Pistas: Conectar la pagina de detalle de embalse con MongoDB

Objetivo: la pagina `front/src/app/embalse/[embalse]/page.tsx` recibe un **slug** como parametro de ruta (ej: `embalse-de-alarcon`). Vamos a conectarla con MongoDB para mostrar datos reales en vez de los datos hardcodeados (mock).

URL de prueba: `http://localhost:3000/embalse/embalse-de-alarcon`

Paso 1: Instalar dependencias

Necesitamos dos dependencias nuevas en `front/package.json`:

- `mongodb` — el driver oficial de MongoDB para Node.js
- `db-model` — nuestro paquete local que tiene la interfaz `Embalse` (el modelo de la BD)

> No usamos `@embalse-info/db` porque arrastra transitivamente `arcgis` y las integraciones SAIH que no necesitamos en el front.

Anade en la seccion `dependencies` de `front/package.json`:

```
"mongodb": "^6.12.0",
"db-model": "file:../packages/db-model"
```

Despues ejecuta `npm install` dentro de la carpeta `front`.

> Asegurate de que el paquete `db-model` este compilado (que exista `packages/db-model/dist/`). Si no existe, entra en `packages/db-model` y ejecuta `npm run build`.

Paso 2: Crear el fichero de variables de entorno

Crea el fichero `front/.env.local` con el siguiente contenido:

```
MONGODB_CONNECTION_STRING=mongodb://localhost:27017/embalse-info
```

Next.js carga automaticamente los ficheros `.env.local` y los expone en `process.env` del lado servidor.

Paso 3: Crear el cliente MongoDB singleton

Crea el fichero `front/src/lib/mongodb.ts`.

Este es el patron estandar de Next.js para conectarse a MongoDB:

- En desarrollo, Next.js hace hot-reload y re-ejecuta los modulos. Si no cacheamos el cliente, cada recarga crearia una conexion nueva.
- Solucion: guardamos la instancia de `MongoClient` en `globalThis` para que sobreviva entre recargas.

```
import { MongoClient, type Db } from "mongodb";

const connectionString = process.env.MONGODB_CONNECTION_STRING;

if (!connectionString) {
  throw new Error(
    "Please define the MONGODB_CONNECTION_STRING environment variable in
.env.local"
  );
}

const globalForMongo = globalThis as typeof globalThis & {
  _mongoClient?: MongoClient;
};

function getClient(): MongoClient {
  if (!globalForMongo._mongoClient) {
    globalForMongo._mongoClient = new MongoClient(connectionString);
  }
  return globalForMongo._mongoClient;
}

export function getDb(): Db {
  return getClient().db();
}
```

Que hace cada parte:

- `connectionString` — lee la URL de MongoDB de las variables de entorno
- `globalForMongo` — hack de TypeScript para tipar `globalThis` con nuestra propiedad custom `_mongoClient`
- `getClient()` — crea el cliente solo si no existe ya (singleton)
- `getDb()` — lo que exportamos; devuelve la instancia de `Db` lista para hacer queries

---

Paso 4: Crear el repositorio (server action)

Crea el fichero `front/src/pods/embalse/embalse.repository.ts`.

Esta es la capa que habla con MongoDB. Usamos la directiva `"use server"` para que Next.js lo ejecute solo en el servidor.

```
"use server";
```

```ts
import { getDb } from "@/lib/mongodb";
import type { Embalse } from "db-model";

export async function getEmbalseBySlug(
  slug: string
): Promise<Embalse | null> {
  const db = getDb();
  const doc = await db.collection("embalses").findOne({ slug });

  if (!doc) {
    return null;
  }

  return {
    _id: doc._id.toString(),
    embalse_id: doc.embalse_id,
    nombre: doc.nombre,
    slug: doc.slug,
    cuenca: {
      _id: doc.cuenca?._id?.toString() ?? "",
      nombre: doc.cuenca?.nombre ?? "",
    },
    provincia: doc.provincia ?? null,
    capacidad: doc.capacidad,
    aguaActualAemet: doc.aguaActualAemet ?? null,
    fechaMedidaAguaActualAemet: doc.fechaMedidaAguaActualAemet ?? null,
    aguaActualSAIH: doc.aguaActualSAIH ?? null,
    fechaMedidaAguaActualSAIH: doc.fechaMedidaAguaActualSAIH ?? null,
    descripcion_id: doc.descripcion_id ?? null,
    uso: doc.uso ?? "",
  };
}
```

Por que serializamos manualmente?

Los objetos que devuelve MongoDB contienen tipos como `ObjectId` y `Date` que **no son serializables** directamente en server actions de Next.js (solo se pueden pasar tipos planos: strings, numbers, nulls...). Por eso convertimos `_id` a string con `.toString()` y nos aseguramos de que cada campo sea un tipo primitivo.

Referencia: la interfaz Embalse (de db-model)

Para que entiendas que campos tiene el documento en MongoDB:

```ts
export interface Embalse {
  _id: string;
  embalse_id: number;
  nombre: string;
  slug: string;
  cuenca: {
    _id: string;
```

```
    nombre: string;
  };
  provincia: string | null;
  capacidad: number;
  aguaActualAemet: number | null;
  fechaMedidaAguaActualAemet: Date | null;
  aguaActualSAIH: number | null;
  fechaMedidaAguaActualSAIH: Date | null;
  descripcion_id: string | null;
  uso: string;
}
```

Paso 5: Actualizar el View Model

Abre `front/src/pods/embalse/embalse.vm.ts`.

Necesitamos anadir el campo `nombre` a la interfaz `ReservoirData` para poder mostrar el nombre real del embalse en la cabecera (en vez del slug).

Anade `nombre: string` como primer campo de `ReservoirData`:

```
export interface ReservoirData {
  nombre: string;       // <-- NUEVO
  currentVolume: number;
  totalCapacity: number;
  measurementDate: string;
  datosEmbalse: DatosEmbalse;
  reservoirInfo: ReservoirInfo;
}
```

> Las interfaces `DatosEmbalse` y `ReservoirInfo` no cambian.

Paso 6: Actualizar los datos mock

Abre `front/src/pods/embalse/embalse-mock-data.ts`.

Tras anadir `nombre` al view model, el mock data ya no compila. Anade el campo:

```
export const MOCK_DATA: ReservoirData = {
  nombre: "Embalse Ejemplo",   // <-- NUEVO
  currentVolume: 1500,
  totalCapacity: 50000,
  // ... resto igual
};
```

Paso 7: Crear el mapper

Crea el fichero `front/src/pods/embalse/embalse.mapper.ts`.

Este fichero transforma un documento `Embalse` (de BD) al view model `ReservoirData` (de UI).

```
import type { Embalse } from "db-model";
import type { ReservoirData } from "./embalse.vm";

function formatDate(date: Date | string | null | undefined): string {
  if (!date) return "";
  const d = typeof date === "string" ? new Date(date) : date;
  if (isNaN(d.getTime())) return "";
  const day = String(d.getDate()).padStart(2, "0");
  const month = String(d.getMonth() + 1).padStart(2, "0");
  const year = d.getFullYear();
  return `${day}/${month}/${year}`;
}

export function mapEmbalseToReservoirData(embalse: Embalse): ReservoirData
{
  const currentVolume =
    embalse.aguaActualSAIH ?? embalse.aguaActualAemet ?? 0;

  const measurementDate = formatDate(
    embalse.fechaMedidaAguaActualSAIH ??
embalse.fechaMedidaAguaActualAemet
  );

  return {
    nombre: embalse.nombre,
    currentVolume,
    totalCapacity: embalse.capacidad,
    measurementDate,
    datosEmbalse: {
      cuenca: embalse.cuenca?.nombre ?? "",
      provincia: embalse.provincia ?? "",
      municipio: "",
      rio: "",
      embalsesAguasAbajo: 0,
      tipoDePresa: "",
      anioConstruccion: 0,
      superficie: 0,
      localizacion: "",
    },
    reservoirInfo: {
      Description: "",
    },
  };
}
```

Decisiones del mapeo:

- **currentVolume**: prioriza el dato SAIH sobre el de Aemet (SAIH es mas frecuente). Si ambos son `null`, pone 0.
- **measurementDate**: mismo criterio de prioridad, formateado a `DD/MM/YYYY`.
- **nombre**: viene directamente del campo `nombre` del documento.
- **Campos vacios**: `municipio`, `rio`, `tipoDePresa`, etc. no estan en el modelo de BD actual, asi que van como string vacio / 0.
- **formatDate**: acepta tanto `Date` como `string` (MongoDB puede devolver ISO strings en vez de Date dependiendo de la serializacion).

Paso 8: Conectar la pagina con el server action + ISR

Abre `front/src/app/embalse/[embalse]/page.tsx`.

Ademas de conectar con MongoDB, vamos a activar **ISR (Incremental Static Regeneration)** para que Next.js cachee la pagina y la regenere cada 5 minutos. Asi no se hace una query a MongoDB en cada visita, pero los datos se mantienen razonablemente frescos.

Antes teniamos:

```
import { EmbalsePod } from "@/pods/embalse";

interface Props {
  params: Promise<{ embalse: string }>;
}

export default async function EmbalseDetallePage({ params }: Props) {
  const { embalse } = await params;
  return <EmbalsePod embalse={embalse} />;
}
```

Cambialo a:

```
import { notFound } from "next/navigation";
import { EmbalsePod } from "@/pods/embalse";
import { getEmbalseBySlug } from "@/pods/embalse/embalse.repository";
import { mapEmbalseToReservoirData } from "@/pods/embalse/embalse.mapper";

export const revalidate = 300; // ISR: regenerar cada 5 minutos

interface Props {
  params: Promise<{ embalse: string }>;
}

export default async function EmbalseDetallePage({ params }: Props) {
  const { embalse } = await params;
```

```
  const embalseDoc = await getEmbalseBySlug(embalse);

  if (!embalseDoc) {
    notFound();
  }

  const reservoirData = mapEmbalseToReservoirData(embalseDoc);

  return <EmbalsePod reservoirData={reservoirData} />;
}
```

Que cambia:

1. Importamos `notFound` de Next.js, el repositorio y el mapper.
2. Exportamos `revalidate = 300` — esto activa ISR. Next.js cachea la pagina generada y la regenera en background cada 300 segundos (5 minutos). La primera visita tras expirar el cache sirve la version antigua y dispara la regeneracion; la siguiente visita ya ve los datos actualizados.
3. Llamamos a `getEmbalseBySlug` con el slug de la URL.
4. Si no se encuentra el embalse, llamamos a `notFound()` que muestra la pagina 404 de Next.js.
5. Si se encuentra, mapeamos el documento a `ReservoirData` y lo pasamos al pod.
6. La prop cambia de `embalse: string` a `reservoirData: ReservoirData`.

Paso 9: Actualizar el Pod

Abre `front/src/pods/embalse/embalse.pod.tsx`.

Cambia la prop de `embalse: string` a `reservoirData: ReservoirData`:

```
import React from "react";
import { Embalse } from "./embalse.component";
import { ReservoirData } from "./embalse.vm";

interface Props {
  reservoirData: ReservoirData;
}

export const EmbalsePod: React.FC<Props> = (props) => {
  const { reservoirData } = props;

  return <Embalse reservoirData={reservoirData} />;
};
```

Paso 10: Actualizar el componente Embalse

Abre `front/src/pods/embalse/embalse.component.tsx`.

Dos cambios clave:

1. Recibir `reservoirData` como prop en vez de `embalse: string`.
2. Eliminar el import de `MOCK_DATA` y usar los datos reales.

```
import React from "react";
import {
  ReservoirCardDetail,
  ReservoirCardGauge,
  ReservoirCardInfo,
} from "./components";
import { ReservoirData } from "./embalse.vm";

interface Props {
  reservoirData: ReservoirData;
}

export const Embalse: React.FC<Props> = (props) => {
  const { reservoirData } = props;
  return (
    <div className="flex flex-col gap-8">
      <div className="space-y-6">
        <ReservoirCardGauge
          name={reservoirData.nombre}
          reservoirData={reservoirData}
        />
        <div className="card bg-base-100 mx-auto w-full max-w-[400px]
items-center gap-6 rounded-2xl p-4 shadow-lg">
          <ReservoirCardInfo reservoirInfo={reservoirData.reservoirInfo}
/>
        </div>
        <div className="card bg-base-100 mx-auto w-full max-w-[400px]
items-center gap-6 rounded-2xl p-4 shadow-lg">
          <ReservoirCardDetail datosEmbalse={reservoirData.datosEmbalse}
/>
        </div>
      </div>
    </div>
  );
};
```

Que cambia:

- Ya no importamos `MOCK_DATA`.
- La prop `name` de `ReservoirCardGauge` ahora recibe `reservoirData.nombre` (el nombre real del embalse desde la BD).
- El resto de componentes hijos (`ReservoirCardInfo`, `ReservoirCardDetail`) reciben los datos mapeados.

Paso 11: Calcular el porcentaje real en el gauge

Abre `front/src/pods/embalse/components/reservoir-card-gauge.tsx`.

Hay tres cambios:

1. **Descomentar el calculo de porcentaje** y anadir proteccion contra division por cero.
2. **Usar el porcentaje real** en `GaugeChart` (en vez del `0.67` hardcodeado).
3. **Quitar el prefijo "Embalse de"** del `<h2>` (el campo `nombre` de la BD ya incluye el nombre completo, ej: "Embalse de Alarcon").

```
import { ReservoirData } from "../embalse.vm";
import { GaugeChart } from "./reservoir-gauge";
import { GaugeLegend } from "./reservoir-gauge/gauge-
chart/components/gauge-legend.component";

interface Props {
  name: string;
  reservoirData: ReservoirData;
}

export const ReservoirCardGauge: React.FC<Props> = (props) => {
  const { name, reservoirData } = props;
  const { currentVolume, totalCapacity, measurementDate } = reservoirData;
  const percentage = totalCapacity > 0 ? currentVolume / totalCapacity :
0;

  return (
    <div className="card bg-base-100 mx-auto w-full max-w-[400px] items-
center gap-6 rounded-2xl p-4 shadow-lg">
      <h2 className="text-center">{name}</h2>
      <GaugeChart percentage={percentage} measurementDate=
{measurementDate} />
      <GaugeLegend
        currentVolume={currentVolume}
        totalCapacity={totalCapacity}
      />
    </div>
  );
};
```

Diferencias con el codigo anterior:

Antes	Despues
`// const percentage = currentVolume / totalCapacity;`	`const percentage = totalCapacity > 0 ? currentVolume / totalCapacity : 0;`
`<GaugeChart percentage={0.67} ...`	`<GaugeChart percentage={percentage} ...`
`<h2>Embalse de {name}</h2>`	`<h2>{name}</h2>`

Resumen de ficheros

Fichero	Accion
`front/package.json`	Modificar: anadir `mongodb` y `db-model`
`front/.env.local`	**Crear**: variable de entorno MongoDB
`front/src/lib/mongodb.ts`	**Crear**: cliente singleton
`front/src/pods/embalse/embalse.repository.ts`	**Crear**: server action con query a MongoDB
`front/src/pods/embalse/embalse.mapper.ts`	**Crear**: mapper Embalse -> ReservoirData
`front/src/pods/embalse/embalse.vm.ts`	Modificar: anadir `nombre`
`front/src/pods/embalse/embalse-mock-data.ts`	Modificar: anadir `nombre`
`front/src/app/embalse/[embalse]/page.tsx`	Modificar: llamar al repository + mapper
`front/src/pods/embalse/embalse.pod.tsx`	Modificar: cambiar prop a `reservoirData`
`front/src/pods/embalse/embalse.component.tsx`	Modificar: usar datos reales, quitar mock
`front/src/pods/embalse/components/reservoir-card-gauge.tsx`	Modificar: porcentaje real, quitar prefijo

Verificacion

1. Asegurate de que MongoDB esta corriendo con datos en la BD `embalse-info` (coleccion `embalses` con campo `slug` poblado).
2. `npm install` en el directorio `front`.
3. `npm run start` en el directorio `front`.
4. Navega a `http://localhost:3000/embalse/embalse-de-alarcon` (u otro slug que exista en la BD).
5. Comprueba que se muestran los datos reales: nombre, capacidad, volumen actual, fecha, cuenca, provincia.
6. Comprueba que el gauge muestra el porcentaje real calculado (no el 67% hardcodeado).
7. Prueba con un slug que no exista -> debe mostrar la pagina 404 de Next.js.