phone) y la hora de reserva (bk_time). Una vez almacenado esto, crearemos otra variable que creará el objeto que insertará en la base de datos, el objeto newLink, le pasa por datos, los sacados del req.body además de el nombre de usuario (user_name) y el id de usuario (user_id) que lo recogerá de los datos que obtuvimos del usuario cuando se logueó.

A continuación ejecutaremos una query asíncrona para insertar el objeto newLink en la tabla de **bookings**, una vez completada la operación, nos redirigirá a la vista de profile, donde podremos ver el resto de reservas y el mensaje de que se ha completado con éxito, mensaje el cual hemos declarado en las variables locales.

```
router.get('/booking', isLoggedIn, (req, res) => {
    res.render('booking');
});

router.post('/booking', isLoggedIn, async(req, res) => {
    const { bk_date, phone, bk_time } = req.body;
    const newLink = {
        bk_date,
        user_name: req.user.username,
        phone,
        user_id: req.user.id,
        bk_time,
    };
    console.log(newLink);
    await pool.query('INSERT INTO bookings set ?', [newLink]);
    req.flash('success', 'You have made the reservation successfully');
    res.redirect('/profile');
});
```

En la ruta **"/contact"** solo mostraremos la vista de contact con la información sobre la empresa así que no necesitaremos nada más que verificar si el usuario está logueado con isLoggedIn.

```
router.get('/contact', isLoggedIn, (req, res) => {
    res.render('contact');
});
```

Para la ruta **"/events"** solo mostraremos los eventos que han sido registrados por el administrador, asi que solo usaremos **get** para un query asíncrona con la que obtendremos todos los datos de los eventos, para luego mostrarlos en la vista **events**.

```
router.get('/events', isLoggedIn, async(req, res) => {
    const events = await pool.query("SELECT id, event_name, adress, description, phone,  DATE_FORMAT(event_date, '%d-%m-%Y') AS event_date, user_id, event_time " +
        "FROM events ORDER BY event_date ASC");
    res.render('events', { events });

});
```

Con la ruta **"/events/:id"** será una extensión de la anterior /events, en la que se seleccionará el id del evento que haya sido clicado por el usuario y mostrará su información, pro eso, en esta query asíncrona solo mostraremos la información de 1 solo evento, el seleccionado en el id de la ruta. Renderizando así la vista **event_info** que mostrará los resultados de la query.

```
router.get('/events/:id', isLoggedIn, async(req, res) => {
    const { id } = req.params;
    const events = await pool.query("SELECT event_name, adress, DATE_FORMAT(event_date, '%d-%m-%Y') AS event_date, phone, description " +
        "FROM events WHERE id = ?", [id]);
    res.render('event_info', { events });
});
```

En la ruta **"/admin"** veremos lo redireccionado desde la vista profile si el id del usuario logueado es 1, por lo tanto nos identificará como Administrador y se mostrará todas las reservas de todos los usuarios con una query asíncrona, mostrando su información también y ordenados de forma ascendente. Además, se mostrarán también los eventos con otra query asíncrona que ha creado el administrador anteriormente y un botón para llevarlo a la vista para crear otro evento.

```
router.get('/admin', isLoggedIn, async(req, res) => {
    const bookings = await pool.query("SELECT id, user_name, DATE_FORMAT(bk_date, '%d-%m-%Y') AS bk_date," +
        " phone, user_id, bk_time FROM bookings ORDER BY bk_date ASC");
    console.log('BOOKINGS: ' + bookings);
    const events = await pool.query("SELECT id, event_name, adress, description, phone,  " +
        "DATE_FORMAT(event_date, '%d-%m-%Y') AS event_date, user_id FROM events ORDER BY event_date ASC");
    console.log('EVENTS: ' + events);
    res.render('admin', { bookings, events });

});
```

Desde admin, accedemos a la ruta **"/events_admin"** que será donde el administrador cree los nuevos eventos. Renderiza la vista de **events_admin**. Obtiene desde la vista los datos al req.body y los almacena en una constante y crearemos el objeto newLink de nuevo para insertarlo en la base de datos. Una vez realizado se mostrará un mensaje de que ha sido realizada la creación con éxito y nos redirige a la vista admin de nuevo.

```
router.get('/events_admin', isLoggedIn, (req, res) => {
    res.render('events_admin');
});

router.post('/events_admin', isLoggedIn, async(req, res) => {
    const { event_name, event_date, adress, description, phone, event_time } = req.body;
    const newLink = {
        event_name,
        event_date,
        adress,
        description,
        phone,
        user_id: req.user.id,
        event_time
    };
    await pool.query('INSERT INTO events set ?', [newLink]);
    req.flash('success', 'Event saved successfully');
    res.redirect('/admin');
});
```

Para los eventos también necesitaríamos borrarlos en algún momento, así que habilito una ruta para poder borrarlos, al igual que con el edit, lo seleccionaremos por id del evento. Coge el id del evento y con una query asíncrona borramos con la función de sql DELETE. Esto lanzará el mensaje de que se ha borrado correctamente y nos devolverá al **admin**.

```
router.get('/event_delete/:id', isLoggedIn, async(req, res) => {
    const { id } = req.params;
    await pool.query('DELETE FROM events WHERE ID = ?', [id]);
    req.flash('success', 'Event removed successfully');
    res.redirect('/admin');
});
```

Siguiendo con las rutas para los eventos, queda tener una forma de poder editarlos, en caso de haber algún cambio o algo así, entonces habilitamos la ruta **"/event_edit/:id"** que como la anterior, obtiene el id del evento que queremos editar desde la vista, y lo selecciona en la ruta para luego mostrarnoslo en la vista con **get**. Nos mostrará la información del evento y nos dará la opción de editarlo.

Para una vez terminado de editarlo, actualizarlo en la base de datos usaremos **post** y obtendremos el id y los demás datos del evento desde la vista, para meterlos en un objeto que luego serán los datos para la query asíncrona y hacer UPDATE. Gracias a la función asíncrona no tendremos que esperar y nos llevará a la pantalla de admin de nuevo, donde se mostrarán los datos de nuevo, actualizados y como hemos vuelto a invocar el mensaje de "success", nos volverá a salir otro mensaje que se ha realizado la edición correctamente.

```
router.get('/event_edit/:id', isLoggedIn, async(req, res) => {
    const { id } = req.params;
    const events = await pool.query('SELECT * FROM events WHERE id = ?', [id]);
    res.render('edit', { events: events[0] });
});

router.post('/event_edit/:id', isLoggedIn, async(req, res) => {
    const { id } = req.params;
    const { event_name, adress, event_date, description, phone, event_time } = req.body;
    const newLink = {
        event_name,
        event_date,
        adress,
        description,
        phone,
        event_time
    };
    await pool.query('UPDATE events set ? WHERE id = ?', [newLink, id]);
    req.flash('success', 'Event edited successfully');
    res.redirect('/admin');
});
```

Y por último en este apartado tenemos la ruta **"/booking_delete"** para borrar las reservas hechas, esta estará disponible tanto para los usuarios como para el administrador. Seleccionará por el id se la reserva seleccionada desde la vista y con una query asíncrona se iniciará el comando DELETE para ese id. Como en los anteriores, se mostrará un mensaje que ha sido realizado con éxito.

```
router.get('/booking_delete/:id', isLoggedIn, async(req, res) => {
    const { id } = req.params;
    await pool.query('DELETE FROM bookings WHERE ID = ?', [id]);
    req.flash('success', 'Bookinng removed successfully');
    res.redirect('/profile');
});
```

Y exporta el módulo para que lo reconozca el archivo principal:

```
module.exports = router;
```

4.AUTH

Como comentamos anteriormente, para todos los métodos que nos hacen movernos por las diferentes vistas, hay dos funciones que se le aplicaban a cada uno. Estas nos servirán para que cuando el usuario no esté logueado, no pueda acceder a las distintas funciones de la página, como las reservas, etc. y a su vez, si está logueado, que no pueda acceder al registro o al logueo:

- **isLoggedIn:** es la función que permitirá ver la vista solo si el usuario está registrado. Usando el método isAuthenticated() si la respuesta es true, se ejecutará la función next y continuará el código con normalidad. En caso contrario, nos mandará de nuevo a la pestaña signin.

```
// Módulo que detecta si el usuario está logueado
isLoggedIn(req, res, next) {
    if (req.isAuthenticated()) {
        return next();
    }
    // Te devuleve al login si intentas acceder a alguna
    // vista que el usuario necesite estar logueado para verla
    return res.redirect('/signin');
},
```

- **isNotLoggedIn:** al contrario, esta función si no estás logueado, te dejará seguir tu camino, pero si estás logueado, te mandará de nuevo a tu perfil.

```
// Módulo que detecta si el usuario no está logueado
isNotLoggedIn(req, res, next) {
    if (!req.isAuthenticated()) {
        return next();
    }
    // te devuelve al perfil si el usuario está logeado
    return res.redirect('/profile');

}
```

5.HANDLEBARS

Este es un archivo de configuración de handlebars, que usaremos para implementar la librería **"timeago.js"** que hará que el tiempo si lo requerimos, en vez de mostrar la fecha exacta, mostraría el tiempo que lleva desde que se ha creado: "Hace 5 horas." "Hace 3 días":

```
// script para la configuración de la biblioteca timeago.js
const { format } = require('timeago.js');

const helpers = {};

// Timeago convierte la fecha de timestamp en una fech amás legible para el usuario
helpers.timeago = (timestamp) => {
    return format(timestamp);
}

module.exports = helpers;
```

6.HELPERS

En este archivo, se inicializa todo lo que tiene que ver con la contraseña del usuario. No es muy recomendable que guardes las contraseñas en la base de datos sin encriptar, entonces nos valdremos de estos métodos para encriptar y para que cuando el programa verifique en el login, también coincida con la contraseña guardada encriptada. La biblioteca **bcryptjs** se encargará de la mayoría del trabajo.

```
const bcrypt = require('bcryptjs');

const helpers = {};

helpers.encryptPassword = async(password) => {
    const salt = await bcrypt.genSalt(10);
    const hash = await bcrypt.hash(password, salt);
    return hash;
};

helpers.matchPassword = async(password, savedPassword) => {
    try {
        return await bcrypt.compare(password, savedPassword);
    } catch (e) {
        console.log(e);
    }
};

module.exports = helpers;
```

7.PASSPORT

En este script definiremos la forma en la que se autentican a los usuarios y a mostrarles solo, los datos de su propia sesión. Cada usuario tendrá una sesión propia creada en la base de datos, y con estas funciones haremos que solo cargue para ellos los datos de su propia sesión o crearla en caso de que sea la primera vez que inicien sesión.

```
passport.use('local.signin', new LocalStrategy({
    usernameField: 'username',
    passwordField: 'password',
    passReqToCallback: true
}, async(req, username, password, done) => {
    console.log(req.body);
    const rows = await pool.query('SELECT * FROM users WHERE username = ?', [username]);
    if (rows.length > 0) {
        const user = rows[0];
        // Validar que la contraseña sea igual
        const validPassword = await helpers.matchPassword(password, user.password)
            // Mensajes para el usuario
        if (validPassword) {
            done(null, user, req.flash('success', 'Welcome ' + user.username));
        } else {
            done(null, false, req.flash('message', 'Incorrect Password'));
        }
    } else {
        return done(null, false, req.flash('message', 'Username not found'));
    }
}));
```

Con local.signin verifica que cuando el usuario quiera entrar con su cuenta, busque en la base de datos el username para verificar que los datos sean correctos, que las contraseñas coinciden.

Con local.signup se guardarán los datos del usuario en la base de datos, insertándose después de encriptarlos con la función de encriptado que habías creado antes.

```
// Recoge los datos del usuario, encripta la contraseña
// y los inserta en la base de datos
passport.use('local.signup', new LocalStrategy({
    usernameField: 'username',
    passwordField: 'password',
    passReqToCallback: true
}, async(req, username, password, done) => {
    // Datos que almacenará
    const { fullname } = req.body;
    const newUser = {
        username,
        password,
        fullname
    };
    newUser.password = await helpers.encryptPassword(password); // Encriptado
    const result = await pool.query('INSERT INTO users SET ?', [newUser]);
    newUser.id = result.insertId; // Añadimos un id al usuario
    return done(null, newUser);


}));
```

Con serializer lo usaremos para el guardado de sesiones del servidor con respecto a los usuarios. Creará una sesión en la tabla "sessions" en la base de datos para el servidor. Esta tabla será creada gracias al sistema de sesiones de express, por lo tanto, nosotros no tendremos que crear una tabla extra. Se almacenan los datos de sesión en el servidor, sólo guarda el ID de sesión en la propia cookie, no los datos de sesión. De forma predeterminada, utiliza el almacenamiento en memoria.

```
// Para el guardado en las sesiones de express
passport.serializeUser((user, done) => {
    done(null, user.id);

});

// Consulta de los datos del usuario desde su propia sesión
passport.deserializeUser(async(id, done) => {
    const rows = await pool.query('SELECT * FROM users Where id = ?', [id]);
    done(null, rows[0]);
});
```

Y con esto acaba el manual extendido de las funciones del proyecto, con todo esto se podrá iniciar el servidor simplemente ejecutando el comando por consola de:

npm start

Y el mismo servidor nos irá dando información de los movimientos que está haciendo el usuario. El servidor está preparado para ejecutarse en local, con una conexión a un host, se puede desplegar en un dominio personalizado, pero este host deberá ser especial para ser compatible con NodeJS y con Handlebars.

8.PROGRESSIVE WEB APPLICATION

Por último, uno de los objetivos de este proyecto era que la página web, pudiera funcionar como una aplicación móvil nativa. Para ellos tendremos que convertirla en una progressive web application. El problema de estas aplicaciones es que solo pueden ser desplegadas en un servicio https sino no funcionará. Por lo tanto, no podré desplegar la aplicación móvil en nuestro servidor local. Pero aun así la dejaremos preparada para que si algún día quisiéramos desplegarla en un servidor https, funcionara sin ningún problema.

Para ello tendremos que crear un script de Service Worker que inicie este proceso ("sw.js") y un manifest.json para configurarlo. Nuestro manifest será así:

```
{
    "name": "VirtualON",
    "short_name": "VirtualON",
    "description": "Web Site for bookings and events in Virtual Reality",
    "background_color": "#000000",
    "theme_color": "#000000",
    "orientation": "portrait",
    "display": "standalone",
    "start_url": "./?utm_source=web_app_manifest",
    "scope": "./",
    "lang": "en-US",
    "icons": [{
        "src": "./src/public/img/virtual-1.png",
        "sizes": "240x240",
        "type": "image/png"
    }]
}
```

Donde saldrá el nombre, el nombre corto, la orientación, etc. Cabe destacar que el modo de display lo estableceremos como "standalone" para que la barra de navegación sea invisible y funcione como una app nativa.

Una vez tengamos el manifest, tocará la configuración del Service Worker. Asignará un nombre y la versión para el caché de la memoria interna:

```
//asignar un nombre y version al cache
const CACHE_NAME = 'v1_cache_virtual_on',
    urlsToCache = [
        './',
        './src/public/css/style.css'
    ]
```

Para guardar estos datos en el caché, necesitaremos configurar los archivos que se guardarán en este, los archivos estáticos:

```
// Durante la fase de instalación, se almacenará en caché los activos estáticos
self.addEventListener('install', e => {
    e.waitUntil(
        caches.open(CACHE_NAME)
        .then(cache => {
            return cache.addAll(urlsToCache)
                .then(() => self.skipWaiting())
        })
        .catch(err => console.log('Falló registro de cache', err))
    )
})
```

También, al ser una aplicación instalable, necesitaremos que se pueda iniciar sin conexión y que se carguen los datos del caché, por lo tanto, tendremos esta función que buscará los recursos que hará que funcione sin conexión.

```
//Una vez que se instala el SW, se activa y busca los recursos para hacer que funcione sin conexión
self.addEventListener('activate', e => {
    const cacheWhitelist = [CACHE_NAME]
    e.waitUntil(
        caches.keys()
        .then(cachesNames => {
            cachesNames.map(cacheName => {
                //Elimina lo que ya no se necesita en cache
                if (cacheWhitelist.indexOf(cacheName) === -1) {
                    return caches.delete(cacheName)
                }
            })
        })
        //Le indica al ServiceWorker activar el cache actual
        .then(() => self.clients.claim())
    )
})
```

En caso de que se volviera a recuperar la conexión, se tendría que actualizar y volver a cargar los datos sin ser del caché. Por lo tanto la función quedaría algo así:

```
//Cuando el navegador recupera una url
self.addEventListener('fetch', e => {
    //Responder con el objeto en cache o continuar buscando la url real actualizada
    e.respondWith(
        caches.match(e.request)
        .then(res => {
            if (res) {
                //recuperando del cache
                return res
            }
            //recuperar la peticion a la url
            return fetch(e.request)
        })
    )
})
```