
Universidad Autónoma de San Luis Potosí
Facultad de Ingeniería


# Manual del programador de la aplicación web *Centro Huellitas*

Hecho por:
Cervantes Díaz Ximena - 276978
Martínez García Alberto Enrique - 278603

Materia: Aplicaciones Web Escalables

Semestre: 2021/2022 I

Profesor: Francisco Everardo Estrada Velázquez

# Contenido

# 1. Introducción

En este documento se describe el proceso para la instalación del proyecto, los requerimientos para la misma, las configuraciones necesarias, una breve explicación de los servicios que se utilizaron, la estructura del proyecto, así como una explicación detallada de la parte de la programación.

# 2. Inicialización

## 2.1. Requerimientos

Lo necesario para poder correr el proyecto son los siguientes:

- Node versión 12 o superior.
- MongoDBCompass.
- npm.
- Cualquier terminal (cmd, powershell, etc.).
- Se utiliza git como controlador de versiones.

## 2.2. Descarga

Para facilitar la descarga del proyecto se recomienda hacer uso de una terminal en la cual se puedan ejecutar comandos de Git. Los pasos necesarios para la descarga son los siguientes:

- Dirigirse a la carpeta donde desea guardar el proyecto.
- Utilizar el comando *git clone* para poder clonar el repositorio de GitHub desde la siguiente liga: *https://github.com/Aplicaciones-Escalables/proyecto-los-punkitos.git*

## 2.3. Instalación

Para poder hacer un uso correcto del proyecto y que este no presente errores al momento de estar usándolo es necesario ejecutar algunos comandos para incorporar todos los servicios utilizados. Para esto lo primero que se debe de hacer es ir a la carpeta raíz del proyecto, una

vez estando allí se debe de abrir una terminal y ejecutar los siguientes comandos:

- npm install react-router-dom
- npm install express
- npm install mongoose
- npm install cors
- npm install body-parser –save

Una vez estos comandos hayan sido ejecutados el proyecto estará listo para ser utilizado.

### 2.4. Visualización

Para lograr visualizar el proyecto es necesario abrir una terminal, dirigirse a la carpeta raíz del proyecto y ejecutar el comando *npm start* esto hará que arranquen los scripts de React, de igual manera, para correr la parte del backend es necesario moverse a la carpeta *backend* y ejecutar el comando *node app.js*. Una vez hecho esto se abrirá una pestaña nueva del navegador en donde se mostrará el proyecto.

## 3. Servicios

Para la realización del proyecto fueron utilizados distintos servicios, a continuación, se presenta una descripción breve de cada uno de ellos.

### 3.1. ReactJS

React es una biblioteca JavaScript de código abierto diseñada para crear interfaces de usuario con el objetivo de facilitar el desarrollo de aplicaciones en una sola página. Es mantenido por Facebook y la comunidad de software libre. En el proyecto hay más de mil desarrolladores libres.

### 3.2. React Router DOM

React Router DOM permite implementar enrutamiento dinámico en una aplicación web. A diferencia de la arquitectura de enrutamiento tradicional en la que el enrutamiento se maneja en una configuración fuera de una aplicación en ejecución, React Router DOM facilita el enrutamiento basado en componentes de acuerdo con las necesidades de la aplicación y la plataforma.

## 3.3. Bootstrap

Bootstrap es un framework front-end utilizado para desarrollar aplicaciones web y sitios mobile first, o sea, con un layout que se adapta a la pantalla del dispositivo utilizado por el usuario.

## 3.4. NodeJS

Node.js es un entorno en tiempo de ejecución multiplataforma, de código abierto, para la capa del servidor basado en el lenguaje de programación JavaScript, asíncrono, con E/S de datos en una arquitectura orientada a eventos y basado en el motor V8 de Google.

## 3.5. MongoDB

MongoDB es un sistema de base de datos NoSQL, orientado a documentos y de código abierto.
En lugar de guardar los datos en tablas, tal y como se hace en las bases de datos relacionales, MongoDB guarda estructuras de datos BSON (una especificación similar a JSON) con un esquema dinámico, haciendo que la integración de los datos en ciertas aplicaciones sea más fácil y rápida.

## 3.6. Mongoose

Mongoose es una librería para Node.js que nos permite escribir consultas para una base de datos de MongooDB, con características como validaciones, construcción de queries, middlewares, conversión

de tipos y algunas otras, que enriquecen la funcionalidad de la base de datos.

### 3.7. ExpressJS

Express.js, o simplemente Express, es un marco de aplicación web de backend para Node.js, lanzado como software gratuito y de código abierto bajo la licencia MIT. Está diseñado para crear aplicaciones web y API. Se le ha llamado el marco de servidor estándar de facto para Node.js.

## 4. Estructura del proyecto

El proyecto, en su totalidad, fue creado bajo la estructura MERN, que es un conjunto de tecnologías utilizadas para el desarrollo de aplicaciones web que consta de MongoDB, ExpressJS, ReactJS y NodeJS. El utilizar MERN hace que nuestra aplicación sea escalable, es decir, en algún futuro no va a ser complicado modificar los módulos ya existentes o añadir nuevos módulos. Algunos otros beneficios de utilizar MERN son los siguientes:

- Abarca todo el ciclo de desarrollo, desde el frontend hasta el backend.
- Facilita el proceso de trabajar con una arquitectura modelo vista controlador (MVC) haciendo que el desarrollo fluya sin problemas.
- Se basa en 4 tecnologías probadas y ampliamente respaldadas: Mongo DB, Express, Angular/React, NodeJS.
- Frameworks basados en código abierto y con el respaldado por los apoyos de su comunidad.

## 5. Backend

Como se mencionó anteriormente, la parte del backend está hecha principalmente sobre NodeJS, pero, además, se utilizaron distintas librerías para facilitar el trabajo. De igual manera, para mantener un mejor orden y estructura en el código se separo en distintas carpetas con distintas funciones, pero todas

trabajan en conjunto.

## 5.1. Routes

Las rutas con cómo, desde el frontend, se va a llamar a cada función de la API. En el proyecto se encuentran dos grupos de rutas.

### 5.1.1. event.route.js

| Ruta | Descripción |
|---|---|
| /api/events | Obtiene todos los eventos registrados |
| /api/event/:id | Obtiene un solo evento diferenciándolo por su id |
| /api/editevent | Edita un evento |
| /api/addevent | Añade un nuevo evento |
| /api/deleteevent/:id | Elimina un solo evento por medio de su id |

### 5.1.2. pet.route.js

| Ruta | Descripción |
|---|---|
| /api/pets | Obtiene todas las mascotas registradas |
| /api/pet/:id | Obtiene una sola mascota diferenciándola por su id |
| /api/editpet | Edita una mascota |
| /api/addpet | Añade una nueva mascota |
| /api/deletepet/:id | Elimina una sola mascota por medio de su id |

## 5.2. Controllers

Los controladores son un puente entre las rutas y los servicios,

dependiendo de la ruta que se escoja es la función que se va a ejecutar y por lo tanto se indica que servicio va a ser llamado.

Debido a que los controladores son únicamente un puente, no se explicará a detalle su funcionamiento, pero se puede observar que en cada función de los dos distintos controladores se utiliza *EventSerice* o *PetService* según sea el caso.

A continuación, se describirán los controladores utilizados en el proyecto.

### 5.2.1. Event.controller.js

Este es el controlador encargado de los eventos. Las funciones que realiza son las siguientes.

```
listEvents: async(req, res) => {
     const events = await
     EventService.listEvents(res);
     res.json(events)
}

editEvent: async(req, res) => {
     res.json({
          events: await
          EventService.editEvent(req.body)
    })
}

deleteEvent: async(req, res) => {
     res.json({
          events: await
          EventService.deleteEvent(req.params.
          id)
    })
```

```
}

getOneEvent: async(req, res) => {
     const oneEvent = await
     EventService.getOneEvent(req.params.id)
     res.json(oneEvent)
}

createEvent: async(req, res) => {
     res.json({
          events: await
          EventService.createEvent(req.body)
    })
}
```

### 5.2.2. Pet.controller.js

Este es el controlador encargado de los eventos. Las funciones que realiza son las siguientes.

```
listPets: async(req, res) => {
     const pets = await
     PetService.listPets(res);
     res.json(pets)
}

 editPet: async(req, res) => {
    res.json({
        pets: await
PetService.editPet(req.body)
    })
},
```

```
 deletePet: async(req, res) => {
    res.json({
        pets: await
PetService.deletePet(req.params.id)
    })
},

 getOnePet: async(req, res) => {
     console.log(req.params.id);
     const onePet = await
PetService.getOnePet(req.params.id)
    res.json(onePet)
},

 createPet: async(req, res) => {
     console.log(req.body);
    res.json({
        pets: await
PetService.createPet(req.body)
    })
}
```

## 5.3. Services

Los servicios son los encargados de “manipular” la base de datos ya que estos se relacionan directamente con ella. Permiten la obtención, la modificación, la eliminación y la creación de los datos dentro de MongoDB. Los servicios utilizados en el proyecto son los siguientes.

### 5.3.1. event.service.js

```
async listEvents(){
     try {
```

```
            const events = await Event.find({});
            return events;
        } catch (error) {
            return error;
        }
}
```

Esta función hace uso de la función *find* de Mongoose la cual, al no pasarle ningún parámetro para la búsqueda, nos retorna todos los objetos de la colección de los eventos.

```
async editEvent(editedEvent){
    try {
        await Event.findOneAndUpdate({_id:
        editedEvent._id},
        editedEvent).then(() => {return
        editedEvent});
    } catch (error) {
        return error;
    }
}
```

Esta función es utilizada para modificar los eventos recibe como parámetro un evento, el cual contendrá la información que será la que reemplazará al objeto en la base de datos. Hace uso de la función de Mongoose *findOneAndUpdate* que recibe como primer parámetro el parámetro con el que realizaremos la búsqueda, es este caso el id del evento, y como segundo parámetro el objeto con la información del evento que reemplazará el evento actual.

```
async deleteEvent(id){
    try {
        await Event.findOneAndDelete({_id:
```

```
            id}).then((value) => {return
            value});
        } catch (error) {
            return error;
        }
}
```

Esta función funciona para borrar un único evento, recibe como parámetro el id del objeto a eliminar. Hace uso de la función de Mongoose *findOneAndDelete* que recibe como parámetro el id del evento a buscar y una vez que lo encuentra lo elimina de la base de datos.

```
async getOneEvent(id){
    try {
        return await
        Event.findById({_id:id});
    } catch (error) {
        return error;
    }
}
```

Esta función retorna un único evento de la colección, recibe como parámetro el id a buscar. Hace uso de la función de Mongoose *findById,* la cual usa el índice recibido para buscar dentro de la colección de la base de datos una coincidencia, una vez encontrado, nos retorna dicho evento.

```
async createEvent(newEvent = new Event()){
    try {
        await
        Event.create(newEvent).then((value)
        => {return value;})
    } catch (error) {
```

```
            return error;
        }
}
```

Esta función es la encargada de hacer un nuevo registro de un evento dentro de la base de datos, recibe como parámetro el nuevo evento que será el que introducirá a la base de datos. Hace uso de la función de Mongoose *create* que, como su nombre lo indica, crea un nuevo objeto en la colección correspondiente.

### 5.3.2. pet.service.js

```
async listPets(res){
    try {
        const pets = await Pet.find({});
        return pets;
    } catch (error) {
        return error;
    }
}
```

Esta función hace uso de la función *find* de Mongoose la cual, al no pasarle ningún parámetro para la búsqueda, nos retorna todos los objetos de la colección de las mascotas.

```
async editPet(editedPet){
    try {
        await Pet.findOneAndUpdate({_id:
        editedPet._id}, editedPet).then(()
        => {return editedPet});
    } catch (error) {
        return error;
    }
```

```
}
```

Esta función es utilizada para modificar las mascotas recibe como parámetro una mascota, la cual contendrá la información que será la que reemplazará al objeto en la base de datos. Hace uso de la función de Mongoose *findOneAndUpdate* que recibe como primer parámetro el parámetro, en este caso el id, con el que realizaremos la búsqueda y como segundo parámetro el objeto con la información del evento que reemplazará los datos de la mascota actual.

```
async deletePet(id){
     try {
          await Pet.findOneAndDelete({_id:
          id}).then((value) => {return
          value});
     } catch (error) {
          return error;
     }
}
```

Esta función es utilizada para borrar una única mascota, recibe como parámetro el id del objeto a eliminar. Hace uso de la función de Mongoose *findOneAndDelete* que recibe como parámetro el id de la mascota a buscar y una vez que la encuentra la elimina de la base de datos.

```
async getOnePet(id){
     try {
          return await Pet.findById({_id:
          id});
     } catch (error) {
          return error;
     }
```

```
}
```

Esta función retorna una única mascota de la colección, recibe como parámetro el id a buscar. Hace uso de la función de Mongoose *findById,* la cual usa el índice recibido para buscar dentro de la colección de la base de datos una coincidencia, una vez encontrado, retorna la mascota encontrada.

```
async createPet(newPet = new Pet()){
     try {
          await
          Pet.create(newPet).then((value) => {
               return value;
          })
     } catch (error) {
          return error;
     }
}
```

Esta función es la encargada de hacer un nuevo registro de una mascota dentro de la base de datos, recibe como parámetro la nueva mascota que será la que se introducirá a la base de datos. Hace uso de la función de Mongoose *create* que, como su nombre lo indica, crea un nuevo objeto en la colección correspondiente.

## 5.4. Models

Los modelos son una representación de las colecciones que tenemos en MongoDB en el entorno de NodeJS. Son usados para manipular los datos con mayor facilidad. En el proyecto hay dos modelos, los cuales serán descritos a continuación.

### 5.4.1. event

```
const eventSchema = mongoose.Schema({
    nombre: { type: String, required: true },
    fecha: { type: String, required: true },
    descripcion: { type: String, required:
true },
    image: {type: String}
}, {collection: 'eventos'});
```

### 5.4.2. pet

```
const eventSchema = mongoose.Schema({
    nombre: { type: String, required: true },
    descripcion: { type: String, required:
true },
    edad: { type: String, required: true },
    sexo: { type: String, required: true },
    image: {type: String}
}, {collection: 'mascotas'});
```

## 5.5. App.js

Este es el archivo principal de nuestro backend, es el que se ejecutará con node. Este archivo contiene la especificación del puerto en el cual el backend estará corriendo, así como la conexión con la base de datos de mongoose, de igual manera contiene la importación de todas las rutas que usará el frontend. Su contenido es el siguiente:

```
const express = require("express");
const bodyParser = require("body-parser");
const mongoose = require("mongoose");
const cors = require("cors");
const app = express();
app.use(bodyParser.json());
app.use(cors());
```

```
app.use(require("./routes/pet.route"));
app.use(require("./routes/event.route"));
const port = 3030;
app.listen(port, () => {
  console.log(`Example app listening at
  http://localhost:${port}`)
});

mongoose
  .connect(
    "mongodb://127.0.0.1:27017/test"
  )
  .then(() => {
    console.log("Connected to database!");
  })
  .catch(() => {
    console.log("Connection failed!");
});
module.exports = app;
```

# 6. Frontend

## 6.1. App.js

Es el archivo principal el cual es generado de manera automática cuando inicializamos una aplicación de React. Dentro de este archivo está contenida la totalidad de la aplicación. De igual manera, este archivo contiene la definición de las rutas por las cuales podemos navegar dentro del sistema. Su contenido es el siguiente.

```
function App() {
  return (
      <Routes>
        <Route path="/" element={<Homepage />} />
```

```
        <Route path="/mascotas" element={<Pets />}
/>
        <Route path="/eventos" element={<EventsPage
/>} />
        <Route                path="/agregarevento"
element={<CreateEvent />} />
        <Route             path="/editarevento/:id"
element={<EditEvent />} />
        <Route               path="/agregarmascota"
element={<CreatePet />} />
        <Route            path="/editarmascota/:id"
element={<EditPet />} />
      </Routes>

  );
}
```

Como se puede observar cada elemento es una ruta, dentro del parámetro *path* se define el nombre con el que se reconocerá a dicha ruta y dentro del parámetro *element* se define el componente que será mostrado cuando se acceda a esa ruta.

## 6.2. Componentes

### 6.2.1. HomePage

Es la pagina principal de la aplicación, nos muestra los últimos eventos y mascotas que han sido añadidas en el sistema, de igual manera nos muestra todas las secciones a las que se tiene acceso como administrador. Su definición es la siguiente

```
const Homepage = () => {
    return(
        <>
            <Header />
```

```
            <div className="container-fluid w-
75 mb-5">
                <InfoContainer
category="eventos" />
                <InfoContainer
category="mascotas" />
                <h1>Todas las secciones</h1>
                <div className="row mt-5 d-
flex justify-content-between">
                    <HomeSection name="Todas
las mascotas" route="/mascotas"/>
                    <HomeSection
name="Registrar nueva mascota"
route="/agregarmascota"/>
                </div>
                <div className="row mt-3 d-
flex justify-content-between">
                    <HomeSection name="Todos
los eventos" route="/eventos"/>
                    <HomeSection
name="Registrar nuevo evento"
route="/agregarevento"/>
                </div>
            </div>

        </>
    )
}
```

### 6.2.2. Pets y EventsPage

Son las páginas encargadas de mostrar toda la lista de

mascotas y de eventos, respectivamente. Su funcionamiento es bastante similar, lo único que cambia entre ellas es la ruta a donde se realizan las peticiones y los datos que son utilizadas, por lo que se explicará solamente un componente en vez de los dos.

Su definición es la siguiente.

```
const Pets = () => {
    const [pets, setPets] = useState([]);

    const url =
"http://localhost:3030/api/pets"

    const listPets = async () => {
        const respuesta = await fetch(url);
        const datos = await respuesta.json();
        setPets(datos);
    }

    const deletePet = async (_evento, id) => {
        console.log("Borrando al id - " + id);
        const respuesta = await
fetch("http://localhost:3030/api/deletepet/" +
id, {method: "DELETE"});
        if(respuesta){
            listPets();
        }
    }

    useEffect(() => {
        listPets();
    }, [])
```

```
    return(
        <>
            <Header/>
            <Link to="/agregarmascota"
classNamе="btn btn-pimary sticky-
button">Agregar una nueva mascota</Link>
            <div classNamе="container">
                <div classNamе="row mt-5
justify-content-center">
                {
                    pets && pets.map((pet,
_index) => (
     <PetCard
          key={`pet-${_index}`}
          id={pet._id}
          nombre={pet.nombre}
          descripcion={pet.descripcion}
          edad={pet.edad}
          sexo={pet.sexo}
          image={pet.image}
          deletePet={deletePet}
     />

                    ))
                }
                </div>
            </div>
        </>

    )
}
```

Lo primero que realiza este componente, gracias al hook

*useEffect* es correr la función *listPets()* la cual hace una petición a la API para que nos entregue todos las mascotas registradas, una vez obtenidas define el estado del componente con estas mascotas.

Tiene una función llamada *deletePet* que realizará la eliminación de la mascota requerida.

Lo que retorna el componente es una lista de mascotas o de eventos, según sea el caso, pasándoles como parámetro cada uno de los datos obtenidos en la primera petición.

### 6.2.3. CreateEvent y CreatePet

De igual manera que con los componentes anteriores, estos dos realizan la misma función, solo que cambian los datos que son manejados, pero el funcionamiento es el mismo. La definición es la siguiente.

```
const CreateEvent = () => {

    const [nombre, setNombre] = useState("");
    const [fecha, setFecha] = useState("");
    const [descripcion, setDescripcion] =
useState("");
    const [imagen, setImagen] =
useState(null);
    const handleSubmit = () => {

        const requestOptions = {
            method: 'POST',
            headers: { 'Content-Type':
'application/json' },
            body: JSON.stringify({
                "nombre": nombre,
```

```
                "descripcion": descripcion,
                "fecha": fecha,
                "image": imagen
            })
        };


fetch("http://localhost:3030/api/addevent",
requestOptions)
        .then(response => response.text())
        .then(result => console.log(result))
        .catch(error => console.log('error',
error));
    }
    return(
        <>
            <Header />
            <div className="container">
                <div className="row mt-3 mb-3
justify-content-center">
                    <h2
className="primary">Agenda un nuevo
evento</h2>
                </div>
                <form onSubmit={handleSubmit}>
                    <div className="form-
group">
                        <label
htmlFor="nombre" className="primary">Nombre
del evento:</label>
                        <input type="text"
name="nombre" className="form-control"
```

```
                                onChange={(e) =>
setNombre(e.target.value)}
                                required
                            />
                        </div>
                        <div className="form-
group">
                            <label htmlFor="fecha"
className="primary">Fecha del evento:</label>
                            <input type="text"
name="fecha" className="form-control"
                                onChange={(e) =>
setFecha(e.target.value)}
                                required
                            />
                        </div>
                        <div className="form-
group">
                            <label
htmlFor="descripcion"
className="primary">Descripción del
evento:</label>
                            <textarea
name="descripcion" className="form-control"
                                onChange={(e) =>
setDescripcion(e.target.value)}
                                required
                            />
                        </div>
                        <div className="form-
group">
                            <label
```

```
htmlFor="descripcion" className="primary">Link
de imagen del evento:</label>
                        <input type="text"
name="descripcion" className="form-control"
                            onChange={(e) =>
setImagen(e.target.value)}
                            required
                        />
                    </div>
                    <button type="submit"
className="btn btn-primary sticky-
button">Agendar evento</button>
                </form>
            </div>


        </>
    )
}
```

Lo que hacen estos componentes es muy sencillo, nos muestran un formulario con la información correspondiente que tendremos que introducir para registrar un nuevo evento o una nueva mascota, introducida la información y una vez se de click en el botón que se muestra al final, se ejecutará la función *handleSubmit* la cual hará la petición para que los datos sean introducidos a la base de datos.

### 6.2.4. EditEvent y EditPet

Estos dos componentes son prácticamente iguales a los dos anteriores, realizan las mismas funciones solo que poseen una función extra que permite que se nos muestren los datos de la mascota o el evento en el formulario en vez de que aparezcan

vacíos. La función es la siguiente.

```
const getPet = async () => {
        const respuesta = await fetch(`http://localhost:3030/api/pet/${id}`);
        const mascota = await respuesta.json();
        const {nombre, descripcion, edad, sexo, image} = mascota;
        setNombre(nombre);
        setDescripcion(descripcion);
        setEdad(edad);
        setSexo(sexo);
        setImagen(image);
}
```

Lo que hace esta función, sea el caso de eventos o de mascotas, es realizar una petición a la API del objeto por medio del ID que llega en la URL, una vez que se obtenga la respuesta, los datos obtenidos formarán parte del estado del componente y serán mostrados en los campos correspondientes del formulario.

## 7. Referencias


- MongoDB: https://docs.mongodb.com/
- Bootstrap: https://getbootstrap.com/docs/4.6/getting-started/introduction/
- ReactJS: https://es.reactjs.org/
- Mongoose: https://mongoosejs.com/
- React Router DOM: https://v5.reactrouter.com/web/guides/quick-start