HAZLO TÚ MISMO – 1: DISEÑO DE UNA API REST

Diseña una API REST para la gestión de citas médicas en un hospital que ofrece diversas especialidades. La API será utilizada por distintos tipos de clientes (página web del hospital, aplicaciones internas, sistemas administrativos, etc.) y deberá ser **coherente, bien estructurada, escalable y fácil de consumir** desde un cliente JavaScript.

El hospital cuenta con pacientes registrados, médicos asociados a especialidades y agendas de atención, y permite la reserva, modificación y cancelación de citas médicas.

Se debe **diseñar la API completa**, aplicando las buenas prácticas de diseño REST estudiadas en clase.

NOTA: No se debe programar la API, solo **diseñar su estructura conceptual y documentarla de forma técnica**.

OBJETIVO DEL EJERCICIO

Diseñar una API que permita:

- Consultar especialidades médicas.
- Consultar médicos y su disponibilidad de agenda.
- Registrar y consultar pacientes.
- Gestionar el ciclo de vida de una cita médica: creación, consulta, modificación y cancelación.
- Filtrar, ordenar y paginar colecciones de datos relevantes.
- Establecer un sistema de versionado adecuado.
- Documentar la API de forma comprensible y profesional.

ALCANCE DEL DISEÑO

La especificación debe abarcar, como mínimo, los siguientes aspectos:

IDENTIFICACIÓN DE RECURSOS

Define claramente qué entidades deben considerarse recursos dentro de la API. Cada recurso debe estar nombrado de forma consistente, en línea con las recomendaciones de diseño vistas en clase.

ESTRUCTURA DE LOS ENDPOINTS

Diseña los *endpoints* necesarios para cubrir las funcionalidades del sistema, siguiendo criterios REST:

- operaciones típicas asociadas a los recursos;
- rutas para acceder a colecciones y a elementos individuales;
- rutas para acceder a información relacionada (por ejemplo, citas asociadas a un paciente o a un médico).

Debes justificar el diseño desde la perspectiva de la arquitectura REST estudiada.

PAGINACIÓN, FILTROS Y ORDENACIÓN

La API debe soportar mecanismos que permitan:

- limitar el volumen de datos,
- seleccionar subconjuntos específicos,

- ordenar colecciones según criterios relevantes.

Define los parámetros necesarios y la lógica conceptual de uso.

MANEJO DE ERRORES Y CÓDIGOS DE ESTADO

Identifica los posibles escenarios de error y el código de estado HTTP que resultaría apropiado para cada situación. Indica también qué estructura tendría la respuesta de error para facilitar su consumo por aplicaciones cliente.

ESTRATEGIA DE VERSIONADO

Define cómo se versionará la API y explica por qué dicha estrategia es adecuada al contexto. Describe cómo debería evolucionar la versión en caso de cambios futuros.

DOCUMENTACIÓN TÉCNICA INICIAL

Presenta la API mediante una **descripción técnica clara**. No se exige una especificación completa, pero sí una estructura clara que permita entender los recursos, los parámetros, las respuestas y los ejemplos de uso.

RESULTADO

Para una especificación completa, se debe generar un documento técnico en el que conste:

1. La lista de recursos identificados.
2. El conjunto completo de *endpoints* diseñados.
3. La descripción de mecanismos de paginación, filtros y ordenación.
4. La propuesta de manejo de errores y códigos HTTP.
5. La estrategia de versionado seleccionada y su justificación.
6. La documentación técnica a nivel introductorio.

La documentación resultante debe estructurarse y escribirse con **claridad**, **coherencia interna** y **precisión terminológica**.

HAZLO TÚ MISMO – 2: CONSUMO DE UNA API REST PARA LA GESTIÓN DE PACIENTES Y SUS CITAS MÉDICAS

En este ejercicio se va a programar en JavaScript (solo lado cliente) una pequeña parte de la API especificada en el ejercicio anterior, que consuma una API REST dedicada a:

- la **gestión de pacientes**, y
- la **gestión de sus citas médicas**.

No se debe crear el *backend*, sino **usar correctamente fetch y JavaScript moderno** para interactuar con esa API:

- realizar distintas peticiones HTTP (GET, POST, PATCH o PUT, DELETE),
- enviar y recibir datos en formato JSON,
- procesar respuestas mediante ***destructuring***,
- manejar códigos de estado HTTP y errores de forma robusta.

Puedes trabajar contra una API *mock* (Postman Mock Server, etc.)

Además, para la implementación ten en cuenta las siguientes reglas:

- Debes usar **JavaScript ES6+**.
- Todas las operaciones asíncronas deben implementarse con `async`/`await`.
- Debes utilizar `fetch()` en todas las operaciones de red.
- Debes utilizar **destructuring** sobre la respuesta JSON (por ejemplo, `const { id, name } = patient;`).
- Debes comprobar siempre `res.ok` y, en caso contrario, tratar el error de forma apropiada. Para ello, contempla el el manejo de errores con `try/catch`, incluyendo los errores de red.
- No se permite el uso de librerías externas.

OBJETIVOS ESPECÍFICOS

Al finalizar el ejercicio, se deberá hacer lo siguiente:

1. Construir peticiones HTTP usando fetch con **métodos distintos** (GET, POST, PATCH/PUT, DELETE).
2. Recoger datos introducidos por el usuario (por ejemplo, a través de formularios HTML) y enviarlos en el cuerpo de la petición.
3. Consumir respuestas JSON y aplicar ***destructuring*** para extraer campos de manera clara.
4. Manejar errores de red y errores HTTP (`res.ok`, `res.status`).
5. Estructurar el código en funciones reutilizables, con nombres semánticos.

FUNCIONALIDAD MÍNIMA A IMPLEMENTAR

Aunque se puede tomar toda la API especificada en el ejercicio anterior, el código mínimo JS generado debe implementar las siguientes funcionalidades:

LISTAR PACIENTES (GET)

- Implementa una función que obtenga el **listado de pacientes** desde la API.
- La función debe:
    - realizar una petición **GET** con `fetch`,

- procesar la respuesta JSON,
- usar ***destructuring*** para obtener campos relevantes (por ejemplo, `id`, `name`, `email`),
- mostrar por consola o en el DOM el listado resultante.

Requisito: esta función debe usar `async`/`await` y comprobar `res.ok`.

CREAR UN NUEVO PACIENTE (POST)

- Implementa una función que permita **crear un paciente nuevo** enviando los datos en formato JSON.
- Los datos pueden recogerse desde un formulario HTML o desde un objeto JS construido en el código.
- La función debe:
  - enviar la petición **POST** con el cuerpo JSON (`body: JSON.stringify(...)`),
  - establecer correctamente la cabecera `Content-Type: application/json`,
  - interpretar la respuesta (por ejemplo, un `201 Created`),
  - usar ***destructuring*** para extraer del JSON el id del nuevo paciente y otros campos clave.

Requisito: manejar el caso en que el servidor devuelva un error de validación (por ejemplo, `400` o `422`) y mostrar un mensaje significativo.

LISTAR CITAS DE UN PACIENTE (GET CON PARÁMETROS)

- Implementa una función que, dado el identificador de un paciente, obtenga sus **citas médicas**.
- La función debe:
  - construir una URL con parámetros (por ejemplo, `?patientId=...` o siguiendo la estructura de la API objetivo),
  - usar URL y/o `URLSearchParams` para construir la *query string*,
  - realizar una petición **GET** con fetch,
  - procesar el JSON de la respuesta usando **destructuring** para extraer datos como `date`, `time`, `status`, etc.

Requisito: manejar correctamente el caso en que el paciente no tenga citas o no exista (`404`).

ACTUALIZAR EL ESTADO DE UNA CITA (PATCH O PUT)

- Implementa una función que permita **actualizar el estado** de una cita (por ejemplo, cambiarla a "confirmada", "cancelada", etc.).
- La función debe:
  - recibir al menos el `id` de la cita y el nuevo estado,
  - realizar una petición **PATCH** o **PUT** (según defina la API) con el cuerpo JSON,
  - verificar el código de respuesta (`200`, `204`, etc.),
  - mostrar un mensaje indicando el resultado de la operación.

Requisito: usar ***destructuring*** en la respuesta (si hay cuerpo) para extraer el nuevo estado u otros campos relevantes.

ELIMINAR UNA CITA (DELETE)

- Implementa una función que permita **eliminar una cita médica** dado su identificador.
- La función debe:
  - realizar una petición **DELETE**,

- manejar la respuesta típica (`204 No Content` u otra que defina la API),
- reaccionar adecuadamente si la cita no existe (`404`).

Requisito: la función debe diferenciar entre “eliminación correcta” y “cita inexistente” según el código de estado.

RESULTADO

Como resultado de este ejercicio se espera:

1. Uno o varios archivos **.js** con las funciones implementadas.
2. Un archivo **.html** mínimo que permita invocar dichas funciones (botones, formularios o uso desde consola).