UNIVERSIDAD NACIONAL DE SAN MARTÍN

# Programación con Herramientas Modernas

# Trabajo práctico 2025

## Dominio y precondiciones

El dominio sigue siendo el mismo de la entrega 0. Asumimos que están persistidos los objetos de dominio en una base de datos relacional + una documental.

Esta entrega suma tres capacidades sobre lo existente: una **capa de cache** con Redis, una **API GraphQL** que convive con la REST actual, y el **deploy en la nube** del monolito políglota.

---

### Parte 1: Cache con Redis

Dado que el Home es la pantalla de mayor tráfico, calcular esto en tiempo real por cada petición degrada severamente la performance. Para solucionar esto, el grupo deberá incorporar **Redis** como motor de caché en memoria.

Caché de "Libros Populares" en la búsqueda principal: En lugar de ir a MongoDB a buscar los libros y contar los clicks cada vez que un usuario entra al home, debemos actualizar al momento de un click para que guarde en Redis el conteo para obtener el "Ranking de libros más populares".

En la pantalla de inicio al entrar a la app lee directamente de Redis el ranking obteniendo los 10 primeros elementos, luego intenta obtener de ellos al menos 6 libros cacheados para mostrar logrando una latencia mínima. En caso de no tener éxito, se realiza la búsqueda en MongoDB.

La caché de los libros se actualiza al momento de realizar una búsqueda en MongoDB. Deben estimar un TTL para esta información que crean coherente. Luego las búsquedas siguen realizándose en la base documental.
Sugerimos investigar sorted set.

## Parte 2: GraphQL

La aplicación deberá exponer una **API GraphQL** que **conviva** con la API REST existente (no la reemplaza). Se sugiere usar **Netflix DGS** sobre Spring Boot, de acuerdo al ejemplo de la materia.

Vamos a modelar un tablero con KPI's para un eventual administrador. Cada integrante deberá extender el esquema de **GraphQL** y construir el *Resolver* (o *Data Fetcher*) correspondiente para su métrica asignada. El motor de **GraphQL** deberá orquestar las consultas para que, si el cliente pide todas las métricas juntas, se resuelvan eficientemente delegando la carga a los servicios correspondientes:

- **Integrante 1 - Tasa de Conversión (Clicks vs. Reservas):** Deberá resolver una consulta que cruce mundos. Se debe consultar a la base clave/valor (Redis) el *Top 5 de libros más clickeados* y cruzar esa información con la base relacional (PostgreSQL.) Devolver, por libro: clicks, reservas y tasa de conversión = reservas / clicks.
- **Integrante 2 - Leaderboard de Comunidad (Caché con Redis):** Deberá devolver el "Top 5 de usuarios con más Bibliokarmas". Dado que esta métrica es pesada de calcular y muy consultada, el resolver debe obligatoriamente intentar leer el ranking desde **Redis**. En caso de no existir (Cache Miss), deberá calcularlo desde PostgreSQL, guardarlo en Redis con un *Time To Live* (TTL) prudente, y devolver el resultado.
- **Integrante 3 - Estado de Salud del Catálogo:** El estado a nivel libro solo distingue AVAILABLE/RENTED, así que las cuatro cantidades originales se solapan. Redefinir en buckets disjuntos que sumen el total:
    - total
    - prestados: reserva vigente hoy.
    - disponiblesNuncaReservados: sin reservas.
    - disponiblesReservadosAFuturo: sin reserva vigente, con al menos una futura y ninguna finalizada.
    - disponiblesDevueltos: sin reserva vigente, con al menos una finalizada.

  prestados + disponiblesNuncaReservados + disponiblesDevueltos + disponiblesReservadosAFuturo = total.

- **Integrante 4 - Análisis de Calificaciones:** Devolver el promedio de calificación agrupado por tipo de libro (Común, Con dedicatoria,

Coleccionable). El promedio se calcula a partir de la puntuación que cada libro ya tiene agregada en la base documental, promediando esas puntuaciones dentro de cada tipo; es decir, un promedio de promedios, ponderado por libro. Los libros que todavía no tienen reseñas se excluyen del cálculo para no arrastrar el promedio hacia cero. El tipo de retorno en GraphQL es una lista de pares que asocia cada tipo de libro con su puntaje promedio

- **Integrante 5 - Feed de Actividad Reciente:** Agregar *createdAt* al documento Book (fecha de alta en el sistema, distinta de *publicationDate*). El feed unifica los últimos libros dados de alta (Book.createdAt) y las últimas reservas confirmadas (fecha de la reserva), en una lista heterogénea(union/interface de GraphQL), orden cronológico descendente, top 5, con *fecha*, *tipoEvento* y *usuario*.

**Bonus — Pantalla de KPI's** integrar los KPI a la pantalla principal en un header.

**Bonus — Schema stitching.** Agregar al tipo `Libro` un campo (`portada` o `metadataExterna`) cuyo resolver consulte una **API pública por ISBN** (p. ej. Open Library: `https://openlibrary.org/isbn/{isbn}.json`).

---

## Parte 3: Deploy en la nube

Desplegar el monolito en **Render**:

- **Web Service** para la app Spring Boot (build por Docker o por Gradle).
- **Key Value** (Redis/Valkey) para la cache de la Parte 1.
- **PostgreSQL** gestionado en Render.
- **MongoDB**: Render no ofrece Mongo gestionado, así que se usa **MongoDB Atlas**.
- Configurar las **variables de entorno / secrets** (connection strings, secreto JWT). Nada de credenciales en el repo.

**Decisiones a justificar y aclaraciones obligatorias:**

- El free tier de Atlas (M0) es un **único replica set, sin sharding**. El cluster de 2 shards de la Entrega 2 **no entra** en Atlas gratuito. El grupo debe decidir y justificar: (a) simplificar Mongo a una instancia única en la nube,

o (b) mantener el cluster sharded en local y desplegar solo app + Postgres + Key Value. Dejar la decisión documentada.

- El **Web Service gratuito se duerme tras 15 min de inactividad** y tiene *cold start* de 30–60 s. Para la corrección presencial, "calentar" el servicio con un request previo.
- El **Postgres gratuito de Render expira / se elimina a los ~30 días**. Tener el **script de datos versionado** para poder recrear la base. La pérdida de datos entre la entrega y la corrección no será excusa válida.