Actividad

Diseño de una Solución Backend basada en Microservicios

✨ Objetivo general

Nos vamos a poner el sombrero de **arquitectos de soluciones backend** para una problemática realista, identificando entidades, recursos, sus relaciones, y dividiendo el sistema en **microservicios independientes y cohesivos**. Esperamos que la solución resultante respete los principios REST y las **buenas prácticas de diseño de rutas** que trabajamos en clase.

✨ Introducción

Como arquitectos de una API orientada a microservicios, el diseño de las URIs (rutas de acceso a los recursos) es una de las decisiones más importantes.

Recordemos algunas **buenas prácticas de diseño de APIs REST**:

- Representar los nombres de los recursos como **sustantivos en plural**.
- Evitar verbos en las rutas: el verbo lo define el método HTTP (GET, POST, PUT, DELETE).
- Usar rutas jerárquicas para representar relaciones (ej: `/clientes/{id}/facturas`).
- Utilizar filtros, paginación y ordenamiento como **query params** (`?page=2&sort=fecha`).
- Incluir **sub-recursos** cuando corresponda, pero evitando rutas excesivamente profundas.
- No exponer detalles técnicos o internos de la implementación.
- Incluir el prefijo `/api/v1/` para el versionado. Aunque el versionado puede ser opcional en esta instancia, el uso de `/api` como prefijo es obligatorio.

Además, cuando diseñamos una solución basada en microservicios, debemos tener en cuenta cómo **agrupar endpoints en servicios independientes**, minimizando el acoplamiento y maximizando la cohesión. Algunas recomendaciones generales:

- Un microservicio debe encargarse de un **conjunto funcional coherente** (por ejemplo, gestión de festivales, manejo de contenidos, o servicios de archivos).
- Debemos evitar que múltiples microservicios necesiten modificar la misma entidad.
- Es preferible lograr **consistencia eventual** antes que depender de transacciones distribuidas.
  La consistencia eventual significa que, en sistemas distribuidos, los datos pueden no estar sincronizados instantáneamente entre servicios, pero con el tiempo llegarán a un estado consistente. Por ejemplo, si un servicio registra una obra y otro servicio la muestra en un listado, puede haber un pequeño retraso hasta que esa obra aparezca reflejada en la interfaz de usuario. Este enfoque permite mejorar el rendimiento y la escalabilidad, aceptando una pequeña latencia a cambio de menor acoplamiento entre servicios.
- Los microservicios deben comunicarse entre sí a través de APIs bien definidas, evitando compartir bases de datos.

Un último tema a tener en cuenta en este rol de diseñadores, que sin embargo no es por ello menos importante, es que en este punto no nos vamos a ocupar de la estructura de datos de la/s base/s de datos

asociadas a los microservicios sino que nos vamos a encargar de pensar en terminos de recursos REST.

Es decir que al implementar este sistema podríamos almacernar datos en una base de datos relacional o hacerlo en un nosql de forma indistinta y la estructura de URIs y Microservicios debería mantenerse sin ser alterada.

Todas las decisiones que tomemos sobre rutas y diseño de servicios deben justificarse en base a estos criterios.

🔷 Dominio del problema

Vas a trabajar con un dominio centrado en la **gestión de eventos culturales y su difusión en línea**. El sistema permitirá gestionar festivales, obras artísticas, artistas, publicaciones y recursos multimedia.

Este dominio busca reflejar situaciones reales que requieren el uso de distintos tipos de datos, estructuras jerárquicas y momentos de captura de información distribuidos en el tiempo.

También se incluyen **archivos físicos** (imágenes, afiches, audios, videos, etc.) que no se almacenan en la base de datos, sino en un sistema de almacenamiento externo (por ejemplo, un filesystem o almacenamiento en la nube). La base de datos sólo debe almacenar la **ruta al archivo** y **metadatos** asociados como nombre, tipo, tamaño, y fecha de carga. Estos archivos deben poder ser consultados o descargados mediante endpoints específicos y protegidos.

Algunas entidades clave

- **Festival**: nombre, descripción, fecha de inicio y fin, ubicación, estado (borrador, publicado, cancelado), imagen de portada.
- **Obra**: pertenece a un festival. Tiene título, descripción, género, duración, archivo de afiche, y puede tener varios artistas asociados.
- **Artista**: nombre artístico, nacionalidad, biografía, foto.
- **Publicación**: noticias o actualizaciones asociadas a un festival, con fecha y contenido. Puede tener archivos adjuntos.
- **Configuración**: parámetros del sistema como cantidad de obras por festival, habilitación de comentarios, etc. No deben modelarse como entidades CRUD, sino como valores de configuración leídos desde una fuente centralizada no editable por usuarios comunes.

🌐 Requisitos funcionales

Nuestra solución debe contemplar un conjunto de funcionalidades que cubran tanto la gestión operativa como la organización lógica de los recursos en el tiempo. A continuación detallamos los requisitos mínimos esperados para guiar el diseño de los endpoints:

- **Alta, modificación, consulta y baja de Festivales**: Debemos poder crear un festival en estado "borrador", publicarlo o cancelarlo, y actualizar su información general (nombre, ubicación, fechas, descripción).

- **Cambio de estado del Festival**: Los festivales pueden pasar de "borrador" a "publicado", o ser "cancelados". Este cambio no implica eliminar los datos, sino cambiar su estado. Según el estado,

otras operaciones pueden habilitarse o bloquearse (por ejemplo, agregar obras solo cuando el festival esté publicado).

- **Gestión de Obras artísticas**: Las obras deben poder asociarse a festivales existentes, incluso en distintos momentos del tiempo. Debemos registrar su título, descripción, género, duración, y un archivo digital (afiche). También debemos poder modificarlas o eliminarlas individualmente.
- **Asociación de Artistas a Obras**: Cada obra puede tener uno o varios artistas. Esta asociación debe gestionarse como una operación específica (ej: `POST /obras/{id}/artistas`), sin necesidad de modificar toda la obra.
- **Gestión de Artistas**: Vamos a crear, consultar, actualizar y eliminar artistas con sus datos personales (nombre artístico, nacionalidad, biografía y foto). La foto es un archivo físico que debe subirse y vincularse.
- **Publicaciones del Festival**: Cada festival puede tener múltiples publicaciones (novedades, notas de prensa, etc.), con contenido, fecha, título y archivos adjuntos. Debemos poder agregarlas, editarlas y eliminarlas mientras el festival esté activo.
- **Subida y consulta de Archivos**: Cualquier entidad que tenga un archivo asociado (portada del festival, afiche de obra, foto de artista, adjunto de publicación) debe permitir la subida del archivo, el almacenamiento de su ruta y metadatos, y la consulta o descarga del mismo desde un endpoint específico (ej: `POST /multimedia/afiches`).
- **Dashboard de indicadores**: Debemos ofrecer un recurso o endpoint que devuelva un resumen por festival: cantidad total de obras, cantidad de artistas, número de publicaciones realizadas, estado actual. Este recurso puede utilizarse para construir una vista tipo panel de control.

Estos requerimientos son el punto de partida obligatorio, pero podemos ampliarlos si lo consideramos conveniente y coherente con el modelo propuesto.

🚀 Nuestra misión

1. **Identificar y justificar una división en 2 o 3 microservicios** para el dominio planteado.

   Para tomar esta decisión, vamos a considerar los principios de separación de responsabilidades, bajo acoplamiento y alta cohesión. Justificamos cómo agrupamos los endpoints para lograr independencia entre servicios y una navegación clara de los recursos.

2. Para cada microservicio, debemos identificar:

   - Recursos principales y sub-recursos.
   - Endpoints CRUD.
   - Endpoints especiales (cambio de estado, dashboard, etc.).
   - URIs para cada uno de los recuros y ejemplos de Endpoints con los verbos agregados.

3. Aplicamos las **buenas prácticas de diseño** documentadas aquí y en el material y documentamos las deciciones.

4. Representamos los endpoints con una de estas opciones:

    - Como listado plano de URIs y métodos HTTP.
    - Como definición Swagger/OpenAPI.

5. De cada endpoint se espera request y el response con especial enfoque en los códigos de estado HTTP para cada uno.

🏆 Criterios de aceptación

- Aplicación correcta de buenas prácticas REST.
- División coherente de microservicios.
- Claridad en la navegabilidad de recursos.
- Justificación de decisiones tomadas.
- Uso adecuado de tipos de datos, rutas y parámetros.

ℹ️ Recursos

Podemos apoyarnos en el Apunte de diseño de APIs, en ejemplos previos y en herramientas como:

- Swagger Editor
- API Design Guidelines de Microsoft
- HTTP status cats

📅 Entrega

Se espera que el resultado sea: o bien un un archivo `.md`, o bien una exportación Swagger `.yaml`, o una colección documentada que represente los endpoints.