Relación 12 - Desarrollo en servidor con Node

Gestión del Entorno y Configuración

Ejercicio 1. Un equipo de desarrollo te entrega un proyecto que requiere específicamente la versión 18 de Node.js. Describe los comandos necesarios para instalar dicha versión, activarla para trabajar y verificar que efectivamente es la versión activa en el terminal.

Ejercicio 2. Explica los pasos para iniciar un proyecto Node.js desde cero. ¿Qué comando crea el archivo `package.json`? ¿Cómo instalarías la librería `express` para que se guarde en las dependencias del proyecto? ¿Y si quisieras instalar `nodemon` solo como dependencia de desarrollo?

Ejercicio 3. ¿Qué propiedad debes añadir al `package.json` para poder utilizar la sintaxis `import` en lugar de `require`? Explica una diferencia importante respecto a las extensiones de archivo al importar módulos locales bajo el estándar ES Modules.

Desarrollo en servidor Node.js

Ejercicio 4. Escribe el código necesario para un archivo de servidor que:

- Importe el módulo `express` y `cors`.
- Configure los middlewares correspondientes para recibir peticiones externas al servidor y para recibir cuerpos de mensaje como JSON.
- Levante el servidor en el puerto 3000 mostrando el mensaje "Servidor corriendo" por consola.

Ejercicio 5. Siguiendo el ejemplo de la farmacia, crea una ruta `GET` que reciba una `seccion` y un `idProducto` (ej: `/farmacia/dermocosmetica/p-805`). El servidor debe devolver un objeto JSON con ambos valores. Utiliza desestructuración para extraer los parámetros de `req.params`.

Ejercicio 6. Imagina que estás desarrollando una API para una biblioteca. Crea una ruta `GET` que reciba el nombre de una **sección** y el **ID** de un libro (ej: `/biblioteca/ficcion/libro/452`).

- El servidor debe imprimir en la consola: "Buscando libro 452 en la sección ficcion".
- Debe responder al cliente con un JSON que confirme la recepción de ambos datos.

Ejercicio 7. Diseña una ruta `GET /vuelos` para una agencia de viajes. El servidor debe ser capaz de leer tres parámetros opcionales desde la URL: `origen`, `destino` y `clase` (ej: `/vuelos?origen=sevilla&destino=roma&clase=business`).

- El servidor debe extraer estos valores usando `req.query`.

- Si falta el parámetro `clase`, el servidor debe asignar por defecto el valor "turista".
- Responde con un mensaje: "Buscando vuelos de [origen] a [destino] en clase [clase]".

Ejercicio 8. Crea una ruta `POST /login`. El servidor debe recibir en el cuerpo de la petición (`req.body`) un objeto con las propiedades `usuario` y `password`.

- Si el `usuario` es "admin" y la `password` es "1234", responde con un status **200** y el mensaje "Acceso concedido".
- En cualquier otro caso, responde con un status **401** (Unauthorized) y el mensaje "Credenciales incorrectas".

Ejercicio 9. Crea una ruta `GET /comparar` que reciba una lista de IDs de productos mediante la misma clave en la URL (ej: `/comparar?id=10&id=20&id=30`).

- Express convertirá automáticamente estas claves repetidas en un Array.
- El servidor debe responder indicando cuántos elementos se van a comparar (la longitud del array).

Ejercicio 10. Crea una ruta `PUT /usuario/:id/configuracion`. Esta ruta debe:

- Obtener el `id` del usuario mediante **Params**.
- Obtener el nuevo `idioma` mediante un **Query Parameter**.
- Obtener el `tema` (oscuro o claro) y las `notificaciones` (true/false) mediante el **Body**.
- Devuelve un JSON que unifique toda la información recibida.

Ejercicio 11. [*Método DELETE: Cancelación de citas*] Crea una ruta `DELETE /citas/:idCita`.

- El servidor debe capturar el ID de la cita mediante **Params**.
- Debe responder con un JSON que diga: "La cita con ID [idCita] ha sido cancelada correctamente".

Interacción con Bases de Datos No Relacionales (MongoDB)

Ejercicio 12. [*Búsqueda y Proyección (Ciudades)*]
Utilizando el dataset `mongo_cities1000.json`, implementa la ruta `GET /ciudad/info/:nombre`.

- Busca el documento cuyo campo `name` coincida con el parámetro.
- Utiliza una **proyección** para que MongoDB devuelva únicamente los campos `population` y `timezone`, excluyendo el `_id`.

Ejercicio 13. [*Filtros Avanzados (Población)*]
Crea una ruta `GET /ciudades/grandes`.

- Esta ruta debe recibir por **Query String** un valor llamado `min`.
- Debe devolver todas las ciudades cuya población sea mayor o igual (`$gte`) al valor de `min`.
- *Nota:* Recuerda convertir el parámetro a número con `parseInt()` antes de enviarlo a la consulta de Mongo.

Ejercicio 14. [*Geolocalización Básica*]
Crea una ruta `GET /ciudad/posicion/:nombre`.

- Al buscar la ciudad, el servidor debe responder con un mensaje de texto plano que diga: "La ciudad [nombre] se encuentra en las coordenadas: Latitud [lat], Longitud [long]".
- Extrae estos datos del objeto anidado `location` que aparece en los documentos JSON.

Ejercicio 15. [*Borrado NoSQL: Eliminar ciudades*] Crea una ruta `DELETE /ciudad/:nombre` para el dataset de ciudades.

- Utiliza el método `deleteOne()` de MongoDB para eliminar la ciudad cuyo nombre coincida con el parámetro.
- Responde con el objeto de confirmación que devuelve el driver de Mongo (`deletedCount`).

Interacción con Bases de Datos Relacionales (MySQL)

Ejercicio 16. [*Consultas de Selección con Parámetros*]
Utilizando la tabla `COCHE` del archivo `ejemploCoches.sql`, crea una ruta `GET /coches/:modelo`.

- El servidor debe realizar una consulta SQL para devolver todos los coches cuyo modelo coincida con el parámetro recibido (ej: `'glx'`).
- **Reto:** Asegúrate de usar consultas preparadas (con el símbolo `?`) para evitar la inyección SQL.

Ejercicio 17. [*Inserción de Datos (Ventas)*]
Implementa una ruta `POST /ventas/nueva`. El cuerpo de la petición contendrá los datos necesarios para registrar una venta en la tabla `VENTA` (`cifc, cifcl, codcoche, color`).

- Realiza el `INSERT` correspondiente en MySQL.
- Si la inserción es correcta, devuelve un código **201** y un mensaje confirmando la operación.

Ejercicio 18. [*Actualización de Clientes*]
Crea una ruta `PUT /clientes/:cifcl`.

- Debe permitir actualizar la `ciudad` de un cliente específico identificado por su `cifcl`.
- La nueva ciudad se recibirá a través del **body**.

Ejercicio 19. [*Borrado SQL: Retirada de vehículos*] Implementa una ruta `DELETE /coches/:codcoche`.

- Debe realizar una consulta `DELETE` en la tabla `COCHE` de la base de datos de coches.
- Utiliza consultas preparadas para evitar inyecciones.
- Responde informando si el coche se ha eliminado (puedes usar `result.affectedRows` para confirmar si existía).

Full-Stack: Express, MongoDB y Consumo de APIs

Ejercicio 20. [*Lectura General y Thunder Client*]
Crea una ruta `GET /api/ciudades` que devuelva todos los documentos de la colección.

- **Servidor:** Usa `find().toArray()` para obtener los datos de MongoDB.
- **Cliente:** Realiza la petición desde **Thunder Client** y verifica que el cuerpo de la respuesta sea un array de objetos JSON.

Ejercicio 21. [*Búsqueda por ID y Params*]
Implementa la ruta `GET /api/ciudad/:id`.

- **Servidor:** Usa `new ObjectId` pasándole el id, en el caso de que los ids sean generados por MongoDB, para filtrar el documento por su identificador único de MongoDB.
- **Cliente:** Prueba la ruta en Thunder Client usando un ID válido extraído del ejercicio anterior.

Ejercicio 22. [*Inserción de Datos y Fetch*]
Crea una ruta `POST /api/ciudades`.

- **Servidor:** Extrae los datos del cuerpo de la petición e insértalos en la colección usando `insertOne()`.
- **Fetch:** Escribe el código de cliente que envíe una nueva ciudad. No olvides el `JSON.stringify` y la cabecera `Content-Type`.

Ejercicio 23. [*Filtros de Rango y Query Strings*]
Diseña una ruta `GET /api/busqueda/poblacion`.

- **Servidor:** Debe leer de la **Query String** los parámetros `min` y `max`. Usa los operadores `$gte` y `$lte` simultáneamente.

- **Cliente:** Prueba en el navegador la URL: `/api/busqueda/poblacion?min=1000&max=5000`.

Ejercicio 24. [*Actualización Completa con PUT*]
Implementa una ruta `PUT /api/ciudad/:id`.

- **Servidor:** Sustituye todos los datos de una ciudad identificada por su ID usando el método `updateOne` para todos los campos recibidos en el cuerpo de la petición.
- **Cliente:** Envía desde Thunder Client el objeto completo actualizado.

Ejercicio 25. [*Borrado y Confirmación de Estado*]
Crea una ruta `DELETE /api/ciudad/:id`.

- **Servidor:** Tras ejecutar `deleteOne()`, verifica el campo `deletedCount`. Si es 0, devuelve un status **404** (Not Found).
- **Cliente:** Realiza la petición y observa en Thunder Client el código de estado devuelto por distintas llamadas tanto a ciudades existentes como inexistentes.

Ejercicio 26. [*Proyecciones: Menos es Más*]
Modifica tu ruta de listado para que sea `GET /api/ciudades/nombres`.

- **Servidor:** La consulta a MongoDB debe incluir una **proyección** para devolver solo el nombre y el país, omitiendo el campo `_id`.
- **Cliente:** Muestra los datos en una lista `<ul>` sencilla en tu HTML usando un `forEach`.

Ejercicio 27. [*Ordenación Dinámica*]
Crea una ruta `GET /api/ciudades/ordenadas/:sentido`.

- **Servidor:** El parámetro `sentido` puede ser 'asc' o 'desc'. Usa una condición para pasar `1` o `-1` al método `sort({ name: ... })`.
- **Cliente:** Comprueba que al cambiar el parámetro en la URL, el orden de la lista cambia automáticamente.

Ejercicio 28. [*Contabilización de Documentos*]
Implementa la ruta `GET /api/ciudades/total`.

- **Servidor:** Utiliza el método `countDocuments()` para obtener el número total de registros en la colección.
- **Cliente:** Muestra este número en un encabezado `<h1>` al cargar la página.

Ejercicio 29. [*Manejo de Errores con Try/Catch*]
Refactoriza la ruta de inserción (POST) para manejar posibles errores de conexión.

- **Servidor:** Si la base de datos devuelve un error, el servidor debe responder un estado 500, y un mensaje indicando que no se pudo guardar la ciudad.
- **Cliente:** Intenta enviar tanto un JSON correcto como luego un JSON mal formado y comprueba cómo el `catch` del servidor evita que el proceso se detenga.