

Práctica Marketplace de Anuncios

1. Descripción del Proyecto

El objetivo de la práctica es desarrollar una aplicación web tipo "Marketplace" para la compra y venta de artículos. La aplicación debe ser **Fullstack** utilizando las capacidades de **Next.js**, donde la prioridad es la eficiencia, el renderizado en el servidor y la experiencia de usuario optimizada mediante *Streaming* y *Server Actions*.

El alumno debe demostrar que ha asimilado el "**Nuevo Modelo Mental**": mover la lógica pesada al servidor y dejar al cliente solo para la interactividad estricta.

2. Requisitos Técnicos Obligatorios

- **Framework:** Next.js (última versión estable) usando **App Router**.
- **Lenguaje:** TypeScript (recomendado) o JavaScript moderno.
- **Estilos:** Libre (CSS Modules, Tailwind, o librerías de componentes), pero priorizando la carga rápida.
- **Gestión de Estado:**
 - **Estado del Servidor:** No se permite el uso de librerías globales de cliente (como Redux o Zustand) para datos que residen en la base de datos (anuncios). Se deben usar **Server Components**.
 - **Estado de la URL:** Los filtros y la paginación deben gestionarse mediante *URL Search Params*.

3. Funcionalidades e Hitos de Desarrollo

Hito 1: Listado de Anuncios (RSC & Data Fetching)

La página principal (/) debe renderizar el listado de anuncios disponibles.

- **Server First:** El *fetching* de datos debe ocurrir en un Server Component. No se debe usar *useEffect* para cargar la lista inicial.
- **Filtros vía URL:** Implementar un buscador por nombre y filtros por precio o tags. Al filtrar, la URL debe cambiar (ej: /?query=moto&price=1000). La página debe recargarse con los nuevos datos sin perder el estado de la navegación.
- **Loading UI (Streaming):** Crear un archivo loading.js que muestre un *Skeleton* o indicador de carga mientras el servidor obtiene los anuncios. Evitar que la pantalla se quede en blanco durante la navegación.

Hito 2: Detalle del Anuncio (Dynamic Routes & Metadata)

Al hacer clic en un anuncio, se debe navegar a /ads/[id].

- **Generación Dinámica:** La página debe obtener los datos del anuncio basándose en el ID de la URL.
- **SEO Básico:** Usar la API de generateMetadata para que el título de la pestaña del navegador y la descripción coincidan con el nombre y precio del anuncio.

Hito 3: Autenticación y Mutaciones (Server Actions)

Implementar un sistema de Login y Creación de Anuncios eliminando la dependencia de APIs REST en el cliente.

- **Login:** Crear un formulario que utilice una **Server Action** para autenticar al usuario.
- **Crear Anuncio:**
 - El formulario debe funcionar mediante una **Server Action**.
 - **Validación:** Validar los datos en el servidor (usando **Zod** o similar) antes de enviarlos al backend.
 - **Revalidación:** Tras crear un anuncio, usar revalidatePath para que el listado principal se actualice automáticamente y redirigir al usuario al *Home*.
- **Feedback Visual:** Usar el hook useActionState (o useFormState) para gestionar los estados del formulario (cargando, éxito, error) y mostrar mensajes de error (ej: "Email inválido") directamente en los inputs.

Hito 4: Manejo de Errores (Error Boundaries)

La aplicación debe ser resiliente a fallos.

- **Página 404:** Implementar not-found.js para manejar casos donde un usuario intenta acceder a un anuncio que no existe.
- **Captura de Errores:** Implementar error.js ("Las Matrioshkas") para capturar errores inesperados en el renderizado (ej: fallo de conexión con el backend) y mostrar una interfaz amigable con un botón para "Intentar de nuevo" (reset).

4. Requisitos "Pro" (Opcionales para nota máxima)

1. **Middleware de Protección:** Proteger la ruta de /ads/create. Si un usuario no está logueado e intenta entrar, el Middleware debe interceptarlo y redirigirlo al Login.
2. **Optimistic UI:** Implementar un botón de "Me gusta" o "Favorito" en los anuncios que utilice useOptimistic para actualizar la interfaz instantáneamente antes de que el servidor confirme la acción.
3. **Componente Cliente vs Servidor:** Demostrar una separación clara. Por ejemplo, el botón de "Comprar/Reservar" debe ser un *Client Component* (interactivo) incrustado dentro de la página de detalle que es un *Server Component*.

5. Testing (Calidad)

Siguiendo la pirámide de testing vista en el curso:

- **Unit Testing de Server Actions:** Crear al menos un test unitario (con Jest/Vitest) para una Server Action (ej: función de login o creación). Se debe comprobar que la función retorna los errores correctos ante datos inválidos y éxito ante datos válidos. No es necesario testear la UI de React, solo la lógica de la acción.