**Resumen Técnico: Depuración App Restaurantes (Spring Boot + JPA/Hibernate + PostgreSQL)**

**1. Problema Inicial y Síntomas:**

**Errores intermitentes y difíciles de reproducir en la aplicación desplegada (Railway) y a veces localmente.**

**Síntomas: Errores 500 Internal Server Error y 403 Forbidden en el frontend, toasts de error genéricos. A veces, las primeras llamadas a un endpoint fallaban y las siguientes funcionaban.**

**Errores Backend Clave (Intermitentes): Se identificaron varios errores de bajo nivel en los logs del backend, a menudo relacionados con la interacción JDBC/PostgreSQL:**

**org.postgresql.util.PSQLException: ERROR: prepared statement "S\_X" already exists (donde X era un número)**

**org.postgresql.util.PSQLException: ERROR: current transaction is aborted, commands ignored until end of transaction block**

**org.springframework.orm.jpa.JpaSystemException: ... Unable to commit against JDBC Connection**

**org.postgresql.util.PSQLException: ERROR: bind message supplies X parameters, but prepared statement "S\_Y" requires Z**

**Estos errores solían ocurrir durante operaciones de base de datos, a menudo durante el commit de la transacción o durante las comprobaciones de seguridad iniciales que implicaban buscar usuarios (findByEmail).**

**2. Problemas de Build y Entorno Solucionados:**

**MalformedInputException en mvn resources:resources: Causado por el archivo application.properties no estando guardado con codificación UTF-8.**

**Solución: Se corrigió la codificación del archivo a UTF-8 usando el editor/IDE.**

**ClassNotFoundException: org.apache.tomcat.jdbc.pool.DataSource al arrancar/testear: Causado por una inconsistencia: application.properties forzaba el uso de Tomcat JDBC Pool (spring.datasource.type), pero la dependencia tomcat-jdbc no estaba en el pom.xml (y HikariCP sí estaba).**

**Solución: Se eliminó la propiedad spring.datasource.type de application.properties, permitiendo a Spring Boot usar HikariCP (que estaba presente en el POM) por defecto.**

**Errores de Compilación "Fantasma" / cannot find symbol en IDE: Mensajes de error de tipos ilógicos (Required Long / Provided UserEntity, cannot find symbol para clases importadas) a pesar de que el código fuente y los imports eran correctos.**

**Causa: Estado interno corrupto o desincronizado del IDE (IntelliJ) y/o del build de Maven. También exacerbado por un error real al intentar usar un constructor incorrecto para un DTO de tipo record.**

**Solución: Se corrigió el error del constructor del record (new UserResponseDto(id, name, ...) en lugar de new UserResponseDto(entity)). Se aplicaron limpiezas profundas: mvn clean install/package, File -> Invalidate Caches / Restart en IntelliJ, borrar carpeta target.**

**Errores de Arranque en Railway (502 Bad Gateway): Causados inicialmente por fallos al arrancar la aplicación debido a variables de entorno incorrectas (JWT\_SECRET) o por los errores de base de datos (ClassNotFoundException del pool, errores de commit) que impedían que la aplicación se iniciara correctamente.**

**Solución: Se corrigieron las variables de entorno en Railway y se solucionaron los errores de arranque subyacentes.**

**3. Optimizaciones y Correcciones de Lógica Aplicadas:**

**Problemas N+1 Identificados y Solucionados: Se detectaron múltiples consultas N+1 al cargar entidades y sus relaciones LAZY durante el mapeo a DTOs.**

**RestaurantService (al mapear a RestaurantResponseDto): Faltaba cargar cuisine y userEntity.**

**OrderService (al mapear a OrderResponseDto): Faltaba cargar restaurantId, clientId, la colección details, y el Product dentro de cada detail.**

**ProductService (al mapear a ProductSummaryResponseDto): Faltaba cargar restaurant y category.**

**Solución: Se aplicó carga EAGER usando @EntityGraph(attributePaths = {...}) en los métodos correspondientes de JpaRepository (findById, findAll, findByCategory\_Id, etc.) o se usaron JOIN FETCH en consultas @Query personalizadas (como en OrderRepository).**

**Refactorización de Servicios (AuthService vs UserService): Se eliminó la duplicación de la lógica de creación de usuarios. UserService.createUser quedó como la única fuente para guardar UserEntity. AuthService.register ahora llama a userService.createUser y luego genera el token JWT para el auto-login. AuthService.login también se ajustó para llamar a userService.getUserProfile para obtener el DTO de respuesta.**

**Manejo de DTOs: Se consolidaron DTOs de entrada duplicados (RegisterRequest -> UserRequestDto). Se corrigió la instanciación de DTOs de tipo record. Se aseguró que los métodos de servicio devolvieran DTOs (getUserProfile) en lugar de entidades hacia el controlador.**

**Seguridad: Se añadieron comprobaciones de autorización que faltaban en varios métodos (ej: updateUser, deleteUser, deleteRestaurant, deleteOrder, findRestaurantsByOwnerId) para asegurar que solo los usuarios correctos (el propio usuario, el dueño del restaurante, o un admin) pudieran realizar acciones sensibles.**

**Borrado de Pedidos: Se discutió que el borrado físico es problemático. Se recomendó usar cambio de estado (CANCELADO) o borrado lógico (soft delete) en lugar de orderRepository.delete().**

**Manejo de Resultados Vacíos: Se cambió la lógica en métodos "find" para devolver Collections.emptyList() o Page.empty() en lugar de lanzar excepciones cuando no se encuentran resultados.**

**4. Problema Persistente y Workaround Actual:**

**A pesar de todas las correcciones y optimizaciones, persistían errores intermitentes de bajo nivel (prepared statement exists, transaction aborted, unable to commit) relacionados con la interacción JDBC/PostgreSQL, especialmente durante el commit de transacciones. La causa raíz exacta no pudo ser determinada con certeza (posible bug driver 42.7.x, interacción pool/driver/tx, problema entorno Railway/Supabase).**

**Workaround Aplicado: Se activó el parámetro ?prepareThreshold=0 en la URL de conexión JDBC (spring.datasource.url).**

**Efecto: Evita el uso de prepared statements del lado del servidor para consultas con parámetros, lo que eliminó (o redujo drásticamente hasta ser casi imperceptible) los errores intermitentes, logrando estabilidad en la aplicación.**

**Contrapartida: Introduce una penalización de rendimiento (lentitud observable) porque la base de datos debe re-analizar y re-planificar las consultas más a menudo.**

**5. Estado Final (Para la Demo/Portfolio):**

**La aplicación es funcionalmente correcta y estable. Los errores críticos de arranque, build y los errores intermitentes graves han sido solucionados o mitigados.**

**Se han aplicado buenas prácticas (DTOs, separación de servicios, seguridad básica, optimización N+1).**

**Existe un workaround conocido (prepareThreshold=0) activo para garantizar la estabilidad, a costa de un rendimiento subóptimo. Esto puede ser explicado como una decisión pragmática ante un problema de bajo nivel difícil de aislar en el contexto del proyecto.**