



PROCESO DE GESTIÓN DE FORMACIÓN PROFESIONAL INTEGRAL

FORMATO GUÍA DE APRENDIZAJE

IDENTIFICACIÓN DE LA GUÍA DE APRENDIZAJE

- **Denominación del Programa de Formación:** TGO Análisis y Desarrollo de Sistemas de Información
- **Código del Programa de Formación:** 228106 V102
- **Nombre del Proyecto:** Construcción de un sistema de información que cumpla con los requerimientos del cliente en procesos que se lleven a cabo en el sector productivo del departamento de Caldas
- **Fase del Proyecto:** IMPLEMENTACIÓN
- **Actividad de Proyecto:** Seleccionar la alternativa de solución que cumpla con los requerimientos establecidos por el cliente
- **Competencia:** Construir el sistema que cumpla con los requisitos de la solución informática.
- **Resultados de Aprendizaje Alcanzar:** Realizar la codificación de los módulos del sistema y el programa principal, a partir de la utilización del lenguaje de programación seleccionado, de acuerdo con las especificaciones del diseño
- **Duración de la Guía:** 40 horas

2. PRESENTACIÓN

En esta guía los aprendices con el acompañamiento del instructor, desarrollarán ejemplos y proyectos utilizando herramientas tecnológicas de vanguardia con la arquitectura **API Rest**. Se desarrollará una aplicación que permita desarrollar habilidades específicas en el área Web integrando diversos componentes inicialmente desde el área del Backend.

Rest: Representational State Transfer

API: Application Programming Interface

3. FORMULACIÓN DE LAS ACTIVIDADES DE APRENDIZAJE

- **Descripción de la(s) Actividad(es)**
 - **Actividades de aprendizaje**
 - Creación de Servicios para comunicación con el Backend
 - Utilización del ruteo (Routing) para movernos entre los diferentes componentes
 - Creación de Formularios
 - Creación de API Rest utilizando Express y Node
 - Utilización de base de datos para la persistencia de datos



- **Actividad de Reflexión inicial**

Actividad de reflexión inicial

Antes de comenzar los diferentes proyectos, los aprendices deberán conocer la terminología básica de API Rest, complementando los conceptos fundamentales que se han visto en las guías pasadas. Para ello se deberán analizar los siguientes videos:

¿Que son las APIs y para qué sirven?

<https://www.youtube.com/watch?v=u2Ms34GE14U>

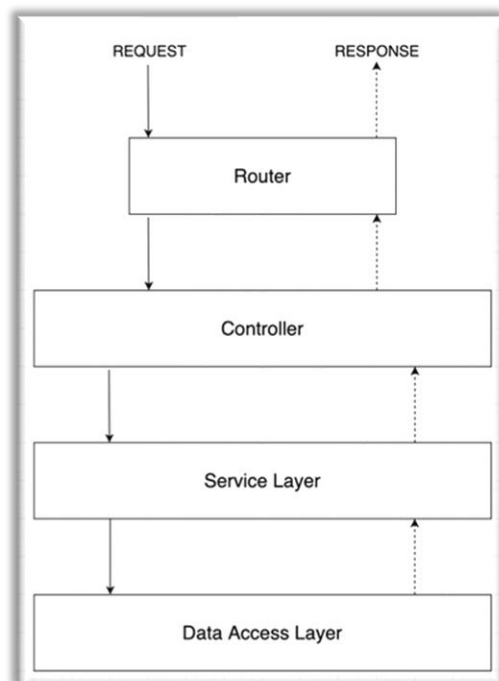
REST y RESTful APIs | Te lo explico en 5 minutos!

<https://www.youtube.com/watch?v=JD6VNRdGI98>

- **Actividades de contextualización e identificación de conocimientos necesarios para el aprendizaje**

Implementando buenas prácticas en nuestra API

Antes de continuar codificando nuestra API, y con el fin de implementar buenas prácticas en la misma, debemos conocer la arquitectura de software que vamos a implementar (Para nuestro caso: Arquitectura de 3 capas)





Router: está directamente asociado a las peticiones del cliente, es un pequeño enrutador de Express que pasa los **requests** (solicitudes) al controller correspondiente.

Controller: Dentro del controlador manejaremos todo lo relacionado con **HTTP**. Eso significa que estamos tratando con **requests** (solicitudes) y **responses** (respuestas) para nuestros endpoints.

Service Layer: Toda la lógica del negocio estará en la capa de Servicio que exporta ciertos servicios (métodos) que utiliza el controlador.

Data Access Layer: esta es la capa de acceso a datos donde trabajaremos con nuestra base de datos. Exportaremos algunos métodos para ciertas operaciones de base de datos, como por ejemplo crear un Usuario que pueda ser utilizado por nuestra capa de servicio.

Versionamiento

Es importante tener en cuenta el control de versiones y versionar nuestra API, ya que al igual que en otras aplicaciones, habrá mejoras, nuevas funciones y cosas por el estilo. La ventaja es que podemos trabajar en nuevas funciones o mejoras en una nueva versión mientras los clientes todavía usan la versión actual y no se ven afectados por cambios importantes.

Tampoco obligamos a los clientes a usar la nueva versión de inmediato. Pueden usar la versión actual y migrar por su cuenta cuando la nueva versión sea estable. Las versiones actual y nueva básicamente se ejecutan en paralelo y no se afectan entre sí.

Una buena práctica es agregar un segmento de ruta como **v1** o **v2** en la URL:

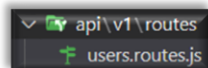
```
// Version 1
"/api/v1/users"

// Version 2
"/api/v2/users"

// ...
```

Para implementar el versionamiento en nuestro proyecto, realizaremos los siguientes cambios:

- Dentro de la carpeta **api**, creamos las subcarpetas llamadas **v1/routes/** y movemos allí el archivo **user.js** y le cambiamos el nombre a **users.routes.js**. Modificamos el archivo **api/v1/users.routes.js** para organizar la ruta de importación de los modelos:



```
1 const db = require('../../../../models');
```

- Modificamos el archivo **server.js** para modificar la ruta existente de usuarios, ya con el versionamiento incluido



```
app.use(bodyParser.json()); //Para que el servidor pueda recibir formato json
app.use(morgan('dev')); // La opción dev da la información principal. Combined da más detalle

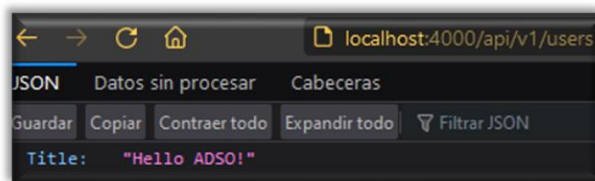
// Routes
app.use('/api/v1/users', require('./api/v1/routes/users.routes')); // Ruta para users con la versión 1 de la API

// Starting de Server
app.listen(app.get('port'), () => { // Se inicia el servidor en el Puerto configurado
```

Acabamos de mover nuestra carpeta de **rutas** a nuestro directorio **v1**. Las otras carpetas como **controladores** o **servicios** aún permanecen dentro de nuestro directorio raíz, ya que para el tipo de API que estamos realizando, es suficiente que sean versiones globales.

Cuando la API vaya creciendo y requiera diferentes métodos de controlador específicos para **v2**, en ese momento sería buena idea mover la carpeta de controladores al directorio v2 para tener toda la lógica específica para esa versión en particular encapsulada, y también se evaluaría si es mejor mover también la carpeta de servicios.

Ahora podemos ensayar nuestra API en el navegador con la nueva ruta:



Nombre Recursos en Plural

Resulta buena práctica nombrar los recursos en plural, para un entendimiento más adecuado de las personas que vayan a utilizar la API.

Después de configurarlo todo, ahora podemos sumergirnos en la implementación real de nuestra API. Vamos a comenzar a implementar **endpoints** para crear, leer, actualizar y eliminar usuarios. Primero, conectemos un controlador, un servicio y un enrutador específicos para nuestros usuarios.

Primero, conectemos un controlador, un servicio y un enrutador específicos para nuestros usuarios. Para ello creamos las carpetas **controllers** y **services**

Después de crear estas carpetas, creamos los siguientes archivos:

- **touch** controllers/userController.js
- **touch** services/userService.js

Modificamos el archivo: **api/v1/routes/user.routes.js** para organizar los endpoints de usuarios. En la anterior guía vimos como a través de los nombres de los endpoints, podíamos identificar de que se trataba cada acción y ver diferentes ejemplos directamente desde el archivo **user.routes.js**, ahora con nuestra nueva arquitectura de proyecto, nos vamos a enfocar en generar rutas más estándares y adecuadas con API Restfull; de esta manera modificaremos este archivo para incluir los llamados a nuestro nuevo **Controlador** de usuarios:



```
1 const {Router} = require('express');
2 const userController = require('../../../../controllers/userController');
3
4 // Creamos el router para poder usar los verbos HTTP
5 const router = Router(); // Llamamos al método Router de Express
6
7 router.get("/", userController.getAllUsers);
8
9 router.get('/:userId', userController.getUser);
10
11 router.post('/', userController.createUser);
12
13 router.put('/:userId', userController.updateUser);
14
15 router.delete('/:userId', userController.deleteUser);
16
17 module.exports = router;
```

Después vamos a organizar el código de nuestro controlador `controllers/userController.js` que a su vez va estar enlazado con nuestra capa de servicio. También tendremos en cuenta los códigos **http** en la respuesta que haga el servidor según la peticiones correspondientes:

Código inicial

```
// Enlazamos nuestro servicio
const userService = require('../services/userService');
```

Obtener todos los usuarios (GET)

```
const getAllUsers = async (req, res) => {
  const allUsers = await userService.getAllUsers();

  if(allUsers)
    res.status(200).send({ status: "OK", data: allUsers });
  else
    res.status(400).send({ status: "FAILED", data: allUsers });
};
```

Obtener un usuario específico (GET)



```
const getUser = async (req, res) => {
  let id = req.params.userId;
  const user = await userService.getUser(id);
  if(user)
    res.status(200).send({ status: "OK", data: user });
  else
    res.status(400).send({ status: "FAILED", data: user });
};
```

Crear un usuario (POST)

```
const createUser = async (req, res) => {
  const {body} = req;
  const createdUser = await userService.createUser(body.name, body.email, body.password);
  if(createdUser)
    res.status(201).send({ status: "OK", data: createdUser });
  else
    res.status(400).send({ status: "FAILED", data: createdUser });
};
```

Modificar un usuario (PUT)

```
const updateUser = async (req, res) => {
  let id = req.params.userId;
  let {name,email,password} = req.body;
  const updatedUser = await userService.updateUser(id,name,email,password);
  if(updatedUser)
    res.status(200).send({ status: "OK", data: updatedUser });
  else
    res.status(400).send({ status: "FAILED", data: updatedUser });
};
```

Eliminar un usuario (DELETE)

```
const deleteUser = async (req, res) => {
  let id = req.params.userId;
  const deletedUser = await userService.deleteUser(id);
  if(deletedUser)
    res.status(200).send({ status: "OK", data: deletedUser });
  else
    res.status(400).send({ status: "FAILED", data: deletedUser });
};
```

Por último la exportación de los módulos



```
module.exports = {  
  getAllUsers,  
  getUser,  
  createUser,  
  updateUser,  
  deleteUser,  
};
```

Después de realizar estos cambios, podemos configurar nuestro servicio de usuarios `services/userServices.js`

Código Inicial

```
const db = require('../models');
```

Obtener todos lo usuarios

```
const getAllUsers = async () => {  
  try {  
    let users = await db.User.findAll();  
    return users;  
  } catch (error) {  
    return error.message || "Failed to get users";  
  }  
};
```

Obtener un usuario específico

```
const getUser = async (id) => {  
  try {  
    let user = await db.User.findByPk(id);  
    return user;  
  } catch (error) {  
    return error.message || "Failed to get user";  
  }  
};
```

Crear un usuario

```
const createUser = async (name, email, password) => {  
  try {  
    let newUser = await db.User.create({  
      name,  
      email,  
      password,  
    });  
    return newUser;  
  } catch (error) {  
    return error.message || "User could not be created";  
  }  
};
```



Actualizar un usuario

```
const updateUser = async (id,name,email,password) => {
  try {
    let updatedUser = await db.User.update({
      name,
      email,
      password
    }, {
      where: {
        id,
      }
    });
    return updatedUser;
  } catch (error) {
    return error.message || "User could not be updated";
  }
};
```

Eliminar un usuario

```
const deleteUser = async (id) => {
  try {
    const deletedUser = await db.User.destroy({
      where: {
        id,
      }
    });
    return deletedUser;
  } catch (error) {
    return error.message || "User could not be deleted";
  }
};
```

Exportación de módulos

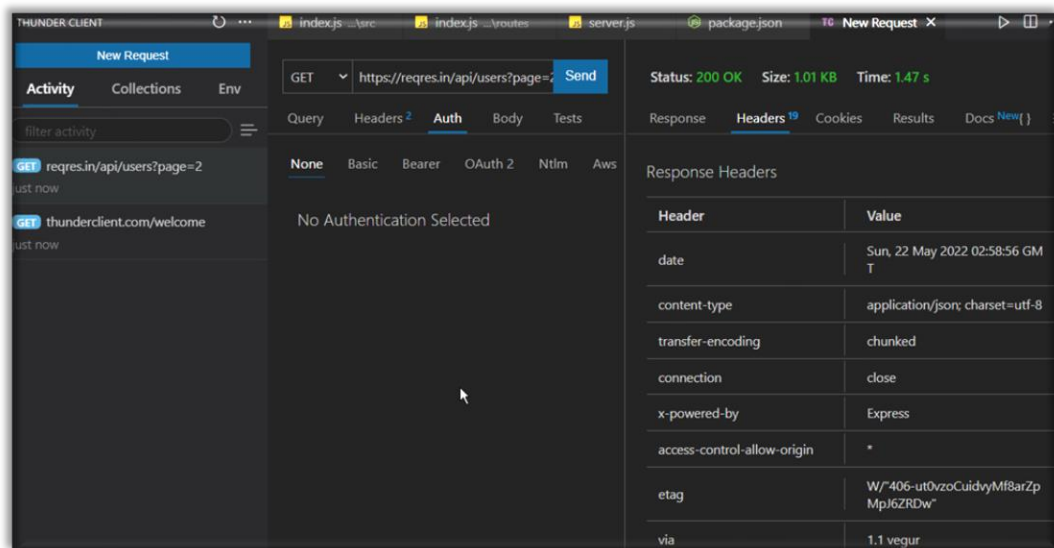
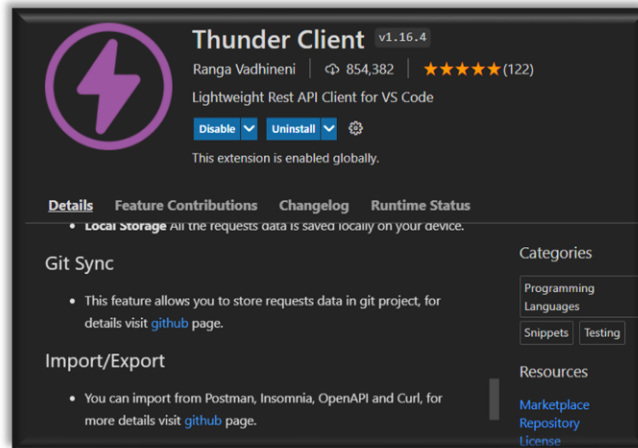
```
module.exports = {
  getAllUsers,
  getUser,
  createUser,
  updateUser,
  deleteUser,
};
```

Probando nuestra API

Thunder Client

Thunder Client es una extensión de cliente HTTP bastante ligera para Visual Studio Code, fue hecha por Ranga Vadhineni como alternativa a POSTMAN. Thunder Client tiene un diseño simple y limpio y permite realizar los llamados a nuestra API desde el mismo ambiente de desarrollo.

- <https://www.thunderclient.com>
- <https://www.youtube.com/watch?v=NKZ0ahNbmak>



También existe otra alternativa llamada Insomnia que puede visualizarse en el siguiente enlace:

<https://docs.insomnia.rest/insomnia/get-started>

Obteniendo todos los usuarios



GET <http://localhost:4000/api/v1/users> Send

Status: 200 OK Size: 465 Bytes Time: 15 ms

Query Headers² Auth Body¹ Tests

Response Headers⁶ Cookies Results Docs [New](#)

```
1 {
2   "status": "OK",
3   "data": [
4     {
5       "id": 1,
6       "name": "Julian",
7       "email": "julian@sena.edu.co",
8       "password": "654321",
9       "createdAt": "2022-05-05T06:42:30.000Z",
10      "updatedAt": "2022-05-05T06:42:30.000Z"
11    },
12    {
13      "id": 2,
14      "name": "Ramon",
15      "email": "ramon@gmail.com",
16      "password": "12345",
17      "createdAt": "2022-05-11T05:00:47.000Z",
18      "updatedAt": "2022-05-11T05:39:45.000Z"
19    },
20    {
21      "id": 3,
22      "name": "Ana",
23      "email": "ana@gmail.com",
24      "password": "test",
25      "createdAt": "2022-05-11T05:02:39.000Z",
26      "updatedAt": "2022-05-11T05:02:39.000Z"
27    }
28  ]
29 }
```

Obteniendo un usuario

GET <http://localhost:4000/api/v1/users/2> Send

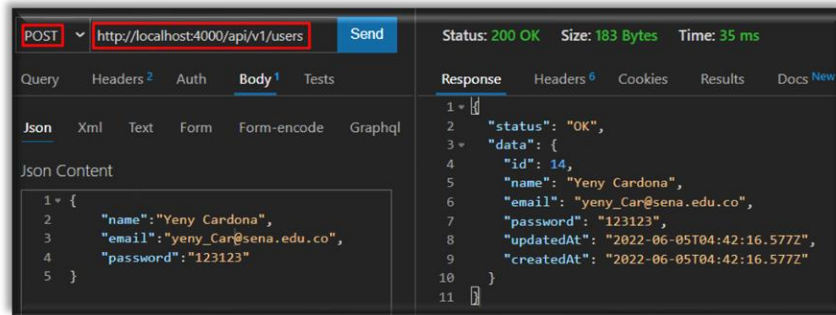
Status: 200 OK Size: 169 Bytes Time: 19 ms

Query Headers² Auth Body¹ Tests

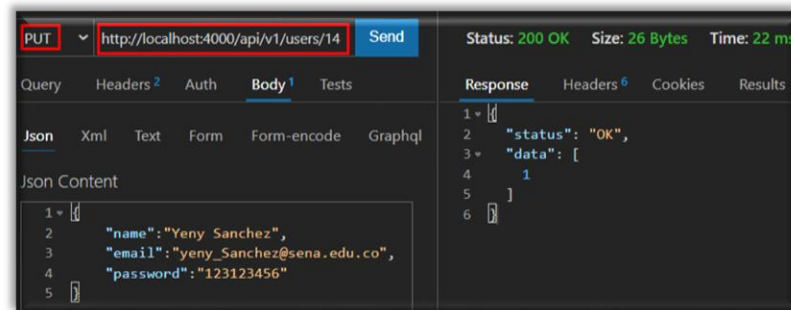
Response Headers⁶ Cookies Results Docs [New](#)

```
1 {
2   "status": "OK",
3   "data": {
4     "id": 2,
5     "name": "Ramon",
6     "email": "ramon@gmail.com",
7     "password": "12345",
8     "createdAt": "2022-05-11T05:00:47.000Z",
9     "updatedAt": "2022-05-11T05:39:45.000Z"
10  }
11 }
```

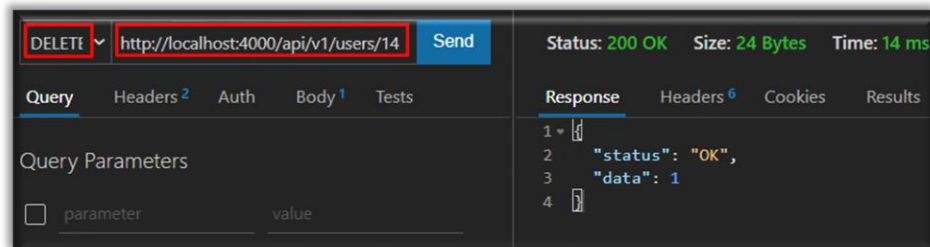
Creando un nuevo usuario



Actualizando Usuario



Eliminando usuario



Personalizando Errores

En estos momentos nuestra API devuelve los errores tal cual como los genera nuestro ORM Sequelize. Podemos personalizar un poco más nuestros errores de la siguiente manera para que se vean de una forma más adecuada en la respuesta de nuestra API en caso de que tengamos que mostrar un error:

En el archivo `UserController.js` ponemos lo siguiente:



```
const getUser = async (req, res) => {  
  let id = req.params.userId;  
  try {  
    const user = await userService.getUser(id);  
    res.status(200).send({ status: "OK", data: user });  
  } catch (error) {  
    res.status(error.status || 500).send({ status: "FAILED", data: { error: error.message } });  
  }  
};
```

En el archivo `userService.js` ponemos lo siguiente:

```
const getUser = async (id) => {  
  try {  
    let user = await db.User.findByPk(id);  
    return user  
  } catch (error) {  
    throw {status: 500, message: error.message || "Failed to get user" };  
  }  
};
```

Actividad 01

Los aprendices deberán modificar los demás métodos del **controlador** y **servicio** de usuario, con el fin de implementar el cambio del ajuste de los errores y manejo del try...catch

Actividad 02

Para esta actividad los aprendices deberán modificar los métodos necesarios para que nuestra API, al recibir los datos para **crear un usuario**, valide que lleguen todos los campos obligatorios, y en caso de que alguno falte, envíe una respuesta con el error correspondiente y con el código 400.



Evitar los verbos en los nombres de los endpoints

No tiene mucho sentido usar verbos dentro de los endpoints y, de hecho, es bastante inútil. En general, cada URL debe apuntar hacia un recurso (solo eso). El uso de un verbo dentro de una URL muestra un cierto comportamiento que un recurso en sí mismo no puede tener.

Con la implementación de esta guía, ya implementamos los endpoints correctamente sin usar verbos dentro de la URL, pero comparemos cómo se vería nuestra URL de una forma y la otra.

// Implementaciones actuales (sin verbos)

```
GET "/api/v1/users"  
GET "/api/v1/users/:userId"  
POST "/api/v1/users"  
PUT "/api/v1/users/:userId"  
DELETE "/api/v1/users/:userId"
```

// Implementaciones usando verbos

```
GET "/api/v1/getAllUsers"  
GET "/api/v1/users/getUserById/:userId"  
POST "/api/v1/createUser"  
PUT "/api/v1/users/updateUser/:userId"  
DELETE "/api/v1/users/deleteUser/:userId"
```

Tener una URL completamente diferente para cada endpoint puede volverse confuso e innecesariamente complejo. Y si tenemos por ejemplo 300 endpoints se volvería algo demasiado difícil de manejar y documentar.

Otra razón para no usar verbos dentro de la URL es que el **verbo HTTP** en sí mismo ya indica la acción. Cosas como "GET /api/v1/getUsers" o "DELETE api/v1/deleteUser/usertId" son innecesarias.

- **Actividades de transferencia del conocimiento**



Los aprendices deberán demostrar los conocimientos conceptuales y procedimentales que adquirieron a partir de las actividades de apropiación mediante el siguiente instrumento de evaluación:

Los aprendices deberán realizar cada una de las actividades propuestas para adquirir dominio en los conocimientos relacionados con los diferentes temas. Se evaluarán los conocimientos adquiridos mediante instrumentos de evaluación de Desempeño y Producto relacionados en la presente guía

- **Ambiente Requerido**

Ambiente de SISTEMAS con conexión eléctrica e internet

- **Materiales**

- Computadores (30)
- Sillas (3)
- Televisor (1)
- Resma tamaño carta (1)
- Marcadores (3)
- Lápiz (1)
- Lapicero (1)

4. ACTIVIDADES DE EVALUACIÓN

Evidencias de Aprendizaje	Criterios de Evaluación	Técnicas e Instrumentos de Evaluación
Evidencias de Conocimiento: Evidencias de Desempeño: Asistencia y participación activa en las diferentes actividades propuestas Evidencias de Producto: Respuestas y procedimiento de los talleres realizados	Crea la base de datos en el motor de base de datos seleccionado, siguiendo especificaciones técnicas del informe, según normas y protocolos de la empresa.	Observación: EXC_D_01 Valoración del Producto: EXC_P_01



5. GLOSARIO DE TÉRMINOS

Sistema de información: es un conjunto de elementos orientados al tratamiento y administración de datos e información, organizados y listos para su uso posterior, generados para cubrir una necesidad o un objetivo

Sistema operativo – Es un conjunto de programas que sirven para manejar un ordenador.

Software - El conjunto de programas, procedimientos y documentación asociado a un sistema informático.

Javascript: es un lenguaje de programación del lado del cliente que se utiliza con frecuencia en diseño WEB para generar efectos más complejos que no se puedan alcanzar usando HTML.

HTML: Siglas de las palabras inglesas: Hypertext Markup Language. Es decir, lenguaje de marcado de hipertexto. Lenguaje informático para crear páginas web. Conjunto de etiquetas o instrucciones que permiten estructurar el contenido de una web e incluir los hipervínculos o enlaces a otras páginas. Este lenguaje lo inventó en 1991 el Doctor Berners-Lee del CERN en Suiza.

HTTPS: Siglas de las palabras inglesas: HyperText Transfer Protocol Secure o versión segura del protocolo HTTP. Es el protocolo empleado para la transferencia de ficheros HTML cifrados que puedan contener información confidencial.

HTTP: siglas de las palabras inglesas: Hypertext Transfer Protocol. A saber en español: Protocolo de Transmisión de Hipertexto. Protocolo estándar de transferencia de hipertexto. Es decir: el protocolo de comunicaciones en el que está basada la Word Wide Web.

Script: es un archivo de órdenes o archivo de procesamiento por lotes. Es un programa usualmente simple, que por lo regular se almacena en un archivo de texto plano.

MySQL: es un sistema de gestión de bases de datos de código abierto que, junto con PHP, permite darle a las páginas web cierto dinamismo, es decir, disponer de manera adecuada los datos solicitados por los navegadores. Es un sistema multiplataforma y su uso está tan extendido en las bases de datos que podría considerarse un estándar.

SEO (Search Engine Optimisation) Optimización en buscadores: técnica utilizada para asegurar que una página Web es compatible con los motores de búsqueda y así tener la posibilidad de aparecer en las posiciones más altas en los resultados de búsqueda.

Diseño web adaptable (responsive web design): se llama así al diseño web de aquellas páginas que se adaptan al tamaño de la pantalla o ventana en que se despliegan, por medio del uso de, idealmente, un solo documento HTML y un solo documento CSS. Esto permite hacer una sola página web para smartphones, phablets, tablets y PC.

Diagrama o Modelo Entidad Relación (DER): denominado por sus siglas en inglés, E-R "Entity relationship", o del español DER "Diagrama de Entidad Relación") es una herramienta para el modelado de datos que permite representar las entidades relevantes de un sistema de información así como sus interrelaciones y propiedades



Bases de Datos (BD): es un banco de información que contienen datos relativos a diversas temáticas y categorizados de distinta manera, pero que comparten entre sí algún tipo de vínculo o relación que busca ordenarlos y clasificarlos en conjunto.

6. REFERENTES BIBLIOGRÁFICOS

- Documentos técnicos relacionados en la plataforma
- API Documentation
 - <https://swagger.io/tools/swaggerhub/>
 - <https://www.javacodegeeks.com/api-documentation-in-node-js-using-swagger.html>
 - <https://javascript.plainenglish.io/how-to-implement-and-use-swagger-in-nodejs-d0b95e765245>
 - <https://itnext.io/setting-up-swagger-in-a-node-js-application-d3c4d7aa56d4>
 - <https://petstore.swagger.io/>
 - <http://dapperdox.io/>
- <https://paystack.com/docs/api>
- <https://curl.se/>
- <https://www.baeldung.com/curl-rest>
- <https://www.freecodecamp.org/news/rest-api-design-best-practices-build-a-rest-api/>
- <https://stasklymenko.hashnode.dev/nodejs-roadmap-for-beginners>
- <https://blog.logrocket.com/best-node-js-cms-platforms-2022/>
- <https://blog.shiftleft.io/node-js-vulnerability-cheatsheet-447b0c9bdb99>
- <https://blog.logrocket.com/free-services-deploy-node-js-app/>
- <https://docs.insomnia.rest/insomnia/get-started>
- <https://www.thunderclient.com>
- <https://www.youtube.com/watch?v=NKZ0ahNbmk>
- <https://reqres.in/>
- <https://jsonplaceholder.typicode.com/>
- <https://expressjs.com/es/starter/hello-world.html>
- Sequelize REST API (usando Postgres)
 - <https://www.youtube.com/watch?v=3xiiOgYdbiE>
- <https://dbdiagram.io>
- <https://geekflare.com/es/api-documentation-tools/>
- <https://geekflare.com/es/api-tools/>
- JWT
 - <https://todoxampp.com/jwt-y-nodejs-como-crear-un-sistema-de-autenticacion/>
 - <https://medium.com/geekculture/understanding-jwt-authentication-using-nodejs-express-7c6b41af801b>
 - <https://dev.to/chaocharles/authentication-in-node-js-and-jwt-video-tutorial-1-hour-389>
 - <https://www.codementor.io/@olawalealadeusi896/building-a-simple-api-with-nodejs-expressjs-postgresql-db-and-jwt-3-mke10c5c5>
 - <https://javascript.plainenglish.io/authentication-systems-using-jwt-and-node-js-9c3cc14aaf82>
 - <https://www.javacodegeeks.com/nodejs-jwt-implementation.html>
 - <https://towardsdev.com/jwt-authentication-with-node-js-and-react-dc41ef0e6136>
 - <https://www.digitalocean.com/community/tutorials/nodejs-jwt-expressjs>
 - <https://helewud.com/how-to-build-an-authorization-system-with-jwt-using-nodejs#heading-prerequisites>
 - <https://itnext.io/auth-with-nodejs-express-mongoose-and-jwt-577aa3f2f707>
 - <https://blog.galmalachi.com/react-nodejs-and-jwt-authentication-the-right-way>
 - <https://medium.com/swlh/nodejs-with-jwt-authentication-feb961763541>
 - <https://medium.com/swlh/how-to-build-simple-and-secure-rest-api-for-user-authentication-using-nodejs-jwt-and-mongodb-2bdeb3e5427e>
 - <https://blog.logrocket.com/how-to-implement-jwt-authentication-vue-nodejs/>



- <https://medium.com/geekculture/jwt-jsonwebtoken-token-based-authentication-using-nodejs-6af661d5a64>
- <https://www.telerik.com/blogs/json-web-token-jwt-implementation-using-nodejs>
- <https://javascript.plainenglish.io/how-to-authenticate-and-authorize-using-jwt-with-node-js-a324437dedb1>
- **Seguridad**
 - <https://restfulapi.net/security-essentials/>
- <https://www.digitalocean.com/community/tutorials/como-crear-una-aplicacion-node-js-con-docker-es>
- <https://profile.es/blog/que-es-docker/>
- <https://github.com/expressjs/multer>
- <https://code.tutsplus.com/es/tutorials/file-upload-with-multer-in-node--cms-32088>
- **Chart.js**
 - <https://www.chartjs.org/>
 - <https://code.tutsplus.com/tutorials/getting-started-with-chartjs-introduction--cms-28278>
 - <https://javascript.plainenglish.io/web-based-visualization-with-chart-js-7aa5af9102cd>
 - <https://vuejsexamples.com/a-simple-chartjs-wrapper-for-vue3/>
 - <https://dzone.com/articles/chartjs-example-with-dynamic-dataset>
 - <https://blog.logrocket.com/using-chart-js-react/>
 - <https://code.tutsplus.com/tutorials/getting-started-with-chartjs-line-and-bar-charts--cms-28384>
 - <https://dev.to/changoman/vuejs-and-chartjs-weather-api-example-1e7>
 - <https://vuejsexamples.com/a-simple-wrapper-around-chart-js-3-for-vue-2-3/>
 - <https://www.thepolyglotdeveloper.com/2018/01/use-chartjs-display-attractive-charts-vuejs-web-application/>
 - <https://medium.com/risan/vue-chart-component-with-chart-js-db85a2d21288>
 - <https://blog.logrocket.com/creating-chart-components-in-vue-with-chart-js/>

7. CONTROL DEL DOCUMENTO

	Nombre	Cargo	Dependencia	Fecha
Autor (es)	Julian Salazar Pineda	Instructor	Centro de Procesos Industriales y Construcción	14 Abril de 2022

8. CONTROL DE CAMBIOS (diligenciar únicamente si realiza ajustes a la guía)

Nombre	Cargo	Dependencia	Fecha	Razón del Cambio
--------	-------	-------------	-------	------------------



Autor (es)					
------------	--	--	--	--	--