	INSTITUTO TECNOLOGICO DE TUXTEPEC Departamento de Sistemas y Computación
Formato para prácticas de Laboratorio	

Carrera	Plan de estudios	Clave de la materia	Nombre de la materia	Semestre	Gpo.	Periodo
Ingeniería Informática	IINF-2010-220	7F5 A	Programación en ambiente cliente servidor	7	A	Ago/Dic 2025

Practica No.	Laboratorio de:	Nombre de la práctica	Duración (Hora)
1	SC-07	mini_pokeapi y ¿Quién es el Pokémon?	1

1. Enunciado

- Desarrollar una mini API REST denominada *mini_pokeapi* y una aplicación cliente llamada *¿Quién es el Pokémon?*, aplicando el modelo cliente–servidor, con el fin de consumir servicios web y realizar operaciones CRUD utilizando tecnologías del lado del servidor y del cliente.

2. Introducción

El modelo cliente–servidor es fundamental en el desarrollo de aplicaciones modernas, ya que permite separar la lógica del negocio del acceso a los datos. Las API REST facilitan esta comunicación mediante el uso del protocolo HTTP y el intercambio de información en formato JSON.

En esta práctica se integran dos desarrollos: una *mini_pokeapi*, encargada de la gestión de datos de Pokémon, y una aplicación cliente denominada *¿Quién es el Pokémon?*, la cual consume dicha API para mostrar información de manera dinámica, reforzando así el aprendizaje práctico de servicios web.

3. Objetivo (Competencia)

Implementar un sistema cliente–servidor mediante el desarrollo de una API REST y una aplicación cliente que consuma sus servicios, fortaleciendo las competencias en el manejo de operaciones CRUD, consumo de APIs y arquitectura cliente–servidor.

4. Fundamento

La **arquitectura cliente–servidor** es un modelo de comunicación en el que dos componentes principales interactúan:

- **Cliente**: Solicita servicios o recursos.
- **Servidor**: Proporciona esos servicios o recursos.

Este modelo se basa en la idea de que el cliente inicia la comunicación y el servidor responde. Es ampliamente usado en aplicaciones web, sistemas distribuidos y redes.

Funciones del Cliente	Funciones del Servidor
• **Interfaz de usuario**: Permite la interacción con el sistema. • **Generar solicitudes**: Envía peticiones al servidor (por ejemplo, consultar datos). • **Procesamiento local**: Puede realizar validaciones o cálculos simples. • **Presentación de resultados**: Muestra la información que recibe del servidor.	• **Procesar solicitudes**: Recibe y atiende las peticiones del cliente. • **Gestión de datos**: Accede, almacena y actualiza información en bases de datos. • **Control de seguridad**: Autenticación, autorización y protección de datos. • **Escalabilidad**: Maneja múltiples clientes simultáneamente.

¿Qué es una API?

Una API (Application Programming Interface) **es un conjunto de reglas y protocolos que permite que dos aplicaciones se comuniquen entre sí.**

- Define **cómo** se deben enviar y recibir datos.
- Facilita la integración entre sistemas sin necesidad de conocer la implementación interna.

En el caso de **mini_pokeapi**, la API expone datos de Pokémon para que los clientes (apps web, móviles) puedan consultarlos.

¿Qué es REST?

REST (Representational State Transfer) es un estilo de arquitectura para diseñar APIs que usan el protocolo HTTP. Características principales:

- Basado en **recursos** (por ejemplo, /pokemon, /abilities).
- Usa métodos HTTP estándar: **GET**, **POST**, **PUT**, **DELETE**.
- Es **ligero** y fácil de implementar.

Principios REST

1. **Recursos**
   Todo se representa como un recurso identificado por una **URI** (por ejemplo: https://pokeapi.co/api/v2/pokemon/1).
2. **Stateless**
   Cada petición del cliente al servidor debe contener toda la información necesaria. El servidor **no guarda estado** entre peticiones.
3. **Uso de URI**
   Las URIs son únicas y describen el recurso. Ejemplo:
   - GET /pokemon/25 → Obtiene información de Pikachu.
4. **Operaciones HTTP**
   - **GET**: Consultar datos.
   - **POST**: Crear un recurso.
   - **PUT/PATCH**: Actualizar un recurso.
   - **DELETE**: Eliminar un recurso.

Las operaciones CRUD (Create, Read, Update, Delete)

son las cuatro funciones fundamentales para gestionar datos en bases de datos y aplicaciones, permitiendo a los usuarios interactuar con la información: Crear nuevos registros, Leer (o consultar) existentes, Actualizar información y Eliminar registros, siendo la base de la manipulación de datos en cualquier sistema, desde un simple cuaderno hasta APIs RESTful.

¿Qué es JSON?

JSON (JavaScript Object Notation) es un formato ligero para el intercambio de datos. Se basa en texto y es fácil de leer y escribir para humanos, además de ser sencillo de

interpretar por máquinas. Se utiliza ampliamente para transmitir datos entre un servidor y una aplicación web.

Estructura básica	Ventajas en APIs

Un documento JSON se compone de:
• **Objetos**: Con llaves `{ }`, contienen pares **clave: valor**.
• **Arreglos**: Con corchetes `[ ]`, contienen listas de valores.
• **Tipos de datos**:
  ○ **String**: `"texto"`
  ○ **Número**: `123`
  ○ **Booleano**: `true` / `false`
  ○ **Null**: `null`
  ○ **Objeto**: `{ "clave": "valor" }`
  ○ **Arreglo**: `[ "valor1", "valor2" ]`
	• **Ligero y rápido**: Ideal para transmitir datos por HTTP.
	• **Compatible**: Funciona con casi todos los lenguajes de programación.
	• **Estandarizado**: Fácil de integrar en servicios REST y aplicaciones web.
	• **Legible**: Tanto para humanos como para máquinas.
	• **Flexible**: Permite estructuras complejas (objetos dentro de objetos, listas, etc.).

¿Qué es Node.js?

Node.js es un **entorno de ejecución** para JavaScript que se ejecuta en el **lado del servidor**. Está construido sobre el motor **V8 de Google Chrome**, lo que le permite ejecutar código JavaScript fuera del navegador.

Características	Uso en servidores
• **Asíncrono y no bloqueante**: Utiliza un modelo basado en eventos, ideal para aplicaciones que manejan muchas conexiones simultáneas. • **Alto rendimiento**: Gracias al motor V8 y su arquitectura orientada a eventos. • **Single-thread**: Opera en un solo hilo, pero maneja múltiples operaciones concurrentes mediante callbacks y promesas.	Node.js se utiliza principalmente para: • **Servidores web**: Crear APIs REST y aplicaciones en tiempo real. • **Aplicaciones en tiempo real**: Chats, juegos online, streaming. • **Microservicios**: Arquitecturas distribuidas y escalables.

• **Gran ecosistema**: A través de **npm (Node Package Manager)**, ofrece miles de librerías. • **Multiplataforma**: Funciona en Windows, Linux y macOS.	• **Integración con bases de datos**: MongoDB, MySQL, PostgreSQL, etc.

¿Qué es Express?

Express es una librería minimalista que proporciona herramientas para gestionar **rutas**, **peticiones HTTP**, **middlewares** y **respuestas** en aplicaciones Node.js. Es muy popular para desarrollar **APIs REST** y aplicaciones web escalables.

Endpoints y Rutas

- **Rutas**: Son las direcciones dentro de una aplicación que permiten acceder a recursos o ejecutar acciones.
  Ejemplo: /usuarios, /productos/123.
- **Endpoints**: Son puntos específicos de acceso a la API que responden a una petición HTTP (GET, POST, PUT, DELETE).
  Cada endpoint está asociado a una **ruta** y un **método HTTP**.

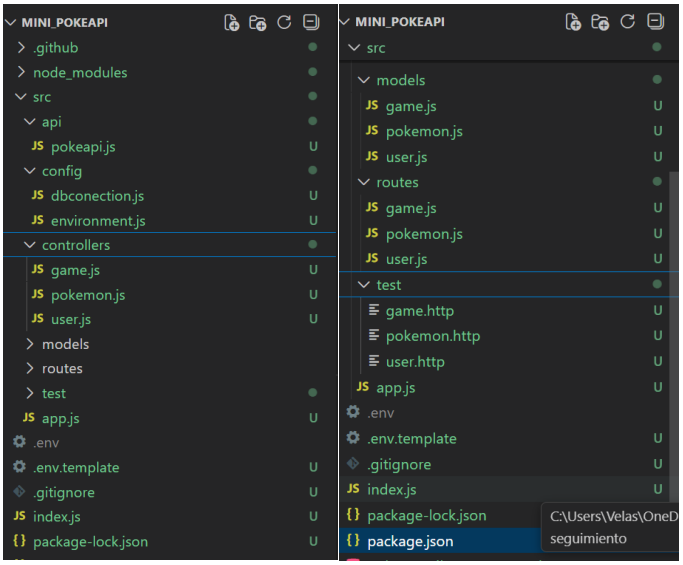
5. Descripción (Procedimiento)

A) Equipo necesario	**Material de apoyo**
○ Computadora personal ○ Sistema operativo Windows	○ Visual Studio Code ○ Navegador web ○ Node.js

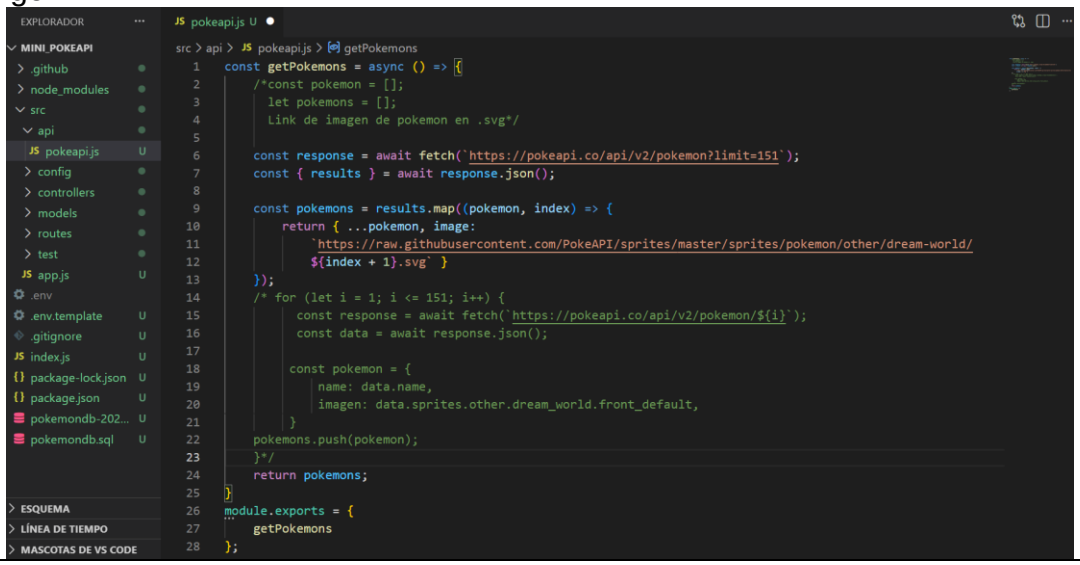
1. Desarrollo de la práctica

Parte 1: Mini PokeAPI

Se creó la estructura del proyecto **MINI_POKEAPI**. se desarrolló una API REST utilizando **Node.js** y el framework **Express**, el cual permitió la creación de rutas HTTP para el acceso a los datos. Para la conexión con la base de datos se empleó el módulo **mariadb**, facilitando la ejecución de consultas SQL. Posteriormente se creó una carpeta llamada src en donde se encontraran las siguientes carpetas y sus archivos con extensiones .js y .http mostrado en las imágenes, dentro de la carpeta de MINI_POKEAPI se debe de encontrar el siguiente archivo .env, .env.template y el index.js.

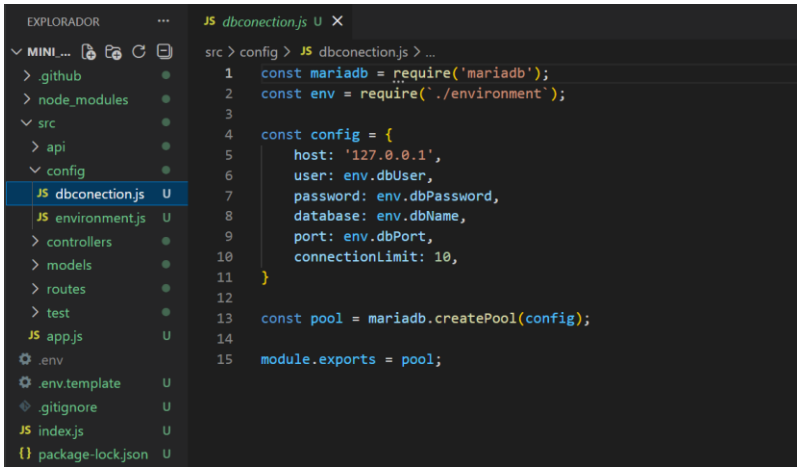


Empezando con la carpeta **api** y el archivo **pokeapi.js** se mostrará el siguiente código:



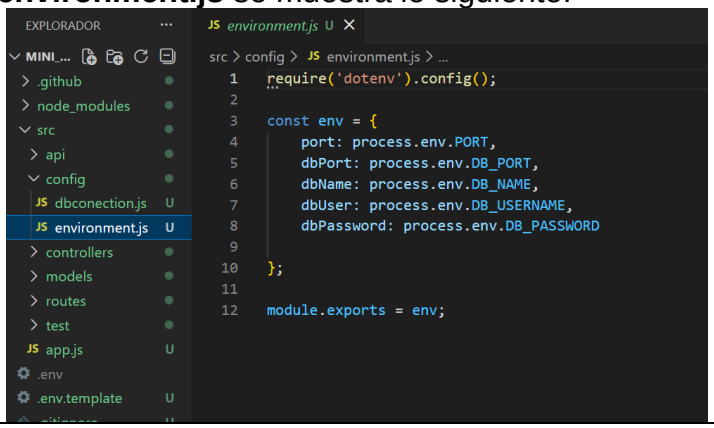
El archivo **pokeapi.js** define una función asíncrona llamada **getPokemons** que consume la PokeAPI para obtener los primeros **151 Pokémon**. Mediante una petición HTTP, se recupera la lista básica de Pokémon y se procesa usando **map()** para agregar a cada uno una imagen en formato SVG proveniente del repositorio oficial de sprites de PokeAPI. Finalmente, la función retorna un arreglo con los datos del Pokémon (nombre y URL) junto con su imagen y se exporta para ser utilizada en otras partes de la aplicación, como controladores o rutas de una API REST.

Carpeta **config** y el archivo **dbconection.js** se muestra el siguiente código:



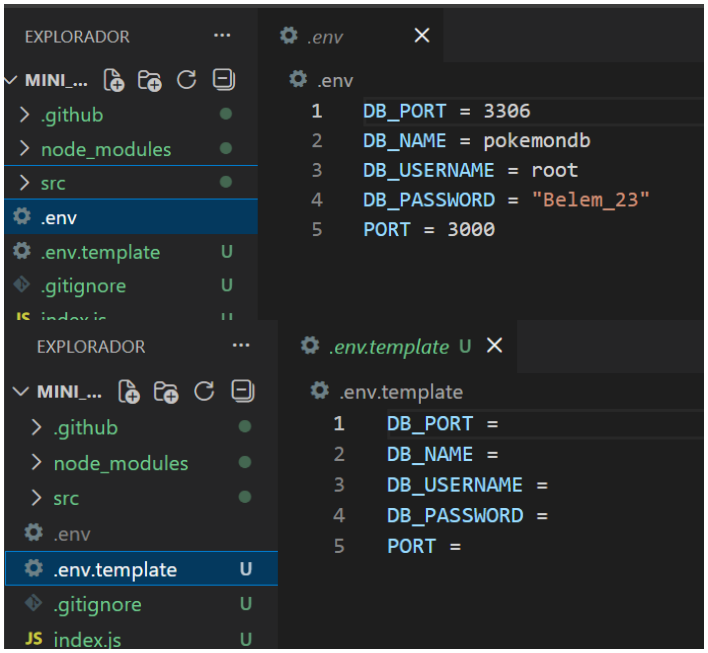
En donde el archivo **dbconection.js** configura la conexión a una base de datos MariaDB utilizando un pool de conexiones. Los datos sensibles se cargan desde variables de entorno, lo que mejora la seguridad y facilita la configuración. Este módulo permite que toda la aplicación acceda de forma eficiente y controlada a la base de datos.

Base al archivo **environment.js** se muestra lo siguiente:



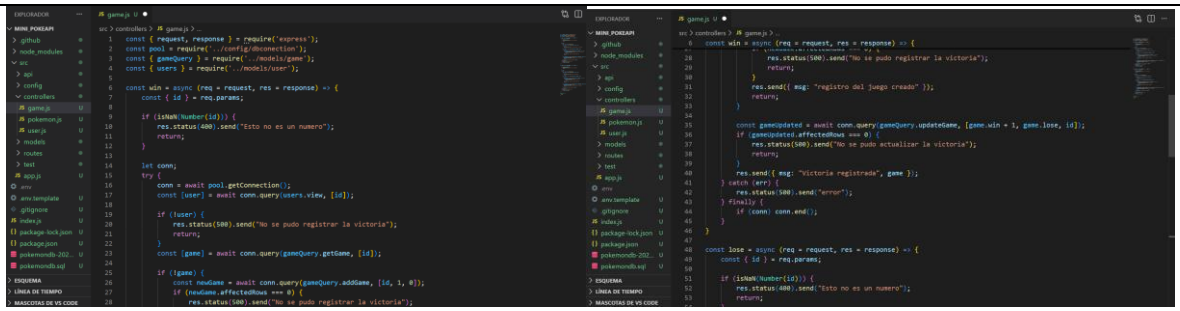
El archivo environment.js carga las variables de entorno desde el archivo **.env** utilizando la librería dotenv y las centraliza en un objeto exportable. Esto permite configurar el puerto del servidor y los datos de conexión a la base de datos de forma segura, flexible y sin exponer información sensible en el código.

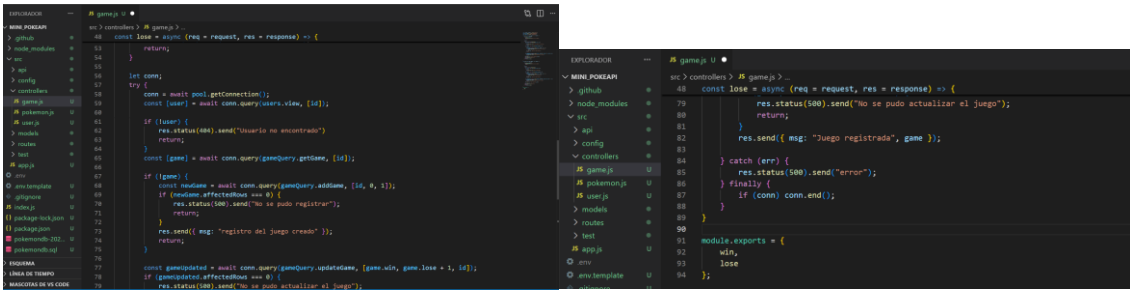
El archivo **.env** muestra lo siguiente a lo que también se relaciona con el archivo **.env.templade**



Por su parte, el archivo **.env** contiene los valores reales de configuración, como el puerto de la base de datos, el nombre de la base de datos, el usuario, la contraseña y el puerto del servidor. Estos valores son leídos por environment.js en tiempo de ejecución. Asimismo, el archivo **.env.template** funciona como una **plantilla de referencia**, mostrando la estructura de las variables de entorno que deben definirse, pero sin incluir datos reales. Esto facilita la configuración del proyecto en distintos entornos y evita compartir información sensible en repositorios públicos.

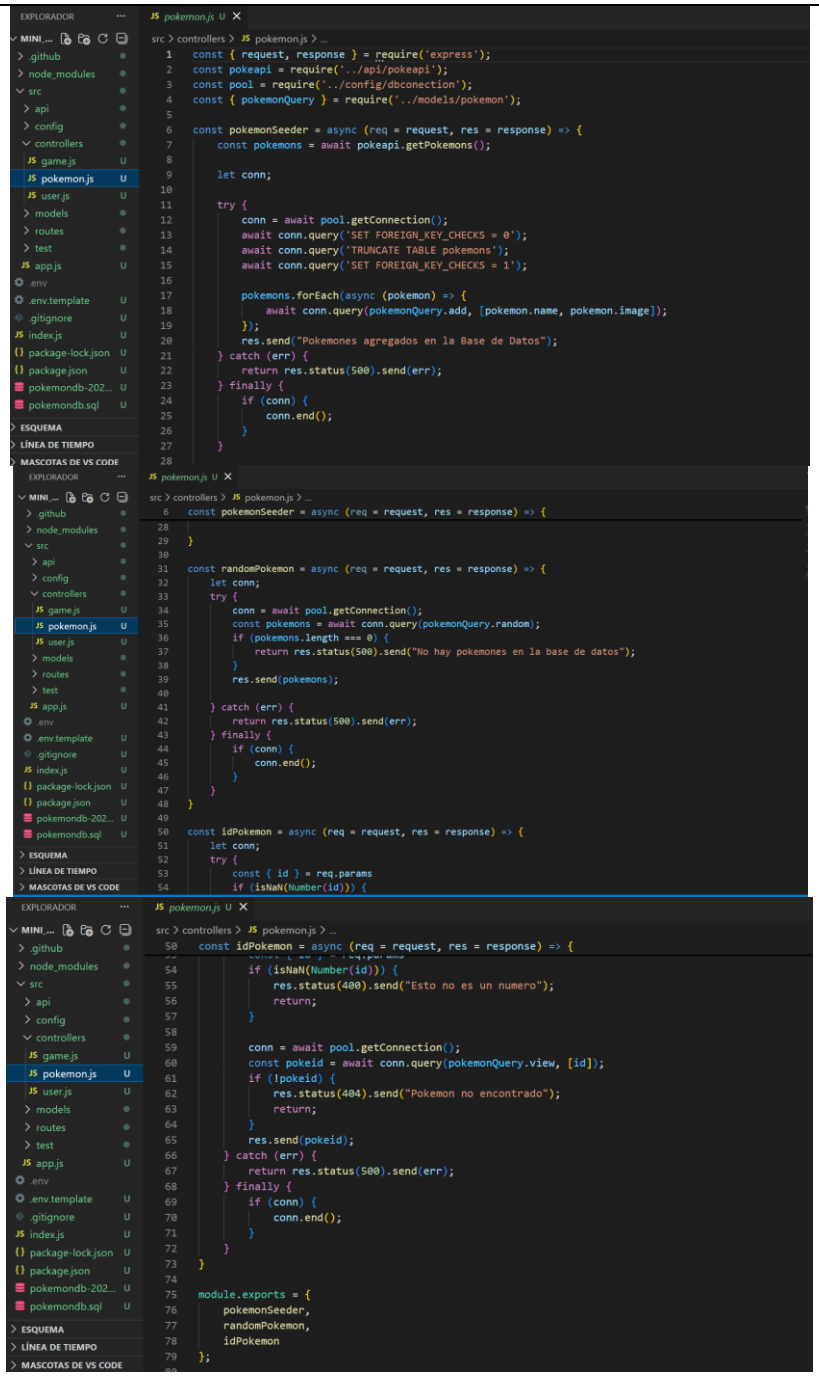
Posteriormente la carpeta **controllers** en donde se encuentra los archivos **game.js, pokemon.js y user.js**





Este archivo corresponde al archivo **game.js** encargado de manejar la lógica para registrar **victorias y derrotas** de los usuarios en la base de datos. Las funciones **win** y **lose** reciben el identificador del usuario desde los parámetros de la solicitud **HTTP** y, antes de realizar cualquier operación, validan que dicho identificador sea numérico, evitando errores y peticiones inválidas. Posteriormente, el controlador establece una conexión con la base de datos mediante un **pool de conexiones**, lo que permite un acceso eficiente y controlado a MariaDB.

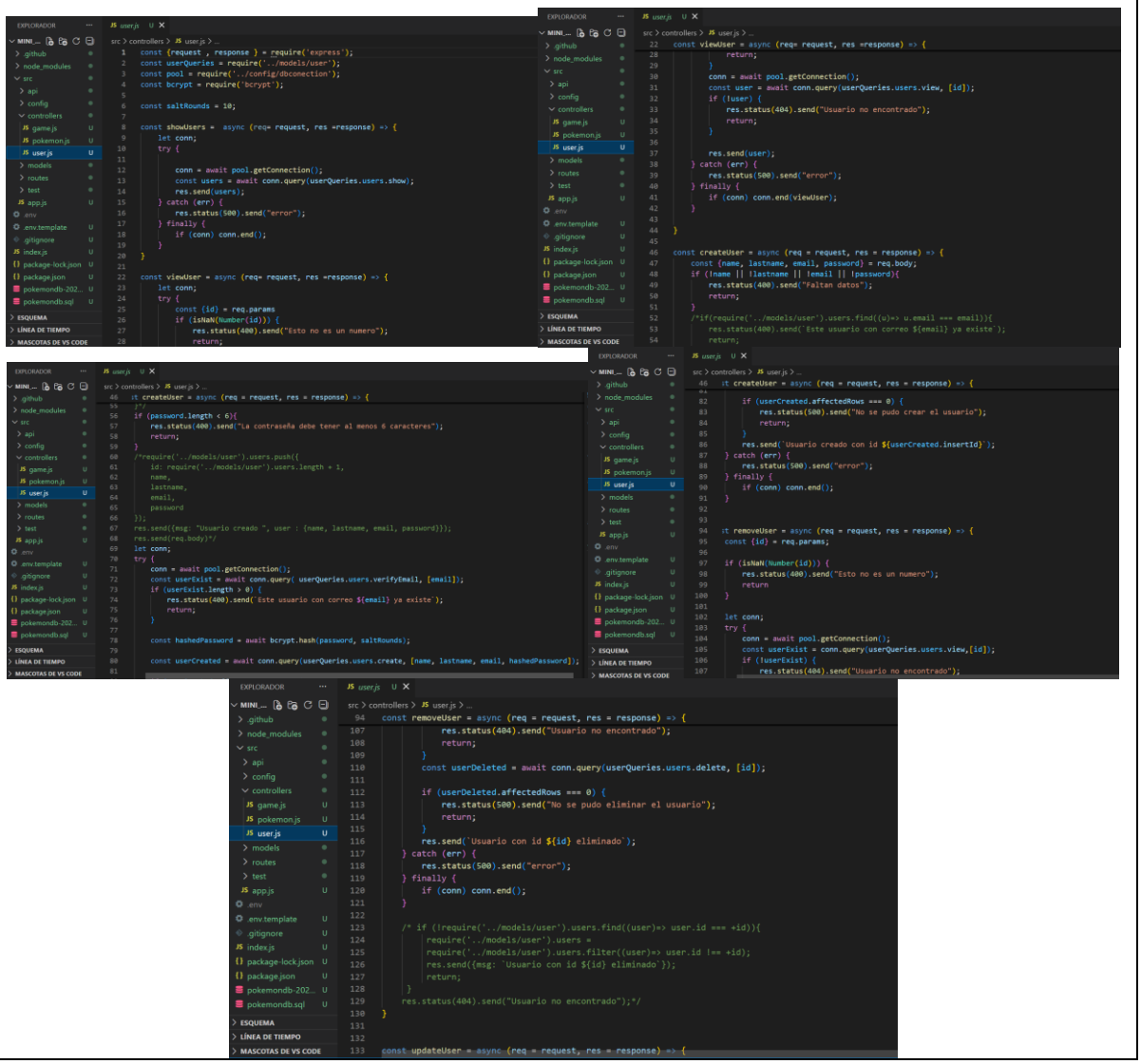
Una vez establecida la conexión, se verifica que el usuario exista en la base de datos. Si el usuario no se encuentra, la operación se detiene y se envía una respuesta de error, garantizando la integridad de la información. Si el usuario existe, el sistema consulta si ya cuenta con un registro de juego. En caso de que no exista dicho registro, se crea uno nuevo inicializando los valores de victorias y derrotas según corresponda. Si el registro ya existe, se actualiza incrementando el contador de victorias o derrotas. Durante todo el proceso se valida que las operaciones de inserción o actualización se realicen correctamente, verificando que se haya modificado al menos un registro. Además, el uso de bloques **try-catch-finally** permite manejar errores del servidor y asegurar que la conexión a la base de datos se cierre correctamente al finalizar la operación. Finalmente, las funciones se exportan para ser utilizadas en las rutas de la aplicación, formando parte de la lógica del **CRUD** del sistema y asegurando un manejo seguro y estructurado de las estadísticas de juego.

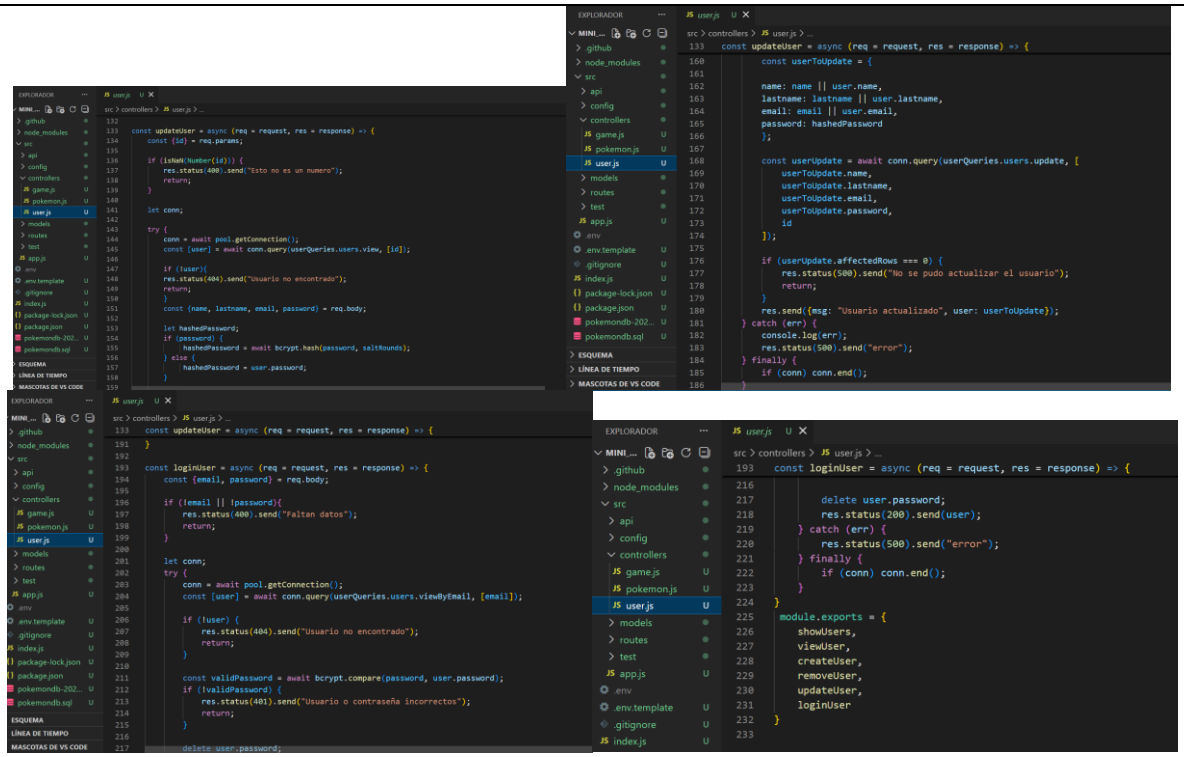


```
const { request, response } = require('express');
const pokeapi = require('../api/pokeapi');
const pool = require('../config/dbconection');
const { pokemonQuery } = require('../models/pokemon');

const pokemonSeeder = async (req = request, res = response) => {
    const pokemons = await pokeapi.getPokemons();

    let conn;

    try {
        conn = await pool.getConnection();
        await conn.query('SET FOREIGN_KEY_CHECKS = 0');
        await conn.query('TRUNCATE TABLE pokemons');
        await conn.query('SET FOREIGN_KEY_CHECKS = 1');

        pokemons.forEach(async (pokemon) => {
            await conn.query(pokemonQuery.add, [pokemon.name, pokemon.image]);
        });
        res.send("Pokemones agregados en la Base de Datos");
    } catch (err) {
        return res.status(500).send(err);
    } finally {
        if (conn) {
            conn.end();
        }
    }

}

const randomPokemon = async (req = request, res = response) => {
    let conn;
    try {
        conn = await pool.getConnection();
        const pokemons = await conn.query(pokemonQuery.random);
        if (pokemons.length === 0) {
            return res.status(500).send("No hay pokemones en la base de datos");
        }
        res.send(pokemons);

    } catch (err) {
        return res.status(500).send(err);
    } finally {
        if (conn) {
            conn.end();
        }
    }
}

const idPokemon = async (req = request, res = response) => {
    let conn;
    try {
        const { id } = req.params
        if (isNaN(Number(id))) {
            res.status(400).send("Esto no es un numero");
            return;
        }

        conn = await pool.getConnection();
        const pokeid = await conn.query(pokemonQuery.view, [id]);
        if (!pokeid) {
            res.status(404).send("Pokemon no encontrado");
            return;
        }
        res.send(pokeid);
    } catch (err) {
        return res.status(500).send(err);
    } finally {
        if (conn) {
            conn.end();
        }
    }
}

module.exports = {
    pokemonSeeder,
    randomPokemon,
    idPokemon
};
```

El archivo **pokemon.js** es un **controlador de Express** que gestiona las operaciones relacionadas con los Pokémon dentro del proyecto, conectando las rutas con la **PokeAPI** y la **base de datos MariaDB**. La función **pokemonSeeder** obtiene los Pokémon desde la PokeAPI, limpia la tabla pokemons y la vuelve a poblar con los datos actualizados. La función **randomPokemon** permite obtener un

Pokémon aleatorio almacenado en la base de datos, mientras que **idPokemon** busca un Pokémon específico mediante su identificador, validando previamente que sea un valor numérico. En todas las funciones se maneja adecuadamente la conexión a la base de datos y los errores, asegurando un funcionamiento correcto y ordenado del sistema.





El archivo **user.js** es un **controlador de Express** encargado de gestionar todas las operaciones relacionadas con los usuarios dentro del sistema, funcionando como parte central del **CRUD de usuarios** y del proceso de **autenticación**. En este archivo se utilizan consultas definidas en el modelo de usuario, un **pool de conexiones** a MariaDB y la librería **bcrypt** para manejar contraseñas de forma segura mediante cifrado.

La función **showUsers** obtiene y devuelve la lista completa de usuarios almacenados en la base de datos. La función **viewUser** permite consultar un usuario específico por su identificador, validando previamente que el ID sea numérico y verificando que el usuario exista. La función **createUser** se encarga de registrar nuevos usuarios, validando que los datos requeridos estén presentes, comprobando que el correo no esté previamente registrado y cifrando la contraseña antes de almacenarla en la base de datos, lo que refuerza la seguridad de la información.

Por otro lado, **removeUser** permite eliminar un usuario a partir de su ID, validando su existencia antes de realizar la eliminación. La función **updateUser** se utiliza para modificar la información de un usuario existente, permitiendo actualizar datos personales y la contraseña, la cual se vuelve a cifrar en caso de ser modificada. Finalmente, la función **loginUser** gestiona el proceso de inicio de sesión, verificando que el correo exista y comparando la contraseña ingresada con la almacenada

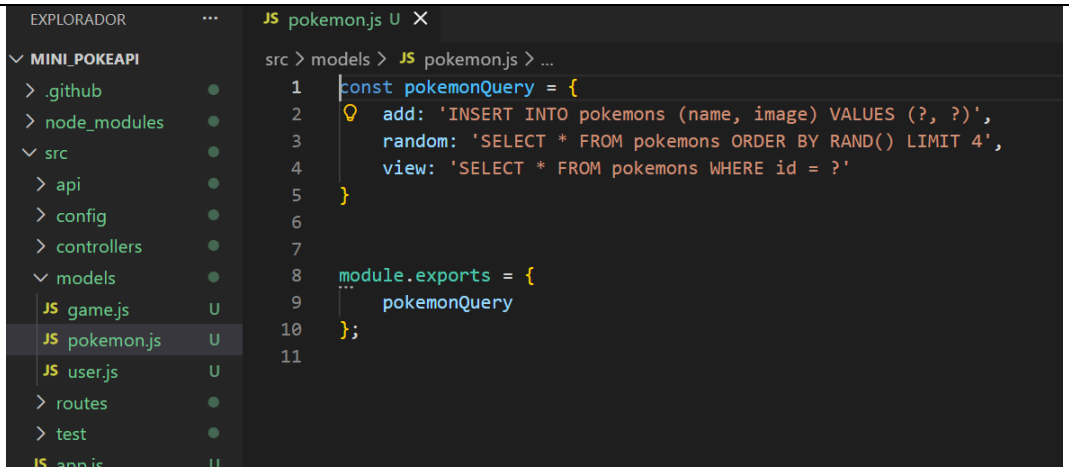
mediante bcrypt; si la autenticación es correcta, se devuelve la información del usuario sin incluir la contraseña.
En todas las funciones se emplea el uso de try-catch-finally para manejar errores y asegurar el cierre correcto de la conexión a la base de datos. En conjunto, este controlador garantiza un manejo seguro, validado y estructurado de los usuarios dentro de la aplicación.

Posteriormente la carpeta **models** contiene los siguientes archivos **game.js, pokemon.js y user.js:**

El archivo **game.js** pertenece a la carpeta models y define las consultas SQL necesarias para gestionar la información de los juegos en la base de datos. En él se agrupan las sentencias que permiten interactuar con la tabla games de forma ordenada y reutilizable.

La consulta **getGame** se utiliza para obtener el registro de un juego asociado a un usuario específico, buscando por su identificador (**user_id**). La consulta **addGame** permite crear un nuevo registro de juego para un usuario, inicializando los campos de victorias (**win**) y derrotas (**lose**). Por su parte, la consulta **updateGame** se encarga de actualizar los valores de victorias y derrotas de un usuario cuando se registra un nuevo resultado.

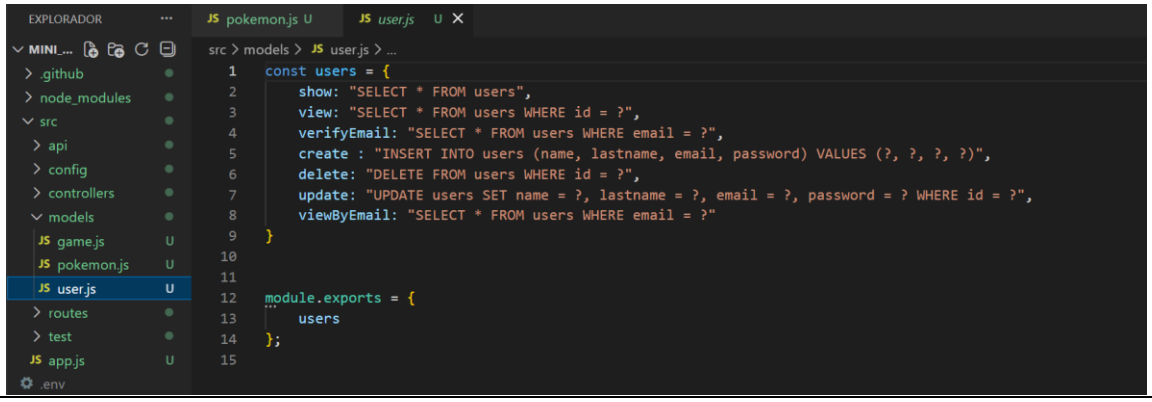
Finalmente, el objeto **gameQuery** se exporta para que pueda ser utilizado por los controladores, manteniendo la separación entre la lógica de negocio y el acceso a datos, lo que mejora la organización y el mantenimiento del proyecto.



El archivo **pokemon.js** ubicado en la carpeta models define las consultas SQL necesarias para interactuar con la tabla **pokemons** de la base de datos. En este archivo se agrupan las sentencias que permiten insertar y consultar información de los Pokémon de forma centralizada y reutilizable.

La consulta **add** se utiliza para insertar un nuevo Pokémon, almacenando su nombre y la URL de su imagen. La consulta **random** permite obtener cuatro Pokémon aleatorios de la base de datos, lo cual es útil para dinámicas de juego o selecciones al azar. Por su parte, la consulta **view** se encarga de buscar un Pokémon específico por su identificador, facilitando la consulta individual de registros.

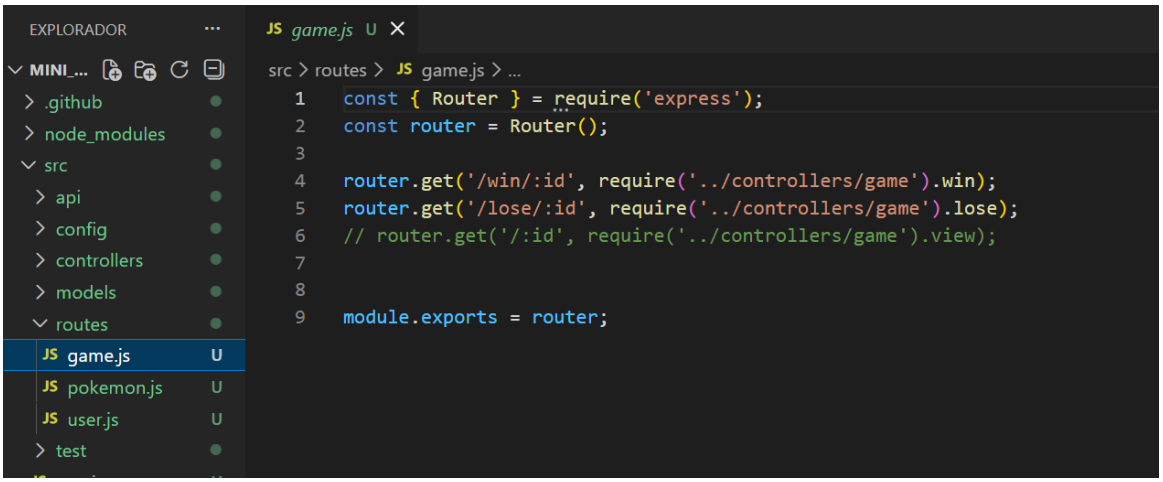
Finalmente, el objeto **pokemonQuery** se exporta para que pueda ser utilizado por los controladores, manteniendo la separación entre la lógica de acceso a datos y la lógica de negocio, lo que contribuye a un código más organizado, mantenible y claro.



El archivo **user.js** define las consultas SQL necesarias para la gestión de usuarios en la base de datos. En este archivo se centralizan las sentencias que permiten realizar operaciones de consulta, inserción, actualización y eliminación sobre la tabla users, facilitando su reutilización desde los controladores.

La consulta show obtiene todos los usuarios registrados, mientras que **view** permite consultar un usuario específico mediante su identificador. La consulta **verifyEmail** se utiliza para verificar si un correo electrónico ya se encuentra registrado, lo cual es clave para evitar duplicados. La consulta **create** permite insertar un nuevo usuario almacenando sus datos personales y la contraseña cifrada. Por su parte, **delete** elimina un usuario por su ID y **update** actualiza la información de un usuario existente. Finalmente, **viewByEmail** se emplea durante el proceso de autenticación para buscar un usuario a partir de su correo electrónico.

El objeto **users** se exporta para ser utilizado por los controladores, manteniendo una clara separación entre el acceso a datos y la lógica de negocio, lo que mejora la organización, seguridad y mantenimiento del proyecto.



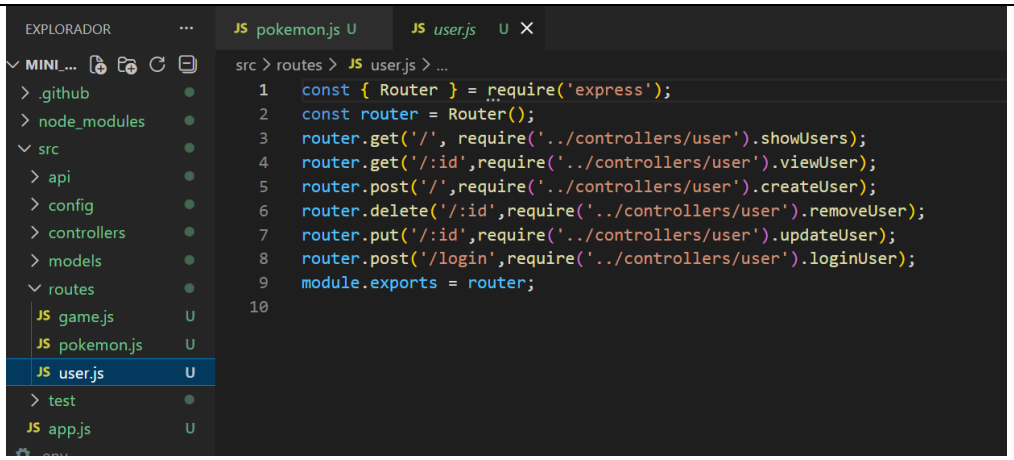
El archivo **game.js** ubicado en la carpeta routes define las rutas HTTP relacionadas con la lógica del juego dentro de la API. En este archivo se utiliza el enrutador de Express para asociar las solicitudes del cliente con las funciones correspondientes del controlador de juegos.

La ruta /win/:id recibe solicitudes de tipo GET y se encarga de registrar una victoria para el usuario cuyo identificador se envía como parámetro en la URL. De manera similar, la ruta /lose/:id registra una derrota para el usuario indicado. Ambas rutas delegan la lógica al controlador game, manteniendo separada la definición de rutas de la lógica de negocio. Finalmente, el enrutador se exporta para ser integrado en

la aplicación principal, permitiendo que estas rutas formen parte del flujo de la API y contribuyendo a una estructura ordenada y mantenible del proyecto.

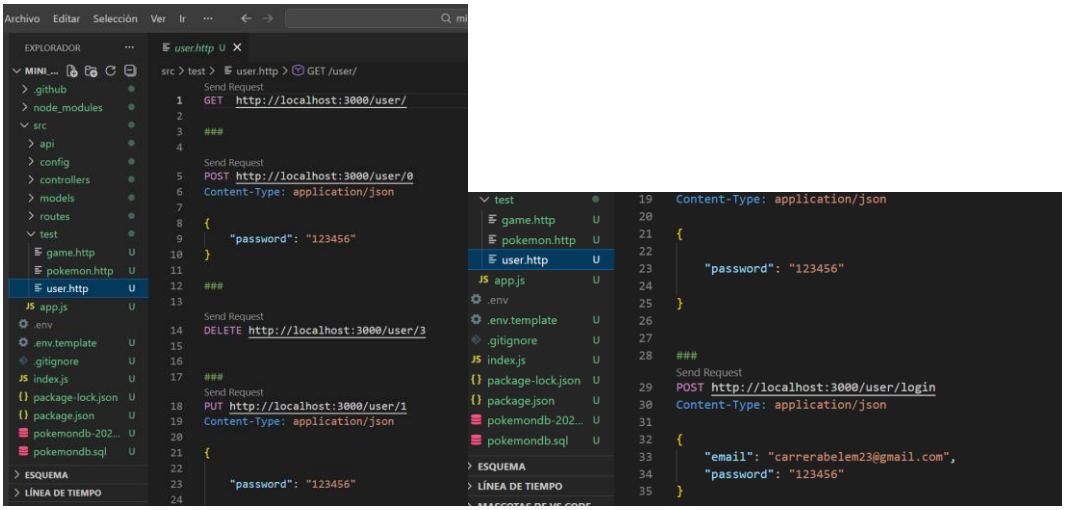
El archivo pokemon.js ubicado en la carpeta routes define las rutas HTTP relacionadas con los Pokémon dentro de la API. Utiliza el enrutador de Express para enlazar las solicitudes del cliente con las funciones correspondientes del controlador de Pokémon.

La ruta /seed ejecuta el controlador pokemonSeeder, el cual obtiene los Pokémon desde la PokeAPI y los inserta en la base de datos. Esta ruta se utiliza principalmente para inicializar o recargar la información de Pokémon en el sistema. Por otro lado, la ruta /random invoca el controlador randomPokemon, que devuelve Pokémon aleatorios almacenados en la base de datos, permitiendo funcionalidades como juegos o selecciones al azar. El enrutador se exporta para ser integrado en la aplicación principal, manteniendo una estructura clara donde las rutas únicamente definen los endpoints y delegan la lógica al controlador correspondiente.

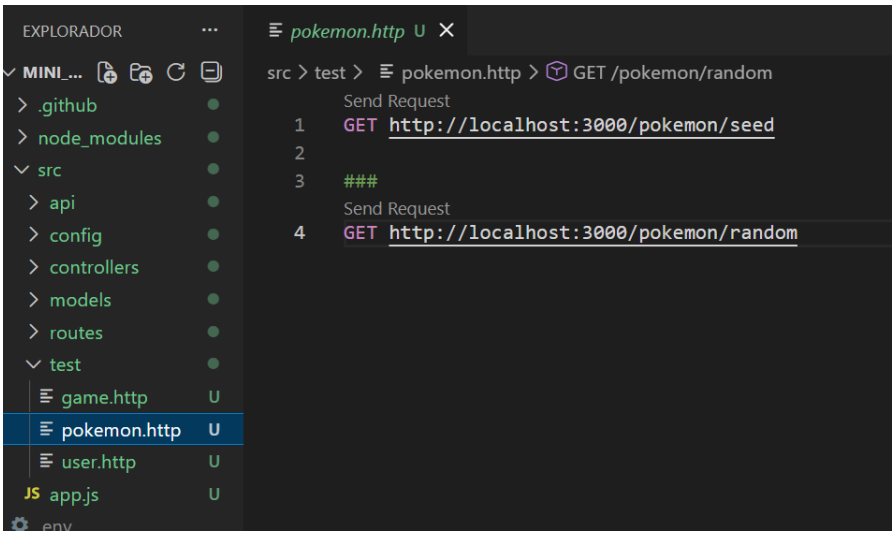


El archivo user.js ubicado en la carpeta routes define las rutas HTTP relacionadas con la gestión de usuarios dentro de la API. En este archivo se utiliza el enrutador de Express para asociar cada endpoint con la función correspondiente del controlador de usuarios.

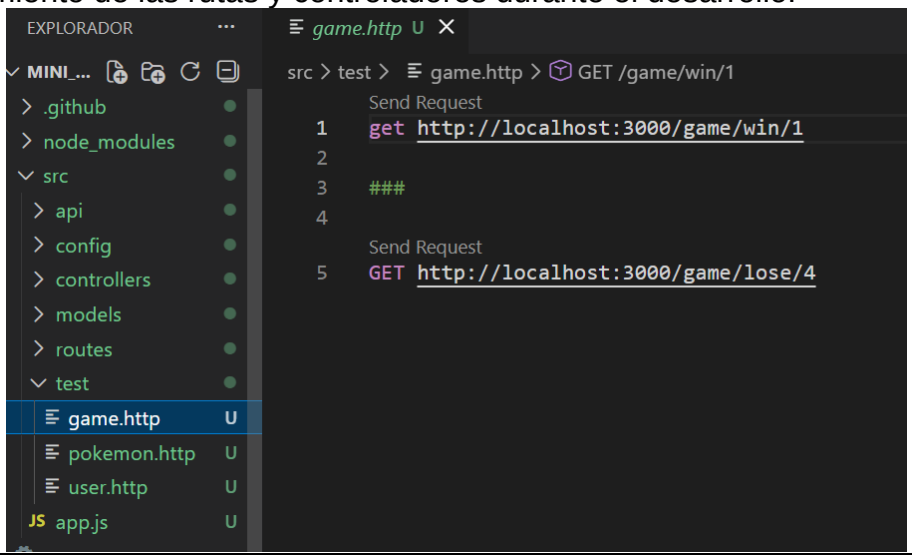
La ruta GET / permite obtener la lista completa de usuarios, mientras que GET /:id se utiliza para consultar un usuario específico mediante su identificador. La ruta POST / se encarga de crear un nuevo usuario, recibiendo los datos desde el cuerpo de la solicitud. Por su parte, DELETE /:id permite eliminar un usuario y PUT /:id actualizar la información de un usuario existente. Finalmente, la ruta POST /login gestiona el proceso de autenticación, validando las credenciales del usuario. Este archivo se exporta para integrarse en la aplicación principal, manteniendo una clara separación entre la definición de rutas y la lógica de negocio, lo que favorece una estructura organizada y mantenible del proyecto.



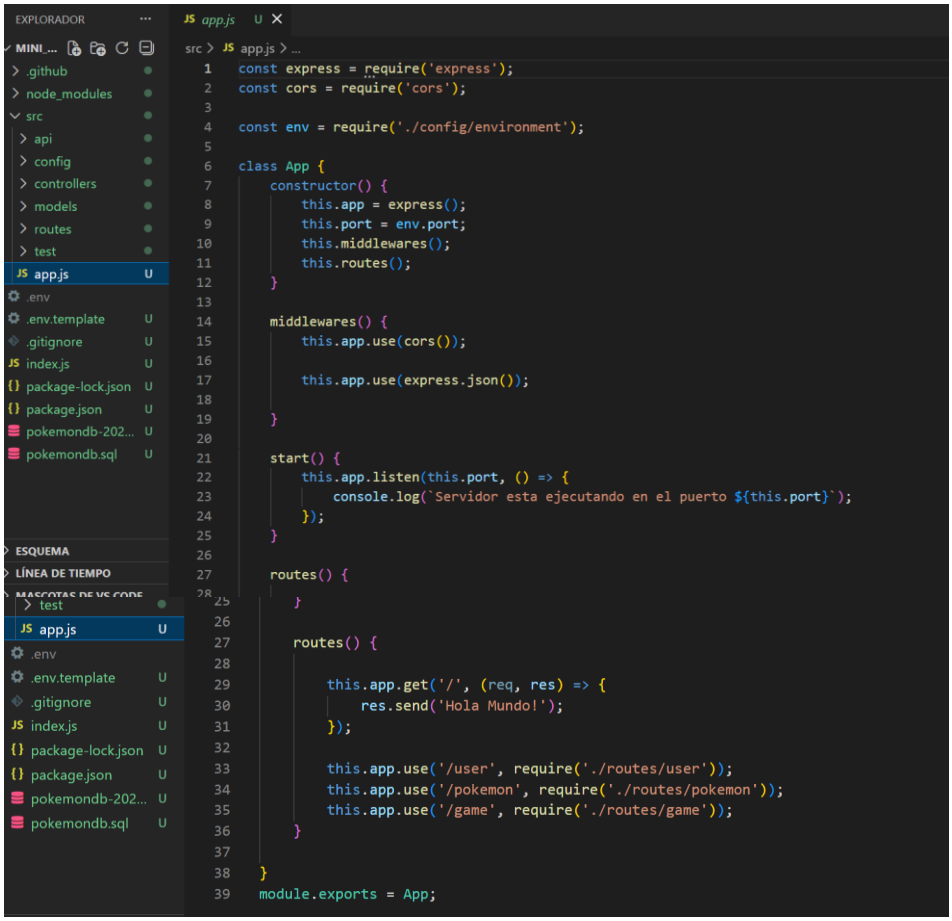
El archivo **user.http** se utiliza para probar manualmente las rutas del módulo de usuarios de la API. Contiene solicitudes HTTP que permiten listar usuarios, crear usuarios, actualizar datos, eliminar usuarios y realizar el inicio de sesión, enviando información en formato JSON. Este archivo facilita la validación del correcto funcionamiento del CRUD de usuarios y del proceso de autenticación durante el desarrollo.



El archivo **pokemon.http** se utiliza para probar las rutas del módulo de Pokémon de la API. Incluye solicitudes HTTP para cargar los Pokémon en la base de datos mediante la ruta **/pokemon/seed** y para obtener Pokémon aleatorios a través de la ruta /**pokemon/random**. Este archivo permite verificar rápidamente el correcto funcionamiento de las rutas y controladores durante el desarrollo.



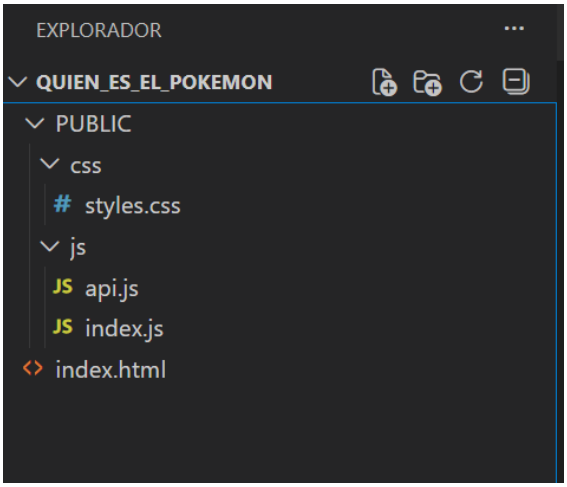
El archivo **game.http** se utiliza para probar las rutas del módulo de juegos de la API. Contiene solicitudes HTTP que permiten registrar victorias y derrotas de usuarios mediante peticiones GET, facilitando la verificación del correcto funcionamiento de las rutas y controladores durante el desarrollo.



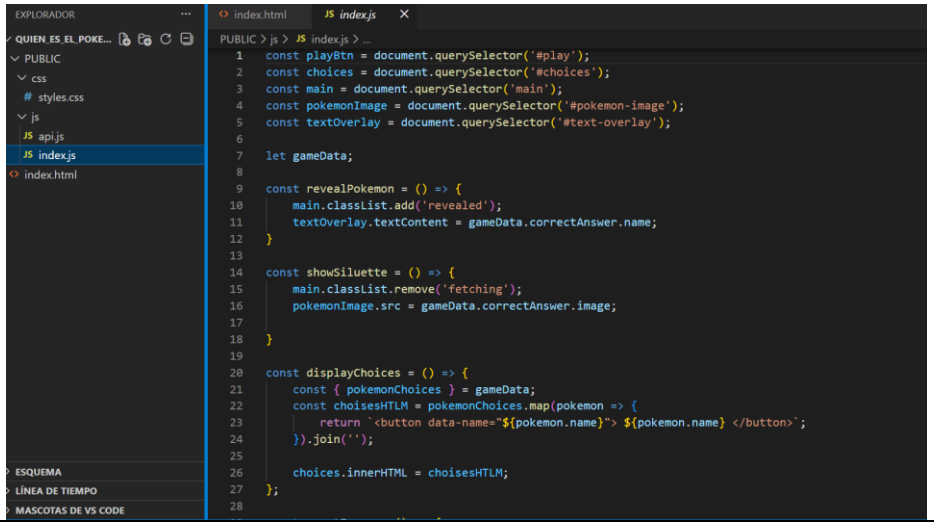
El archivo **app.js** es el núcleo de la aplicación. En él se configura el servidor con Express, se habilitan los middlewares como cors y el manejo de JSON, se define el puerto usando las variables de entorno y se registran las rutas principales (/user, /pokemon, /game). Además, incluye una ruta básica de prueba y el método que inicia el servidor, dejando la API lista para recibir peticiones.

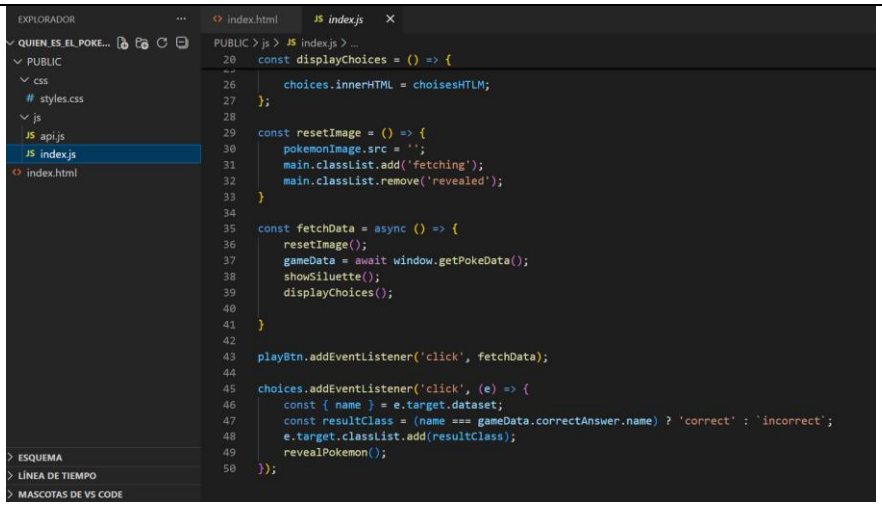
Parte 2: ¿Quién es el Pokémon?

Se desarrolló una aplicación cliente que consume los servicios de la *mini_pokeapi*. La aplicación realiza peticiones HTTP para obtener información de los Pokémon.

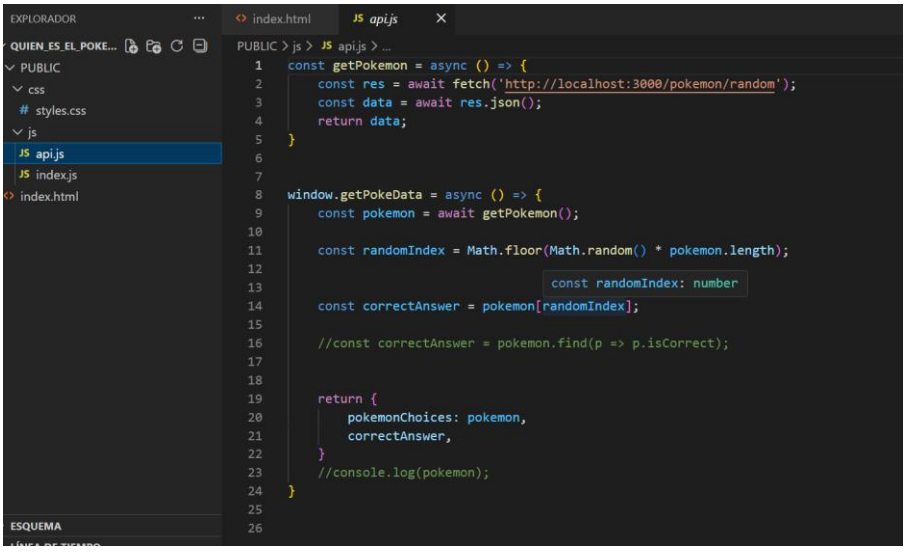


El proyecto **QUIÉN_ES_EL_POKEMON** es el frontend del juego. **index.html** muestra la interfaz, **styles.css** define el diseño, **api.js** se comunica con la API para obtener los Pokémon y **index.js** controla la lógica del juego y la interacción del usuario.

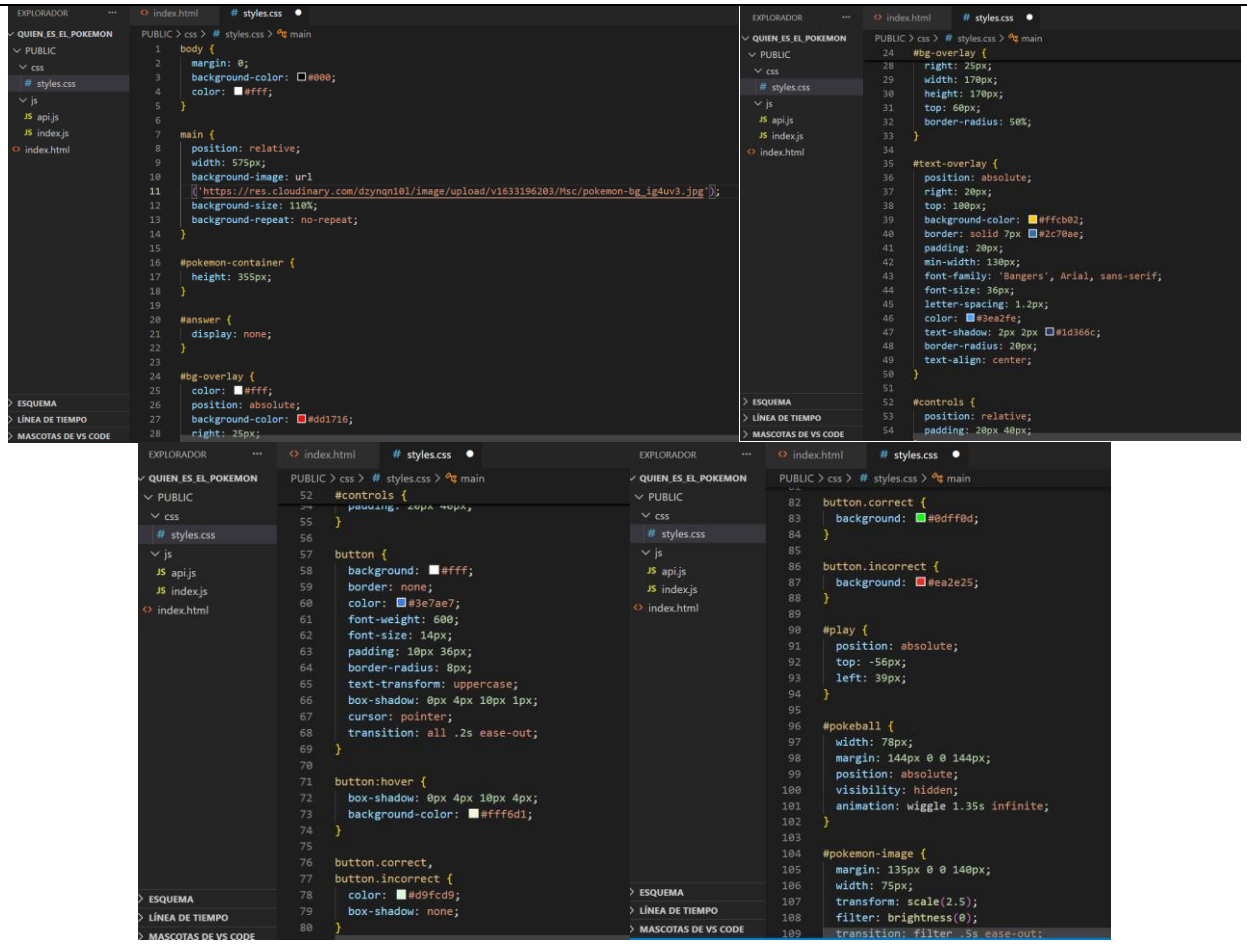




El archivo **index.js** controla la **lógica del juego "¿Quién es el Pokémon?"**. Se encarga de obtener los elementos del DOM, solicitar los datos del Pokémon a la API, mostrar la silueta y las opciones de respuesta, manejar los clics del usuario y revelar el Pokémon correcto indicando si la respuesta fue correcta o incorrecta.



El archivo **api.js** se encarga de la **comunicación con el backend**. Realiza una petición a la ruta /pokemon/random para obtener un conjunto de Pokémon, selecciona uno de forma aleatoria como **respuesta correcta** y devuelve tanto las **opciones de Pokémon** como el **Pokémon correcto**, datos que luego utiliza el archivo index.js para ejecutar el juego.



```css
body {
  margin: 0;
  background-color: #000;
  color: #fff;
}

main {
  position: relative;
  width: 575px;
  background-image: url
  ('https://res.cloudinary.com/dzynqn101/image/upload/v1633196203/Msc/pokemon-bg_ig4uv3.jpg');
  background-size: 110%;
  background-repeat: no-repeat;
}

#pokemon-container {
  height: 355px;
}

#answer {
  display: none;
}

#bg-overlay {
  color: #fff;
  position: absolute;
  background-color: #dd1716;
  right: 25px;
```

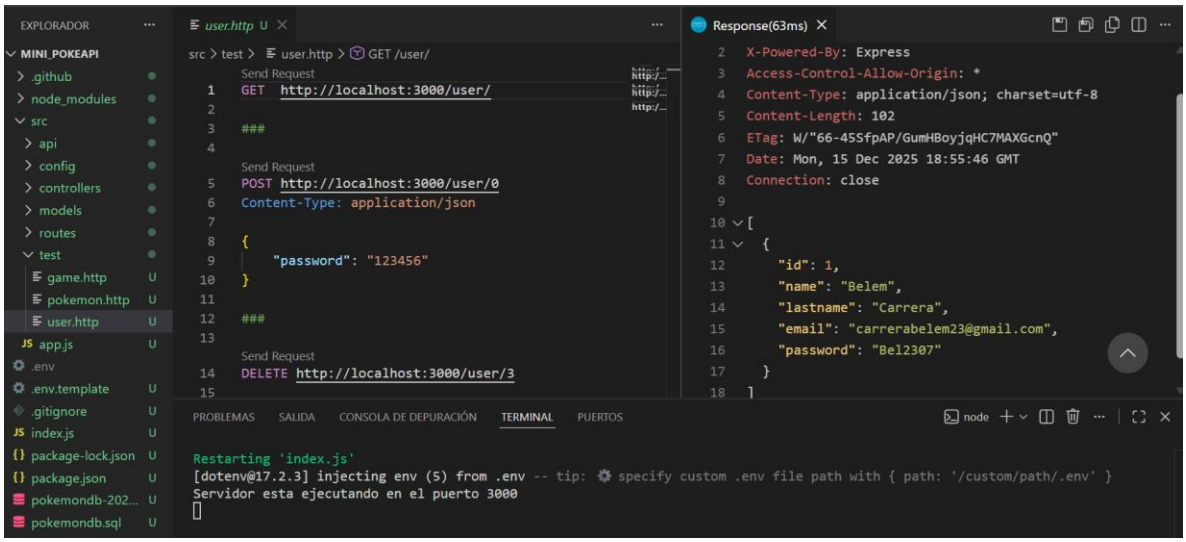
```css
#bg-overlay {
  right: 25px;
  width: 170px;
  height: 170px;
  top: 60px;
  border-radius: 50%;
}

#text-overlay {
  position: absolute;
  right: 20px;
  top: 100px;
  background-color: #ffcb02;
  border: solid 7px #2c70ae;
  padding: 20px;
  min-width: 130px;
  font-family: 'Bangers', Arial, sans-serif;
  font-size: 36px;
  letter-spacing: 1.2px;
  color: #3ea2fe;
  text-shadow: 2px 2px #1d366c;
  border-radius: 20px;
  text-align: center;
}

#controls {
  position: relative;
  padding: 20px 40px;
```

```css
#controls {
  padding: 20px 40px;
}

button {
  background: #fff;
  border: none;
  color: #3e7ae7;
  font-weight: 600;
  font-size: 14px;
  padding: 10px 36px;
  border-radius: 8px;
  text-transform: uppercase;
  box-shadow: 0px 4px 10px 1px;
  cursor: pointer;
  transition: all .2s ease-out;
}

button:hover {
  box-shadow: 0px 4px 10px 4px;
  background-color: #fff6d1;
}

button.correct,
button.incorrect {
  color: #d9fcd9;
  box-shadow: none;
}
```

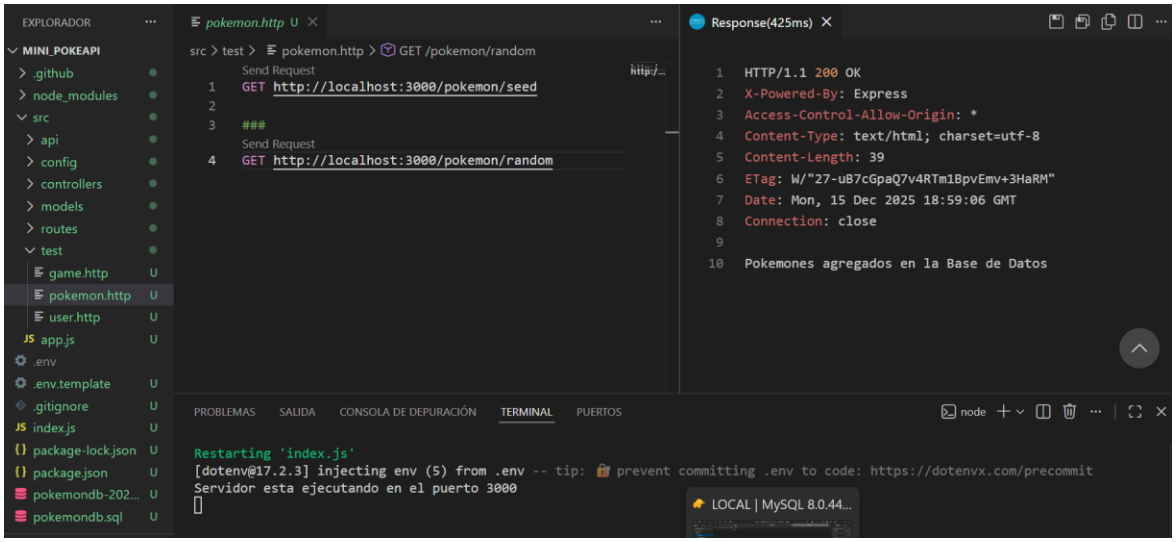
```css
button.correct {
  background: #0dff0d;
}

button.incorrect {
  background: #ea2e25;
}

#play {
  position: absolute;
  top: -56px;
  left: 39px;
}

#pokeball {
  width: 78px;
  margin: 144px 0 0 144px;
  position: absolute;
  visibility: hidden;
  animation: wiggle 1.35s infinite;
}

#pokemon-image {
  margin: 135px 0 0 140px;
  width: 75px;
  transform: scale(2.5);
  filter: brightness(0);
  transition: filter .5s ease-out;
```

El archivo **styles.css** define la **apariencia visual del juego "¿Quién es el Pokémon?"**. Establece un fondo oscuro general, el diseño del contenedor principal con una imagen de fondo, y controla el tamaño y posición de los elementos del juego. Además, da estilo a los botones (colores, sombras, animaciones y estados correcto/incorrecto), oculta y revela elementos según el estado del juego, y aplica efectos visuales como la silueta del Pokémon (oscurecimiento con brightness(0)) y animaciones decorativas. En conjunto, este archivo se encarga de que la interfaz sea clara, atractiva e interactiva.

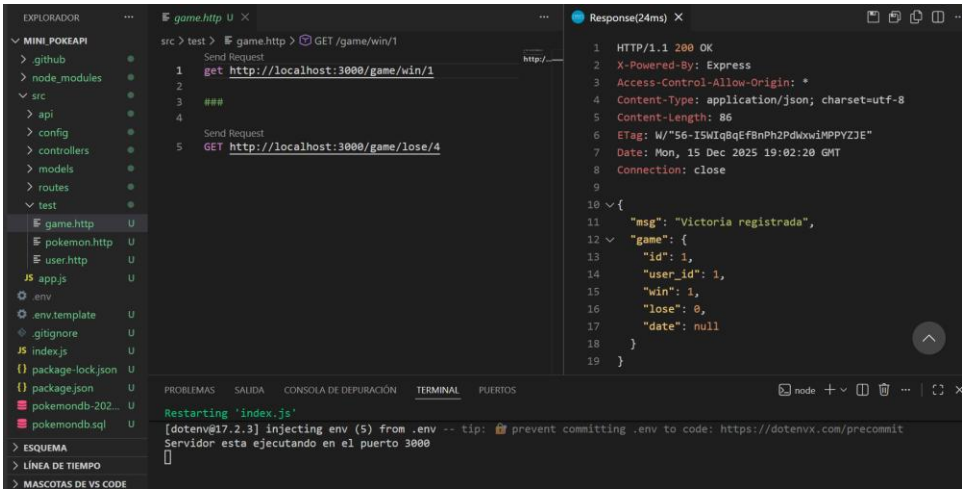
6. Resultados y conclusiones



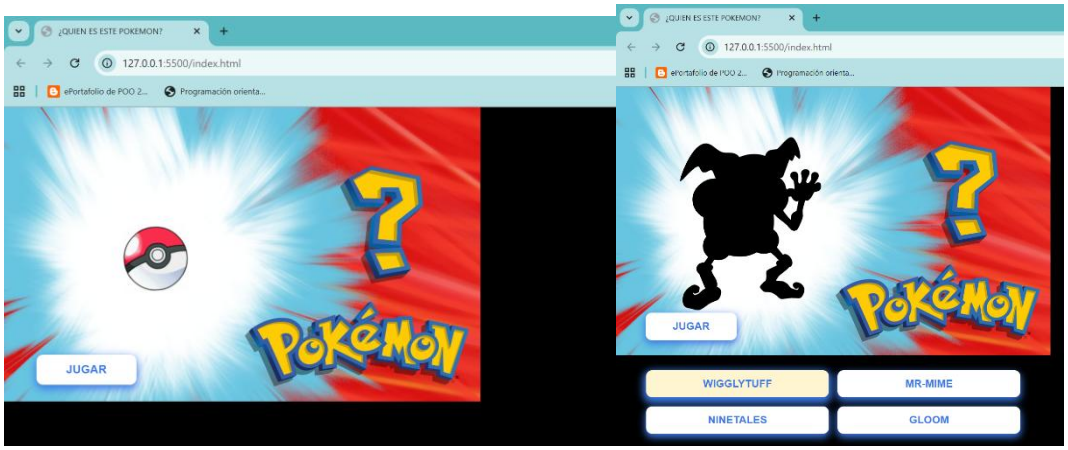
La **prueba del endpoint /user** usando un archivo **user.http**. Se realiza una petición **GET a http://localhost:3000/user/**, y el servidor responde correctamente con un **JSON** que contiene la lista de usuarios registrados. La respuesta confirma que el **servidor Express está activo en el puerto 3000**, que CORS está habilitado y que la API devuelve los datos del usuario (id, nombre, apellido, correo y contraseña) desde la base de datos, validando así el funcionamiento correcto del módulo de usuarios y su conexión con el backend.

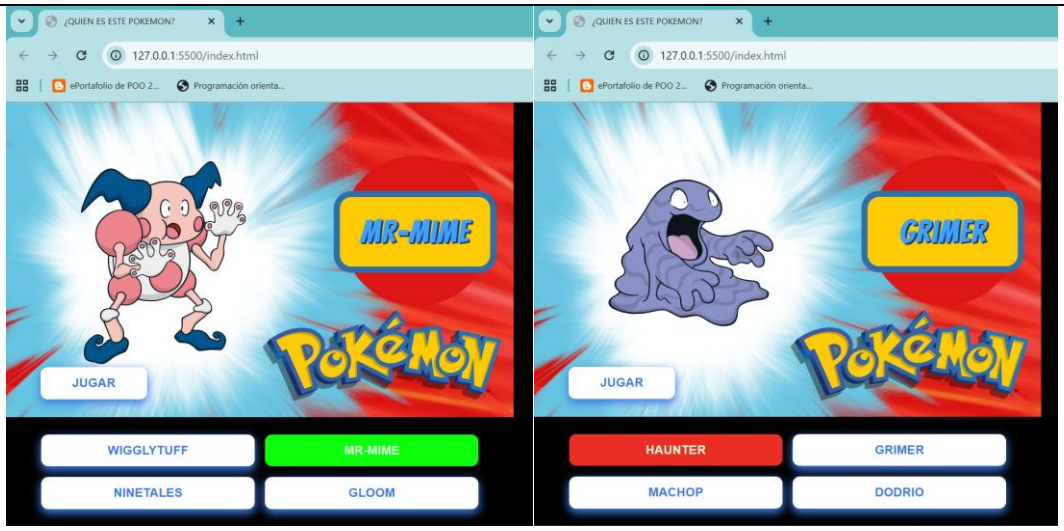


la **prueba del endpoint /pokemon** desde el archivo **pokemon.http**. Primero se ejecuta la petición **GET /pokemon/seed**, la cual responde con **200 OK** y el mensaje *"Pokemones agregados en la Base de Datos"*, confirmando que los Pokémon fueron cargados correctamente en la base de datos. Posteriormente, la ruta **GET /pokemon/random** queda disponible para obtener Pokémon de forma aleatoria, validando que el módulo Pokémon y su conexión con la base de datos funcionan correctamente.



El endpoint /game usando el archivo **game.http**. Al ejecutar la petición **GET /game/win/1**, el servidor responde con **200 OK** y confirma que la **victoria fue registrada correctamente** para el usuario con ID 1, incrementando el contador de **wins** en la base de datos. Esto valida que el módulo de juego funciona correctamente, actualiza las estadísticas del usuario y mantiene la conexión adecuada entre Express y la base de datos.





El **funcionamiento final del juego "¿Quién es este Pokémon?"**. Al iniciar, el usuario presiona **JUGAR** y se muestra la **silueta de un Pokémon** junto con varias **opciones de respuesta**. Al seleccionar una opción, el juego **revela la imagen real del Pokémon**, muestra su **nombre** y resalta la respuesta en **verde si es correcta** o en **rojo si es incorrecta**. Luego, el jugador puede volver a jugar para intentar con otro Pokémon. En conjunto, se evidencia una interacción completa entre el **frontend (HTML y CSS)** y el **backend**, ofreciendo una experiencia visual clara y dinámica.

Conclusión:

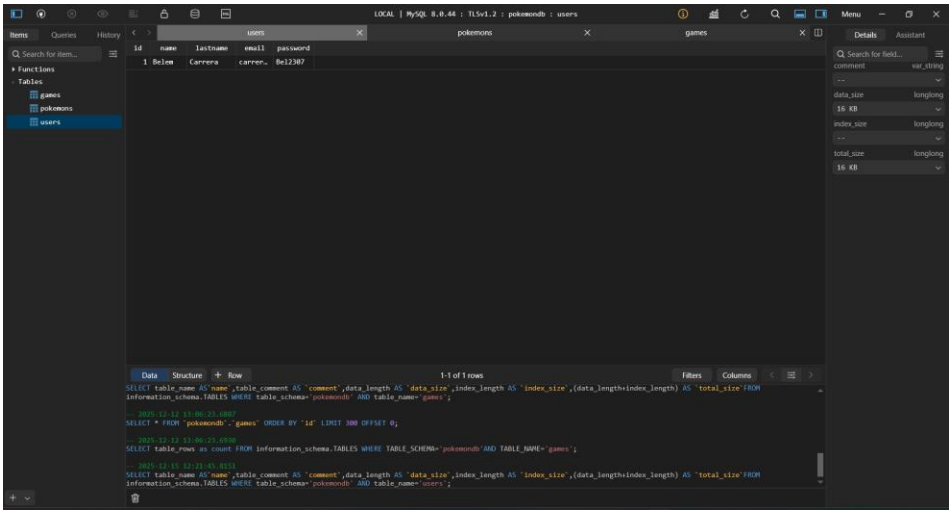
En la presente práctica se logró desarrollar e integrar exitosamente un sistema basado en la arquitectura cliente–servidor, compuesto por una API REST (mini_pokeapi) y una aplicación cliente denominada ¿Quién es el Pokémon?, cumpliendo con los objetivos planteados. A través del uso de Node.js y Express, se implementaron correctamente los endpoints necesarios para realizar operaciones CRUD, permitiendo la gestión de usuarios, Pokémon y estadísticas de juego de manera estructurada y segura.

La correcta conexión con la base de datos MariaDB, mediante el uso de pools de conexión y variables de entorno, permitió garantizar un acceso eficiente a la información y una adecuada separación entre la lógica de negocio y el acceso a datos. Asimismo, la implementación de mecanismos de validación y cifrado de contraseñas con bcrypt reforzó la seguridad del sistema.

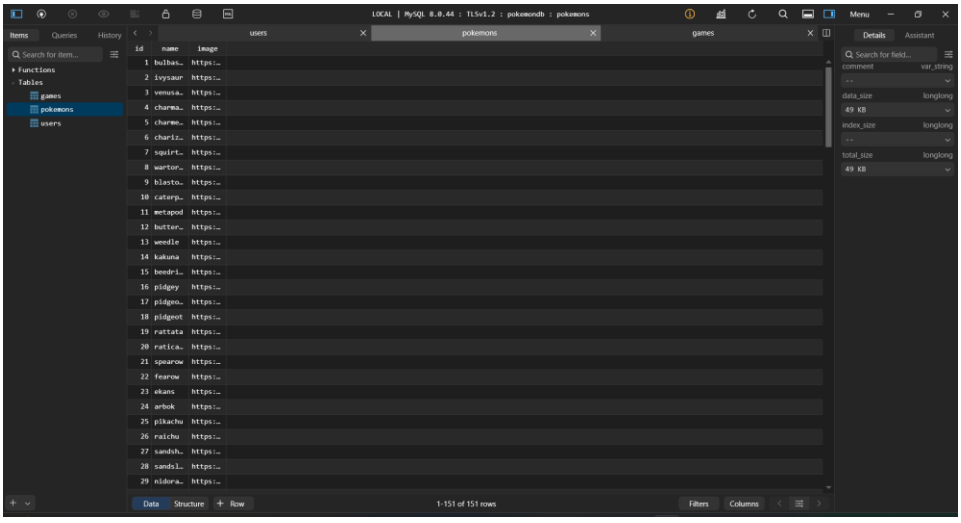
Por otro lado, el desarrollo de la aplicación cliente permitió comprobar de forma práctica el consumo de servicios web REST, logrando una interacción dinámica entre el frontend y el backend. El juego "¿Quién es el Pokémon?" evidenció el flujo completo de datos, desde la solicitud HTTP hasta la presentación visual de los resultados, demostrando el correcto

funcionamiento de la comunicación entre ambas capas, esta práctica fortaleció las competencias en el desarrollo de aplicaciones cliente–servidor, el consumo de APIs REST y la implementación de operaciones CRUD, brindando una experiencia integral que combina teoría y práctica, y sentando bases sólidas para el desarrollo de aplicaciones web y móviles más complejas en entornos reales

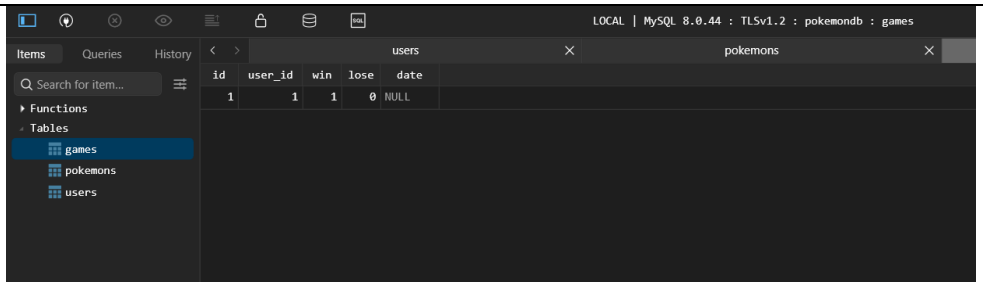
7. Anexos



Anexo 1: resultados de los datos de la base de datos en la tabla user



Anexo 2: resultados de los datos de la base de datos en la tabla pokemons



Anexo 3: resultados de los datos de la base de datos en la tabla games

Fecha de realización:

Formuló:	**Realizó**
Julio Aguilar Carmona	**María Belen Carrera Velasco 22350634**
Catedrático(a)	Nombre, No. Control y firma del participante