**UNIVERSIDAD NACIONAL DE INGENIERÍA**

**FACULTAD DE INGENIERÍA INDUSTRIAL Y DE SISTEMAS**



**ESCUELA DE INGENIERÍA DE SISTEMAS**

**Curso:**

**PROGRAMACIÓN ORIENTADA A OBJETOS**

**Proyecto:**

**Sistema de de control de pedidos en un taller de manufactura**

**Autores:**

Chicmana Arias, Piero Alejandro ([orcid.org/0009-0009-7963-6901](https://orcid.org/0009-0009-7963-6901))

Carlos Falcón, Enrique Armando ([orcid.org/0009-0003-6091-1332](https://orcid.org/0009-0003-6091-1332))

Chuctaya Quispe, Frank Daniel ([orcid.org/0009-0005-4351-1809](http://orcid.org/0009-0005-4351-1809))

**Docente:**

Mag. Coronel Castillo, Eric Gustavo ([orcid.org/0000-0003-0494-5629](http://orcid.org/0000-0003-0494-5629))

**LIMA – PERÚ**

**2025**

# DEDICATORIA

A nuestro querido profesor,

Este proyecto, más allá de ser un simple conjunto de líneas de código, es el reflejo de todo lo que hemos aprendido bajo su guía. Cada decisión tomada, cada algoritmo implementado y cada validación diseñada ha sido posible gracias a la base sólida que nos ha proporcionado. En un mundo que avanza a la velocidad de la inteligencia artificial, donde la tecnología redefine constantemente lo posible, usted ha sido nuestro mentor, enseñándonos no solo el "cómo", sino también el "por qué" de la programación.  
“DIVIDE Y VENCERÁS”

# ÍNDICE GENERAL

Pág.

[**DEDICATORIA 2**](#_62hdyorlex4y)

[**ÍNDICE GENERAL 3**](#_nvute62x0glr)

[**RESUMEN 5**](#_noj72wjnlow2)

[**ABSTRACT 6**](#_gztgf4ddh5qj)

[**INTRODUCCIÓN 7**](#_qkclk9f0ounn)

[**OBJETIVOS 8**](#_fr09bydjg5di)

[OBJETIVO GENERAL 8](#_mpb9g2epbduc)

[OBJETIVOS ESPECÍFICOS 8](#_ubmp8h7ex6ql)

[**REQUERIMIENTOS 9**](#_85erlkxo41g1)

[REQUERIMIENTOS FUNCIONALES 9](#_ikhrmq5b5561)

[REQUERIMIENTOS NO FUNCIONALES 12](#_wbc6msbu2zu0)

[ALCANCES Y LÍMITES 13](#_wa38pd2427m1)

[**ANÁLISIS DE REQUERIMIENTOS 15**](#_63766m9oc0m2)

[IDENTIFICCIÓN DE ACTORES 15](#_k1xe1j9vtle)

[CASOS DE USOS 15](#_7zi0bz709gux)

[**CU01 - Registrar Pedido 16**](#_e65k0nljhpr)

[ESPECIFICACIÓN DEL CASO DE USO 16](#_n650vzml65xn)

[CRITERIOS DE ACEPTACIÓN 17](#_36ruqav52n02)

[**CU02 – Asignar Operario a Pedido 18**](#_w7hna6uoznfs)

[ESPECIFICACIÓN DEL CASO DE USO 18](#_w2ywuk9j4les)

[CRITERIOS DE ACEPTACIÓN 19](#_dnyew8h8gu1n)

[**CU03 – Actualizar Estado de Pedido 20**](#_jfv4a6p107r2)

[ESPECIFICACIÓN DEL CASO DE USO 20](#_np78gg3nf0ke)

[CRITERIOS DE ACEPTACIÓN 21](#_ocrpas7sbiif)

[**CU04 – Generar Alertas de Vencimiento 24**](#_mbsogh745dm4)

[ESPECIFICACIÓN DEL CASO DE USO 24](#_nvdjvy5sma8d)

[CRITERIOS DE ACEPTACIÓN 25](#_sjrn12genw1c)

[**CU05 – Confirmar Entrega y Generar Boleta 26**](#_z2ftraer66vp)

[ESPECIFICACIÓN DEL CASO DE USO 26](#_unjaxslzo97p)

[CRITERIOS DE ACEPTACIÓN 27](#_ifltbq8kvbj0)

[**CU06 – Consultar Pedidos 29**](#_du01xs280pu1)

[ESPECIFICACIÓN DEL CASO DE USO 29](#_jzueehoqlock)

[CRITERIOS DE ACEPTACIÓN 29](#_317schscam2g)

[**MODELO DE BASE DE DATOS 32**](#_64drk7ogh0uu)

[MODELO RELACIONAL 32](#_zc8imojitt9)

[**IMPLEMENTACIÓN DE LA BASE DE DATOS 33**](#_rrpsmxejxfvt)

[CREACIÓN DE LA BASE DE DATOS 33](#_7r6e1zhkpd7b)

[CREACIÓN DE LOS OBJETOS DE LA BASE DE DATOS 35](#_8bakxbilo4mq)

[**CONCLUSIONES 37**](#_cyfyvzy3d6rg)

[**LECCIONES APRENDIDAS 39**](#_9m755c3743af)

[**BIBLIOGRAFÍA 40**](#_vn80n13a6erp)

# RESUMEN

El presente informe describe el desarrollo del módulo backend del “Sistema de control de pedidos en un taller de manufactura” denominado Mecafab. Este módulo, implementado con Spring Boot y SQL Server, abarca las siguientes funcionalidades principales:

1. Registro de pedidos: Permite ingresar nuevas órdenes con datos del cliente, descripción del producto, fechas de solicitud y compromiso, generando un código único de pedido.
2. Asignación de pedidos: El jefe de taller puede asignar cada pedido a un operario específico, actualizando el estado a “En Producción” y registrando la fecha de modificación.
3. Actualización de estado: Se registran los cambios de estado intermedios en la tabla de historial, evitando asignaciones redundantes y asegurando trazabilidad.
4. Alertas de vencimiento: Mediante consultas a la base de datos, el sistema identifica pedidos próximos a su fecha comprometida (falta ≤ 2 días) y los filtra para notificar al jefe de taller.
5. Confirmación de entrega y comprobantes: Al marcar un pedido como entregado, se actualiza su estado a “Entregado”, se almacenan las fechas real y confirmada, y se genera un comprobante de entrega.
6. Validaciones y manejo de errores: Se aplican restricciones de integridad en la base de datos (CHECK), validaciones de DTOs con Jakarta Bean Validation y manejo centralizado de excepciones REST.

# ABSTRACT

This document presents the design and implementation of the backend module for **Mecafab**, an order control system developed for a manufacturing workshop. Built with Spring Boot and SQL Server, the backend provides a RESTful API that supports: (1) creation of new orders with unique codes and client data; (2) assignment of orders to operators, updating their status to “In Production”; (3) recording of all intermediate state changes in a history table to ensure full traceability; (4) automated identification of orders nearing their committed delivery date (≤ 2 days) for proactive alerts; (5) confirmation of delivery with real delivery timestamps and generation of delivery receipts; and (6) comprehensive data-validation and business-rule enforcement via Bean Validation and custom exceptions. The database schema enforces referential integrity and domain constraints, while the service layer orchestrates transactional workflows. Centralized exception handling guarantees consistent JSON error responses. This robust, layered architecture replaces manual procedures, enhancing planning accuracy, operational efficiency, and auditability.

# INTRODUCCIÓN

El presente informe describe el desarrollo del módulo **backend** del sistema de control de pedidos para el taller de manufactura “Mecafab”. En el contexto actual, la gestión manual de órdenes —realizada mediante formularios de papel y un pizarrón en el taller— ha demostrado ser ineficiente: se pierden registros, los retrasos no se detectan a tiempo y la visibilidad del flujo de trabajo es limitada.

Con el objetivo de digitalizar y optimizar este proceso, se eligió la arquitectura de microservicios basada en **Spring Boot** para exponer una API RESTful robusta, y **SQL Server** como sistema de gestión de la base de datos. Esta combinación permite separar claramente las capas de presentación, negocio y datos, facilitando la mantenibilidad, escalabilidad y seguridad de la aplicación.

La sección de Backend implementa las siguientes funcionalidades clave:

1. **Registro de pedidos**: generación de un identificador único y almacenamiento de datos del cliente, descripción del producto, fechas de solicitud y entrega comprometida.
2. **Asignación y cambio de estado**: asignación de operarios, registro de cada transición de estado en un histórico y validación para evitar cambios redundantes.
3. **Alertas de vencimiento**: consulta de pedidos con fecha comprometida próxima (≤ 2 días) para notificaciones proactivas.
4. **Confirmación de entrega**: registro de la fecha real y generación de comprobantes de entrega, con validaciones de integridad de datos.
5. **Manejo de errores**: validaciones a nivel de DTO y base de datos, excepciones de negocio y control centralizado de errores REST.

Además, se detallan los requerimientos funcionales y no funcionales, la estructura de la base de datos (modelo relacional y diccionario de datos), los criterios de aceptación para cada caso de uso y los límites del sistema. Este enfoque integral garantiza una solución que mejora la planificación, reduce errores manuales y ofrece trazabilidad completa del ciclo de vida de cada pedido.

# OBJETIVOS

## OBJETIVO GENERAL

Desarrollar e implementar el módulo backend del sistema de control de pedidos **Mecafab**, basado en Spring Boot y SQL Server, que permita gestionar de forma digital todo el ciclo de vida de las órdenes —desde su creación hasta la entrega— garantizando integridad, trazabilidad y alertas proactivas.

## OBJETIVOS ESPECÍFICOS

* **Diseñar el modelo de datos** relacional que incluya tablas para clientes, pedidos, estados, historial de cambios y comprobantes de entrega, estableciendo claves primarias, foráneas y restricciones de integridad.
* **Implementar la creación de pedidos** mediante un endpoint REST que genere códigos únicos y almacene datos de cliente, descripción, fechas de solicitud y compromiso.
* **Desarrollar la asignación y actualización de estado** de pedidos, registrando cada transición en una tabla de historial para asegurar trazabilidad completa.
* **Configurar consultas de alerta** que identifiquen automáticamente los pedidos con fecha comprometida próxima (≤ 2 días) para notificaciones al jefe de taller.
* **Implementar la confirmación de entrega**, actualizando el estado a “Entregado”, registrando la fecha real y generando el comprobante de entrega de forma transaccional.
* **Aplicar validaciones y manejo de errores**, utilizando Jakarta Bean Validation en los DTOs, validaciones de base de datos (CHECK) y un controlador global de excepciones REST para respuestas JSON consistentes.
* **Documentar la API** y definir los criterios de aceptación para cada caso de uso, facilitando su mantenimiento y futura evolución.

# REQUERIMIENTOS

## REQUERIMIENTOS FUNCIONALES

**RF1 - Registrar Pedidos**

* **Actor principal**: Jefe de taller (usuario autenticado).
* **Descripción**: El sistema permite crear un nuevo pedido, generando automáticamente un código único con formato PED-YYYYMM-XXX. Se capturan: datos del cliente, descripción del producto, tipo de trabajo, fecha de solicitud y fecha de entrega comprometida.
* **Entradas**:
  + Identificador del cliente (ID).
  + Descripción del producto.
  + Tipo de trabajo.
  + Fecha de entrega comprometida.
* **Salida**:
  + Confirmación de creación con código de pedido.
* **Precondición**:
  + El usuario debe estar autenticado y tener rol “Jefe”.
  + El cliente debe existir en la base de datos.
* **Postcondición**:
  + Se inserta un registro en Pedido con estado\_id = 1 (Registrado).
  + Se registra la fecha de creación (fecha\_registro) y el historial inicial en HistorialEstadoPedido.

**RF2 – Asignar Pedidos**

* **Actor principal**: Jefe de taller.
* **Descripción**: Permite asignar un operario a un pedido existente, cambiando su estado a “En Producción” y registrando la fecha de modificación.
* **Entradas**:
  + Código o ID del pedido.
  + ID del operario a asignar.
* **Salida**:
  + Confirmación de la asignación y del nuevo estado.
* **Precondición**:
  + El pedido debe existir y estar en estado “Registrado”.
  + El operario debe existir y estar activo.
* **Postcondición**:
  + Se actualizan operario\_asignado\_id, estado\_id = 2 (En Producción) y fecha\_modificacion.
  + Se añade un registro en HistorialEstadoPedido con la transición.

**RF3 – Actualizar Estado del Pedido**

* **Actor principal**: Operario / Empleado.
* **Descripción**: Permite cambiar el estado de un pedido a cualquiera de los intermedios (“Listo para Entregar”, “En Espera de Material”, etc.), siempre que el nuevo estado difiere del actual, y registra cada cambio para trazabilidad.
* **Entradas**:
  + Código o ID del pedido.
  + Nuevo estado (ID de EstadoPedido).
* **Salida**:
  + Confirmación del cambio aplicado.
* **Precondición**:
  + El pedido debe existir.
  + El nuevo estado debe tener un orden\_logico superior al actual y no ser igual.
* **Postcondición**:
  + Se actualiza estado\_id, fecha\_modificacion.
  + Se inserta un registro en HistorialEstadoPedido.

**RF4 – Alertas de Vencimientos o Retrasos**

* **Actor principal**: Sistema (visualización por Jefe de taller).
* **Descripción**: El sistema identifica y lista los pedidos cuyo plazo de entrega comprometida vence en ≤ 2 días o ya está vencido, para permitir acciones preventivas.
* **Entradas**:
  + Parámetro de anticipación (por defecto, 2 días).
* **Salida**:
  + Listado de pedidos próximos a vencer o vencidos, con sus fechas y responsables.
* **Precondición**:
  + Existen pedidos en la base de datos con estado distinto de “Entregado” o “Cancelado”.
* **Postcondición**:
  + Se muestra un reporte/filter endpoint consumible por el frontend o consola.

**RF5 – Registrar Entrega Final y Generar Comprobante**

* **Actor principal:** Jefe de taller / Supervisor.
* **Descripción**: Permite confirmar la entrega final de un pedido. Al hacerlo, se registran las fechas de entrega real y confirmada, se marca el pedido como entregado y se genera un comprobante (guardado en ComprobanteEntrega).
* **Entradas**:
  + Código o ID del pedido.
* **Salida**:
  + Confirmación de entrega y contenido del comprobante.
* **Precondición**:
  + El pedido debe estar en estado “Listo para Entregar” y no estar ya marcado como entregado.
* **Postcondición**:
  + Se actualizan es\_entregado = 1, fecha\_entrega\_real, fecha\_entrega\_confirmada.
  + Se inserta un registro en ComprobanteEntrega.

## REQUERIMIENTOS NO FUNCIONALES

**Usabilidad**

El sistema debe contar con una **interfaz amigable y accesible** para usuarios que no son expertos en informática, como operarios o jefes de taller. Las pantallas deben ser simples e intuitivas: listas de pedidos con botones visibles para cambiar estados y formularios claros para registrar nuevos pedidos, reduciendo al mínimo la posibilidad de error humano.

**Eficiencia**

Dado que el taller maneja un volumen moderado de pedidos (decenas simultáneamente), el sistema debe **responder sin demoras perceptibles**. Las consultas, filtros y actualizaciones deben ejecutarse en tiempo real para apoyar la toma de decisiones del jefe de taller.

**Confiabilidad**

La aplicación debe garantizar la **consistencia y persistencia de los datos**, incluso ante cierres inesperados. Los pedidos no deben perderse ni corromperse. Se deben implementar **copias de respaldo automáticas** —por ejemplo, cada noche— para asegurar la recuperación ante fallos graves.

**Seguridad**

El sistema debe contar con **mecanismos de autenticación** y control de roles. Solo los supervisores podrán realizar acciones críticas como eliminar pedidos o marcar entregas finales, mientras que los operarios solo podrán modificar estados intermedios. Esto previene accesos no autorizados y mantiene la integridad de la información.

**Mantenibilidad**

El sistema debe ser fácil de modificar y escalar. Agregar nuevos estados de pedido o ajustar el flujo de trabajo no debe requerir reescribir el sistema desde cero. Se recomienda una arquitectura modular y una buena documentación del código para facilitar futuras mejoras.

## ALCANCES Y LÍMITES

**Alcance del módulo backend**

1. **Sin capa de presentación (frontend)**: El backend expone únicamente los endpoints REST para gestión de pedidos; no incluye interfaces de usuario web o móvil ni componentes UI, delegando esa responsabilidad a un cliente externo.
2. **Sin gestión de inventario y costos**: No contempla inventario de materiales, cálculo de costos de producción o control de stock; el alcance se limita al ciclo de vida de pedidos.
3. **Sin notificaciones automáticas**: No envía correos electrónicos, SMS ni notificaciones push de forma nativa; solo provee endpoints para consultar alertas de vencimiento que deben integrarse con servicios externos.
4. **Historial de auditoría limitado**: El único mecanismo de trazabilidad es la tabla HistorialEstadoPedido; no existe un registro de auditoría extendido para otras operaciones.
5. **Instancia única de base de datos**: Opera con una sola base de datos SQL Server sin soporte integrado de replicación, particionamiento o multi-tenant.

**Restricciones**

* **Uso de JdbcTemplate**: Se implementa con Spring Boot y JdbcTemplate, descartando frameworks ORM (JPA/Hibernate).
* **Almacenamiento de comprobantes**: Los comprobantes de entrega se guardan como texto codificado en la columna contenido de ComprobanteEntrega, sin integración con almacenamiento externo.
* **Dependencia de SQL Server**: La lógica SQL (funciones GETDATE(), DATEADD()) está pensada para SQL Server, por lo que no es compatible con otros motores sin ajustes.
* **Despliegue stateless**: Diseñado para ejecutarse como servicio stateless en contenedores (Docker/Kubernetes), sin depender de archivos locales.

# ANÁLISIS DE REQUERIMIENTOS

## IDENTIFICCIÓN DE ACTORES

| **CÓDIGO** | **NOMBRE** | **DESCRIPCIÓN** |
| --- | --- | --- |
| **ACT01** | Jefe de Taller | Usuario con rol “Jefe”. Registra pedidos, asigna operarios, confirma entregas y genera alertas. |
| **ACT02** | Operario | Usuario con rol “Operario”. Actualiza estados intermedios de los pedidos. |
| **ACT03** | Sistema | Actor “automático” que ejecuta la lógica de alertas de vencimiento y expone el endpoint correspondiente. |

## CASOS DE USOS

| **CÓDIGO** | **ACTOR** | **NOMBRE** | **DESCRIPCIÓN** |
| --- | --- | --- | --- |
| **CU01** | Jefe de Taller | Registrar Pedido | Crea un nuevo pedido con código único y datos de cliente, describe el producto y fechas de solicitud/entrega. |
| **CU02** | Jefe de Taller | Asignar Operario a Pedido | Asigna un operario a un pedido existente, cambia estado a “En Producción” y registra la fecha de modificación. |
| **CU03** | Operario | Actualizar Estado de Pedido | Marca un pedido con cualquiera de los estados intermedios (En Proceso, En Espera, Listo para Entregar, etc.) |
| **CU04** | Sistema | Generar Alertas de Vencimiento | Identifica pedidos con fecha comprometida a ≤ 2 días o vencidos y los expone en un endpoint de alertas. |
| **CU05** | Jefe de Taller | Confirmar Entrega y Generar Boleta | Cambia de estado a “Entregado”, registra fecha real y genera comprobante de entrega. |
| **CU06** | Jefe de Taller | Consultar Pedidos | Filtra y lista pedidos por estado, operario, cliente o rango de fechas (endpoint REST consumible). |

## 

## 

# CU01 - Registrar Pedido

## ESPECIFICACIÓN DEL CASO DE USO

| **Código** | **CU01** | |
| --- | --- | --- |
| **Nombre** | Registrar Pedido | |
| **Descripción** | Permite al Jefe de Taller crear un nuevo pedido en MECAFAB, generando automáticamente un código único, validando los datos del cliente y de las fechas, y almacenando la orden con su estado inicial. | |
| **Actores** | **Jefe de taller** | |
| **Precondición** | * El Jefe debe estar autenticado con rol “Jefe”. * El cliente indicado debe existir en la base de datos. * La fecha\_entrega\_comprometida debe ser posterior o igual a la fecha\_solicitud. | |
| **Secuencia normal** | **Paso** | **Acción** |
| **1** | El Jefe envía **POST /api/pedidos** con un PedidoRequestDTO (clienteId, descripciónProducto, tipoTrabajo, fechaSolicitud, fechaEntregaComprometida). |
| **2** | El sistema valida la existencia del cliente; si no existe, lanza ClienteNoEncontradoException (404). |
| **3** | El sistema genera automáticamente el código del pedido con formato PED-YYYYMM-XXX. |
| **4** | El sistema inserta el registro en Pedido con estado\_id = 1 (Registrado). |
| **5** | El sistema registra un evento en HistorialEstadoPedido indicando la transición a “Registrado”. |
| **6** | El sistema retorna **200 OK** con mensaje “Pedido registrado correctamente.” |
| **Postcondición** | * El pedido queda almacenado con estado\_id = 1 (“Registrado”) y fecha\_registro = GETDATE(). * Existe un historial inicial en HistorialEstadoPedido. | |
| **Excepciones** | **Paso** | **Acción** |
| **1** | ClienteNoEncontradoException (404): “No se encontró el cliente con ID: X.” |
| **2** | Error de base de datos al insertar el pedido (500). |
| **Comentarios** | — | |

## CRITERIOS DE ACEPTACIÓN

#### 1. Validación de Datos de Entrada

* Todos los campos de PedidoRequestDTO (clienteId, descripcionProducto, tipoTrabajo, fechaSolicitud, fechaEntregaComprometida) deben cumplir sus anotaciones de Bean Validation (@NotBlank, @NotNull, @FutureOrPresent, @Future).
* Si falla, responde **400 Bad Request** con JSON de campos y mensajes de error.

#### 2. Generación de código único

* El servidor crea un código alfanumérico PED-YYYYMM-XXX y lo guarda en Pedido.codigo.

#### 3. Inserción del pedido

* Se ejecuta el INSERT en Pedido con estado\_id = 1, fecha\_registro = GETDATE(), es\_entregado = 0.
* La operación debe afectar exactamente 1 fila.

#### 4. Registro en el historial de estados

* Tras la inserción, se inserta en HistorialEstadoPedido un registro con (pedido\_id, estado\_anterior\_id = NULL, estado\_nuevo\_id = 1, usuario\_id, fecha\_cambio = GETDATE()).
* Debe formar parte de la misma transacción; si falla, la creación del pedido se revierte.

#### 5. Respuesta de éxito

* El endpoint retorna **200 OK** (o **201 Created** si se prefiere) con el mensaje:

{ "mensaje": "Pedido registrado correctamente", "codigo": "PED-YYYYMM-XXX" }

#### 6. Manejo de errores y formato de respuesta

* Toda excepción debe mapearse a un JSON con { timestamp, codigo, mensaje }, sin exponer stack trace.

# CU02 – Asignar Operario a Pedido

## ESPECIFICACIÓN DEL CASO DE USO

| **Código** | **CU03** | |
| --- | --- | --- |
| **Nombre** | **Asignar Operario a Pedido** | |
| **Descripción** | Permite al Jefe de Taller asignar un operario a un pedido existente y actualizar su estado, registrando la fecha de modificación correspondiente. | |
| **Actores** | **Jefe de Taller** | |
| **Precondición** | * Usuario autenticado con rol “Jefe”. * El pedido debe existir y estar en estado “Registrado”. * El operario debe existir y estar activo. | |
| **Secuencia normal** | **Paso** | **Acción** |
| **1** | El Jefe envía **PUT /api/pedidos/asignar** con un AsignacionPedidoDTO (pedidoId, operarioId, nuevoEstadoId). |
| **2** | El sistema valida que el pedido exista; si no, lanza PedidoNoEncontradoException (404). |
| **3** | Valida que el operario exista; si no, lanza OperarioNoEncontradoException (404). |
| **4** | Actualiza en Pedido los campos operario\_asignado\_id, estado\_id y fecha\_modificacion. |
| **5** | Inserta un registro en HistorialEstadoPedido con la transición realizada. |
| **6** | Retorna **200 OK**: “Pedido asignado y estado actualizado.” |
| **Postcondición** | * El pedido tiene asignado el operario y un nuevo estado. * Se registra la modificación en el historial de estados. | |
| **Excepciones** | **Paso** | **Acción** |
| **1** | PedidoNoEncontradoException (404): “No se encontró el pedido con ID: X.” |
| **2** | OperarioNoEncontradoException (404): “No se encontró el operario con ID: Y.” |
| **3** | Error de base de datos (500). |
| **Comentarios** | * — * — | |

## 

## CRITERIOS DE ACEPTACIÓN

#### 1. Validación de Datos de Entrada

* Todos los campos de AsignacionPedidoDTO (pedidoId, operarioId, nuevoEstadoId) deben estar presentes y no ser nulos (@NotNull).
* En caso de validación fallida, el endpoint devuelve **400 Bad Request** con un JSON de errores conforme al manejador global de excepciones.

#### 2. Validación de existencia del pedido

* Antes de actualizar, el sistema verifica que exista un registro en Pedido con **id = pedidoId**.
* Si no existe, lanza PedidoNoEncontradoException y responde **404 Not Found** con:

{ "timestamp": "...", "codigo": "NOT\_FOUND", "mensaje": "No se encontró el pedido con ID: X." }

#### 3. Validación de existencia del operario

* Verifica que exista un registro en Usuario con **id = operarioId** y rol “Operario”.
* Si no existe, lanza OperarioNoEncontradoException y responde **404 Not Found** con un mensaje análogo.

#### 4. Actualización en la base de datos

* Ejecuta un UPDATE Pedido que establezca operario\_asignado\_id, estado\_id = nuevoEstadoId y fecha\_modificacion = GETDATE().
* La operación debe afectar exactamente 1 fila; de lo contrario, se considera error de integridad.

#### 5. Registro del historial de estado

* Inserta en HistorialEstadoPedido un nuevo registro con los campos (pedido\_id, estado\_anterior\_id, estado\_nuevo\_id, usuario\_id, fecha\_cambio = GETDATE()).
* Esta inserción forma parte de la misma transacción; si falla, todo el cambio debe revertirse.

#### 6. Manejo de errores y formato de respuestas

* Todas las respuestas de error deben seguir el esquema JSON: { timestamp, codigo, mensaje }.
* No exponer stack traces ni detalles internos en la respuesta.

# CU03 – Actualizar Estado de Pedido

## ESPECIFICACIÓN DEL CASO DE USO

| **Código** | **CU03** | |
| --- | --- | --- |
| **Nombre** | **Actualizar Estado de Pedido** | |
| **Descripción** | Permite a un operario cambiar el estado de un pedido a cualquiera de los intermedios (por ejemplo, “En proceso”, “En espera de material”, “Listo para Entregar”), siempre que el nuevo estado sea distinto al actual y su orden lógico sea superior, garantizando trazabilidad y evitando retrocesos. | |
| **Actores** | **Operario** | |
| **Precondición** | * Usuario autenticado con rol “Operario”. * El pedido con ID existe en la base de datos. * El estado actual del pedido no es final. | |
| **Secuencia normal** | **Paso** | **Acción** |
| **1** | El operario envía PUT /api/pedidos/cambiar-estado con CambioEstadoDTO (pedidoId, nuevoEstadoId). |
| **2** | El sistema valida que el pedido exista; si no, lanza PedidoNoEncontradoException. |
| **3** | Verifica que nuevoEstadoId ≠ estado actual; si coincide, EstadoYaAsignadoException. |
| **4** | Comprueba que el ordenLogico del nuevo estado sea mayor; si no, ReglaNegocioException. |
| **5** | Actualiza estado\_id y fecha\_modificacion en Pedido. |
| **6** | Inserta un registro en HistorialEstadoPedido. |
| **7** | Retorna 200 OK: “Estado actualizado correctamente.” |
| **Postcondición** | Pedido con el nuevo estado y su historial actualizado. | |
| **Excepciones** | **Paso** | **Acción** |
| **1** | PedidoNoEncontradoException (404): “No se encontró el pedido con ID X”. |
| **2** | EstadoYaAsignadoException (400): “El estado ya está aplicado al pedido”. |
| **3** | ReglaNegocioException (400): “El nuevo estado no es válido según el flujo”. |
| **4** | Error de base de datos (500). |
| **Comentarios** | * — * — | |

## 

## CRITERIOS DE ACEPTACIÓN

#### 1. Validación de Datos de Entrada

**Descripción:**

El sistema debe verificar que el DTO CambioEstadoDTO contenga todos los campos obligatorios (pedidoId, nuevoEstadoId, usuarioResponsableId) y que cumplan las anotaciones de Bean Validation.

**Criterios:**

* pedidoId, nuevoEstadoId y usuarioResponsableId no pueden ser nulos (@NotNull).
* Si falla la validación, el endpoint responde **400 Bad Request** con un JSON de errores:

{ "timestamp": "...", "codigo": "VALIDATION\_ERROR", "mensaje": "campo: mensaje" }

#### 2. Validación de Pedido Existente

**Descripción:**

Antes de procesar el cambio de estado, el sistema debe confirmar que existe un registro en la tabla Pedido con el pedidoId recibido.

**Criterios:**

* Si no existe, lanzar PedidoNoEncontradoException.
* La respuesta será **404 Not Found** con cuerpo:

{ "timestamp": "...", "codigo": "NOT\_FOUND", "mensaje": "No se encontró el pedido con ID: X." }

#### 3. Validación de Estado Nuevo

**Descripción:**

El sistema debe comprobar que el nuevoEstadoId recibido es distinto del estado actual y que su ordenLogico es superior, evitando retrocesos o reasignaciones idénticas.

**Criterios:**

* Si nuevoEstadoId == estadoActual, lanzar EstadoYaAsignadoException → **400 Bad Request** con:

“El estado que se intenta asignar ya está aplicado al pedido.”

* Si ordenLogico(nuevo) <= ordenLogico(actual), lanzar ReglaNegocioException → **400 Bad Request** con:

“El nuevo estado no es válido según el flujo lógico.”

#### 4. Actualización de Estado en la Base de Datos

**Descripción:**

Una vez validados, el sistema actualiza el registro Pedido estableciendo el nuevo estado\_id y la columna fecha\_modificacion.

**Criterios:**

* Se ejecuta el UPDATE Pedido SET estado\_id = ?, fecha\_modificacion = GETDATE() WHERE id = ? y afecta exactamente 1 fila.

“El estado que se intenta asignar ya está aplicado al pedido.”

* El endpoint devuelve **200 OK** con mensaje:

“Estado actualizado correctamente.”

#### 5. Registro en Historial de Estados

**Descripción:**

Cada cambio de estado debe quedar trazado en la tabla HistorialEstadoPedido.

**Criterios:**

* Insertar un nuevo registro con los campos pedido\_id, estado\_anterior\_id, estado\_nuevo\_id, usuario\_id y fecha\_cambio = GETDATE().
* La operación de historial debe ser parte de la misma transacción; si falla, se revierte todo el cambio de estado.

#### 6. Manejo de Errores y Excepciones

**Descripción:**

El backend debe capturar y formatear todas las excepciones para no exponer detalles internos.

**Criterios:**

* Respuestas JSON con campos { timestamp, codigo, mensaje }.
* No exponer stack trace ni excepciones de Java.
* Ejemplos de mensaje:
  + “No se encontró el pedido con ID: X.”
  + “El estado que se intenta asignar ya está aplicado al pedido.”
  + “El nuevo estado no es válido según el flujo lógico.”
  + “Error interno del servidor” (para excepciones genéricas).

# CU04 – Generar Alertas de Vencimiento

## ESPECIFICACIÓN DEL CASO DE USO

| **Código** | **CU04** | |
| --- | --- | --- |
| **Nombre** | **Generar Alertas de Vencimiento** | |
| **Descripción** | Permite al sistema identificar y listar los pedidos cuyo plazo de entrega comprometida vence en ≤ N días o ya está vencido, para notificar proactivamente al Jefe de Taller. | |
| **Actores** | **Sistema (proceso automático)** | |
| **Precondición** | * Existen pedidos en la base de datos con es\_entregado = 0. * Al menos un pedido tiene fecha\_entrega\_comprometida dentro del rango solicitado. | |
| **Secuencia normal** | **Paso** | **Acción** |
| **1** | El sistema invoca **GET /api/pedidos/vencimientos?dias={dias}** (por defecto 2). |
| **2** | El controlador extrae y valida el parámetro dias. |
| **3** | El servicio llama a buscarPedidosPorVencer(dias) en el repositorio. |
| **4** | El repositorio ejecuta la consulta sql. |
| **7** | Retorna la lista de pedidos que vencerán en N días o ya vencieron. si no hay pedido que vaya a vencer en N días o no ha vencido aún, retornara null. |
| **Postcondición** | * Se entrega un listado JSON con los pedidos filtrados. * No se modifica ningún registro en la base de datos. | |
| **Excepciones** | **Paso** | **Acción** |
| **1** | Parámetro `dias` inválido (≤ 0 o no numérico) → 400 Bad Request |
| **2** | Error de base de datos → 500 Internal Server Error |
| **Comentarios** |  | |

## 

## CRITERIOS DE ACEPTACIÓN

#### 1. Parámetro de entrada

* dias debe ser ≥ 0.
* Si no se envía, toma el valor por defecto 2.
* Error 400 con JSON de validación si dias < 0.

#### 2. Filtrado correcto

* Sólo se incluyen pedidos con es\_entregado = 0.
* fecha\_entrega\_comprometida debe caer entre hoy y hoy + dias días.
* El SQL debe coincidir con la lógica.

#### 3. Estructura de la respuesta

* 200 OK con JSON:

[

{

"id": 5,

"codigo": "PED-ABCD1234",

"descripcionProducto": "...",

"fecha\_entrega\_comprometida": "2025-06-14",

"operarioAsignadoId": 2,

…

},

…

]

* Cada objeto Pedido incluye al menos: id, codigo, descripcionProducto, fecha\_entrega\_comprometida, operarioAsignadoId, estadoId.

#### 4. Rendimiento

* La consulta debe ejecutarse en tiempo real (< 200 ms) para un volumen moderado de pedidos (decenas).

#### 5. Manejo de errores

* Cualquier excepción no prevista devuelve 500 Internal Server Error con JSON:

{ "timestamp":"...", "codigo":"INTERNAL\_ERROR", "mensaje":"Error interno del servidor." }

* No exponer detalles internos o stack traces en la respuesta.

# CU05 – Confirmar Entrega y Generar Boleta

## ESPECIFICACIÓN DEL CASO DE USO

| **Código** | **CU05** | |
| --- | --- | --- |
| **Nombre** | **Confirmar Entrega y Generar Comprobante** | |
| **Descripción** | Permite al Jefe de Taller confirmar la entrega de un pedido, registrando la fecha real de entrega y generando un comprobante de forma transaccional. | |
| **Actores** | **Jefe de Taller / Supervisor** | |
| **Precondición** | * El pedido (Pedido.id) debe existir y es\_entregado = 0. * El usuario responsable (usuarioResponsableId) debe existir. * El estado actual no debe ser “Entregado”. | |
| **Secuencia normal** | **Paso** | **Acción** |
| **1** | El Jefe envía **PUT /api/pedidos/entregar** con un ConfirmacionEntregaDTO que incluye:  - pedidoId  - usuarioResponsableId.  - *Opcional*: fechaEntregaConfirmada, tipoComprobante, contenido. |
| **2** | El sistema valida el DTO (Bean Validation) y confirma que el pedido existe; si no, lanza PedidoNoEncontradoException (404). |
| **3** | Verifica que es\_entregado = 0; si ya fue entregado, lanza EntregaYaConfirmadaException (400). |
| **4** | Llama a pedidoRepository.actualizarEntrega(...) para:  - Cambiar estado\_id a “Entregado”  - Marcar es\_entregado = 1  - Establecer fecha\_entrega\_real = GETDATE() y fecha\_entrega\_confirmada |
| **5** | Llama a pedidoRepository.insertarComprobante(...) para añadir un registro en ComprobanteEntrega con el contenido y tipo proporcionados. |
| **6** | Retorna **200 OK** con mensaje: “Entrega registrada y comprobante generado.” |
| **Postcondición** | * El pedido queda con es\_entregado = 1, estado\_id = Id de “Entregado”, y fechas de entrega registradas. * Se genera una fila en ComprobanteEntrega con el comprobante asociado. | |
| **Excepciones** | **Paso** | **Acción** |
| **1** | PedidoNoEncontradoException (404): “No se encontró el pedido con ID: X.” |
| **2** | EntregaYaConfirmadaException (400): “La entrega del pedido con ID X ya fue confirmada.” |
| **3** | Error de base de datos (500). |
| **Comentarios** |  | |

## 

## CRITERIOS DE ACEPTACIÓN

#### 1. Validación de entrada

* pedidoId y usuarioResponsableId no pueden ser nulos (@NotNull).
* Campos opcionales (fechaEntregaConfirmada, tipoComprobante, contenido) si se envían deben tener formato válido.
* Si falla Bean Validation, retorna **400 Bad Request** con JSON de errores.

#### 2. Existencia del pedido

* Se comprueba que exista un registro en Pedido con id = pedidoId.
* Si no existe, lanza PedidoNoEncontradoException y responde **404 Not Found**.

#### 3. Chequeo de estado

* Si es\_entregado = 1, se lanza EntregaYaConfirmadaException y retorna **400 Bad Request**.

#### 4. Actualización transaccional

* actualizarEntrega(...) debe actualizar una única fila de Pedido con:
  + estado\_id = Id de “Entregado”
  + es\_entregado = 1
  + fecha\_entrega\_real = GETDATE()
  + fecha\_entrega\_confirmada = valor enviado
* La operación forma parte de una transacción; si falla, se revierte todo.

#### 5. Generación de comprobante

* insertarComprobante(...) inserta en ComprobanteEntrega con:
  + pedido\_id, contenido, tipo, usuario\_id, fecha\_generacion = GETDATE()
* Debe insertarse exactamente 1 nuevo registro dentro de la misma transacción.

#### 6. Respuesta de éxito

* Retorna **200 OK** con JSON:

{ "mensaje": "Entrega registrada y comprobante generado." }

#### 7. Respuesta de éxito

* Cualquier excepción inesperada responde **500 Internal Server Error** con JSON:

{ "timestamp": "...", "codigo": "INTERNAL\_ERROR", "mensaje": "Error interno del servidor." }

* No exponer stack trace ni detalles internos.

# CU06 – Consultar Pedidos

## ESPECIFICACIÓN DEL CASO DE USO

## 

| **Código** | **CU06** | |
| --- | --- | --- |
| **Nombre** | **Consultar Pedidos** | |
| **Descripción** | Permite al Jefe de Taller o a un sistema cliente filtrar y listar pedidos según estado, operario, cliente o rango de fechas, devolviendo un JSON con los pedidos que cumplan los criterios. | |
| **Actores** | **Jefe de Taller** | |
| **Precondición** | * Usuario autenticado. * Al menos uno de los parámetros (estadoId, operarioId, clienteId o rango de fechas) puede estar presente. | |
| **Secuencia normal** | **Paso** | **Acción** |
| **1** | El cliente invoca **GET /api/pedidos** con opcionales: estadoId, operarioId, clienteId. |
| **2** | El controlador delega a pedidoService.listarPedidos(estadoId, operarioId, clienteId). |
| **3** | El servicio llama a pedidoRepository.buscarPedidos(...), que construye dinámicamente el SQL según parámetros no nulos. |
| **4** | Se ejecuta la consulta y se obtienen los Pedido que coinciden. |
| **5** | El endpoint retorna **200 OK** con un arreglo JSON de objetos Pedido. |
| **Postcondición** | * Se entrega al cliente un listado de pedidos filtrados en JSON. * No se modifica ningún registro en la base de datos. | |
| **Excepciones** | **Paso** | **Acción** |
| **1** | **400 Bad Request** si algún parámetro no es numérico o inválido. |
| **2** | **500 Internal Server Error** si falla la consulta en base de datos. |
| **Comentarios** |  | |

## 

## CRITERIOS DE ACEPTACIÓN

#### 1. Validación de parámetros

* estadoId, operarioId y clienteId, si se envían, deben ser enteros válidos (≥ 1).
* Parámetro inválido → 400 Bad Request con JSON:

{ "timestamp":"…", "codigo":"VALIDATION\_ERROR", "mensaje":"Parámetro 'operarioId' inválido" }

#### 2. Filtrado correcto

* Si estadoId no es nulo, sólo incluir filas con Pedido.estado\_id = estadoId.
* Análogo para operarioId → operario\_asignado\_id, y clienteId → cliente\_id.
* Si no se envía ningún filtro, devolver todos los pedidos.

#### 3. Estructura de la respuesta

* **200 OK** con un array JSON:

[

{

"id": 7,

"codigo": "PED-1A2B3C4D",

"descripcionProducto": "...",

"tipoTrabajo": "...",

"fechaSolicitud": "2025-06-10",

"fechaEntregaComprometida": "2025-06-15",

"estadoId": 2,

"clienteId": 1,

"operarioAsignadoId": 3,

…

},

…

]

* Incluir al menos: id, codigo, descripcionProducto, fechaEntregaComprometida, estadoId, operarioAsignadoId, clienteId.

#### 4. Rendimiento

* La consulta debe ejecutarse en menos de 200 ms para un volumen moderado (decenas de pedidos).

#### 5. Manejo de errores

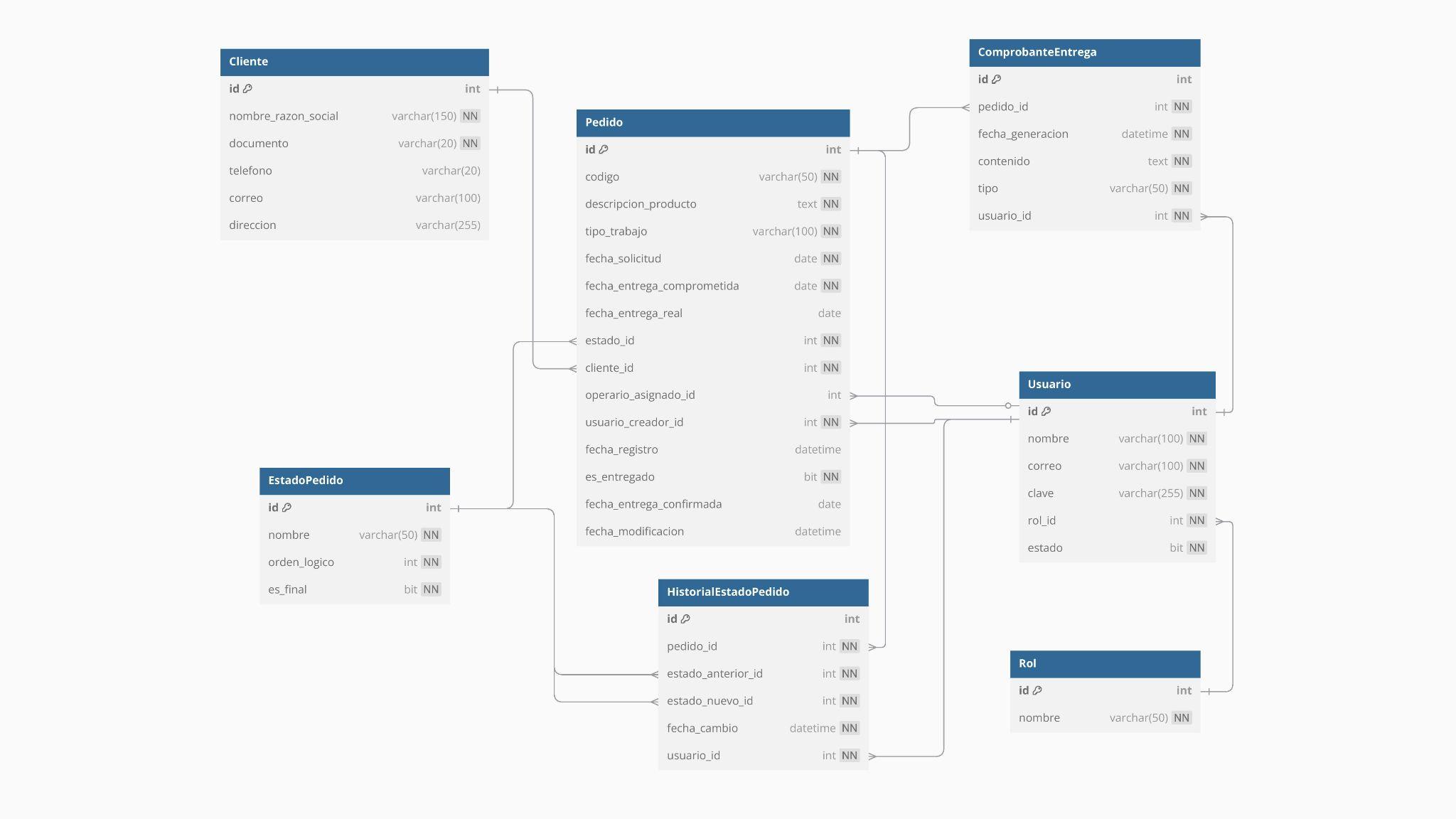
* Cualquier excepción no prevista retorna **500 Internal Server Error** con JSON:

{ "timestamp":"…", "codigo":"INTERNAL\_ERROR", "mensaje":"Error interno del servidor." }

* No exponer stack traces ni detalles de implementación.

# MODELO DE BASE DE DATOS

## MODELO RELACIONAL



# IMPLEMENTACIÓN DE LA BASE DE DATOS

## CREACIÓN DE LA BASE DE DATOS

| CREATE DATABASE MecafabFinal; USE MecafabFinal;  CREATE TABLE Rol (  id INT PRIMARY KEY IDENTITY(1,1),  nombre VARCHAR(50) NOT NULL );  CREATE TABLE Usuario (  id INT PRIMARY KEY IDENTITY(1,1),  nombre VARCHAR(100) NOT NULL,  correo VARCHAR(100) UNIQUE NOT NULL,  clave VARCHAR(255) NOT NULL,  rol\_id INT NOT NULL,  estado BIT NOT NULL,  FOREIGN KEY (rol\_id) REFERENCES Rol(id) );  CREATE TABLE Cliente (  id INT PRIMARY KEY IDENTITY(1,1),  nombre\_razon\_social VARCHAR(150) NOT NULL,  documento VARCHAR(20) NOT NULL,  telefono VARCHAR(20),  correo VARCHAR(100),  direccion VARCHAR(255) );  CREATE TABLE EstadoPedido (  id INT PRIMARY KEY IDENTITY(1,1),  nombre VARCHAR(50) NOT NULL,  orden\_logico INT NOT NULL,  es\_final BIT NOT NULL );  CREATE TABLE Pedido (  id INT PRIMARY KEY IDENTITY(1,1),  codigo VARCHAR(50) UNIQUE NOT NULL,  descripcion\_producto TEXT NOT NULL,  tipo\_trabajo VARCHAR(100) NOT NULL,  fecha\_solicitud DATE NOT NULL,  fecha\_entrega\_comprometida DATE NOT NULL,  fecha\_entrega\_real DATE,  estado\_id INT NOT NULL,  cliente\_id INT NOT NULL,  operario\_asignado\_id INT,  usuario\_creador\_id INT NOT NULL,  fecha\_registro DATETIME DEFAULT GETDATE(),  es\_entregado BIT NOT NULL DEFAULT 0,  fecha\_entrega\_confirmada DATE,  fecha\_modificacion DATETIME,  FOREIGN KEY (estado\_id) REFERENCES EstadoPedido(id),  FOREIGN KEY (cliente\_id) REFERENCES Cliente(id),  FOREIGN KEY (operario\_asignado\_id) REFERENCES Usuario(id),  FOREIGN KEY (usuario\_creador\_id) REFERENCES Usuario(id) );  CREATE TABLE HistorialEstadoPedido (  id INT PRIMARY KEY IDENTITY(1,1),  pedido\_id INT NOT NULL,  estado\_anterior\_id INT NOT NULL,  estado\_nuevo\_id INT NOT NULL,  fecha\_cambio DATETIME NOT NULL DEFAULT GETDATE(),  usuario\_id INT NOT NULL,  FOREIGN KEY (pedido\_id) REFERENCES Pedido(id),  FOREIGN KEY (estado\_anterior\_id) REFERENCES EstadoPedido(id),  FOREIGN KEY (estado\_nuevo\_id) REFERENCES EstadoPedido(id),  FOREIGN KEY (usuario\_id) REFERENCES Usuario(id) );  CREATE TABLE ComprobanteEntrega (  id INT PRIMARY KEY IDENTITY(1,1),  pedido\_id INT NOT NULL,  fecha\_generacion DATETIME NOT NULL DEFAULT GETDATE(),  contenido TEXT NOT NULL,  tipo VARCHAR(50) NOT NULL,  usuario\_id INT NOT NULL,  FOREIGN KEY (pedido\_id) REFERENCES Pedido(id),  FOREIGN KEY (usuario\_id) REFERENCES Usuario(id) );  -- Validaciones CHECK para fechas lógicas ALTER TABLE Pedido ADD CONSTRAINT chk\_fecha\_entrega\_real CHECK (  fecha\_entrega\_real IS NULL OR fecha\_entrega\_real >= fecha\_solicitud );  ALTER TABLE Pedido ADD CONSTRAINT chk\_fecha\_confirmada CHECK (  fecha\_entrega\_confirmada IS NULL OR fecha\_entrega\_confirmada >= fecha\_solicitud ); |
| --- |

## CREACIÓN DE LOS OBJETOS DE LA BASE DE DATOS

| USE MecafabFinal;  -- Insertar roles INSERT INTO Rol (nombre) VALUES ('Jefe'), ('Operario');  -- Insertar usuarios (1 jefe, 2 operarios) INSERT INTO Usuario (nombre, correo, clave, rol\_id, estado) VALUES ('Jefe Maestro', 'jefe@uni.edu.pe', 'clave123', 1, 1), ('Operario 1', 'op1@uni.edu.pe', 'clave123', 2, 1), ('Operario 2', 'op2@uni.edu.pe', 'clave123', 2, 1);  -- Insertar clientes INSERT INTO Cliente (nombre\_razon\_social, documento, telefono, correo, direccion) VALUES ('Cliente Alpha S.A.', '20511111111', '987654321', 'alpha@cliente.com', 'Av. Lima 123'), ('Cliente Beta SAC', '20522222222', '912345678', 'beta@cliente.com', 'Jr. Cusco 456');  -- Insertar estados INSERT INTO EstadoPedido (nombre, orden\_logico, es\_final) VALUES ('Registrado', 1, 0), ('En Producción', 2, 0), ('Listo para Entregar', 3, 0), ('Entregado', 4, 1), ('Cancelado', 5, 1);  -- Insertar pedidos INSERT INTO Pedido (codigo, descripcion\_producto, tipo\_trabajo, fecha\_solicitud, fecha\_entrega\_comprometida, estado\_id, cliente\_id, operario\_asignado\_id, usuario\_creador\_id, fecha\_registro, es\_entregado, fecha\_entrega\_confirmada, fecha\_entrega\_real) VALUES -- Pedido sin operario ni entrega aún (para probar registro y asignación) ('PED-0001', 'Pieza metálica A', 'Torno CNC', GETDATE(), DATEADD(DAY, 4, GETDATE()), 1, 1, NULL, 1, GETDATE(), 0, NULL, NULL), -- Pedido asignado a operario y en producción (para probar cambio de estado) ('PED-0002', 'Soporte estructural', 'Fresadora', GETDATE(), DATEADD(DAY, 2, GETDATE()), 2, 2, 2, 1, GETDATE(), 0, NULL, NULL), -- Pedido ya entregado (para probar filtros y comprobante) ('PED-0003', 'Eje de acero', 'Corte láser', GETDATE(), DATEADD(DAY, -1, GETDATE()), 4, 1, 2, 1, GETDATE(), 1, GETDATE(), GETDATE());  -- Insertar historial de cambios para pedidos INSERT INTO HistorialEstadoPedido (pedido\_id, estado\_anterior\_id, estado\_nuevo\_id, usuario\_id, fecha\_cambio) VALUES -- Pedido 2: registrado -> producción (2, 1, 2, 1, GETDATE()), -- Pedido 3: registrado -> producción -> listo -> entregado (3, 1, 2, 1, DATEADD(DAY, -3, GETDATE())), (3, 2, 3, 1, DATEADD(DAY, -2, GETDATE())), (3, 3, 4, 1, DATEADD(DAY, -1, GETDATE()));  -- Insertar comprobante de entrega INSERT INTO ComprobanteEntrega (pedido\_id, contenido, tipo, usuario\_id, fecha\_generacion) VALUES (3, 'BASE64-COMPROBANTE-MOCK-ENTREGA', 'PDF', 1, GETDATE()); |
| --- |

## 

# CONCLUSIONES

**Cumplimiento de objetivos**

El módulo backend del sistema MECAFAB ha logrado cubrir íntegramente los objetivos planteados: digitalizar el ciclo de vida de un pedido desde su creación hasta su entrega, garantizando integridad, trazabilidad y alertas proactivas.

**Implementación de funcionalidades clave**

Se desarrollaron con éxito los servicios REST para:

* Registro de pedidos con generación de código único.
* Asignación de operarios y cambio de estado con historial de transiciones.
* Detección automática de pedidos próximos a vencer.
* Confirmación de entrega y emisión de comprobantes.

**Arquitectura y calidad del código**

La solución se estructuró en capas (controller, service, repository) aprovechando Spring Boot y JdbcTemplate, lo que facilita la mantenibilidad y la escalabilidad. El uso de Bean Validation y un manejador global de excepciones asegura respuestas JSON coherentes y control de errores uniforme.

**Integridad y rendimiento**

* Las validaciones de DTO y las restricciones CHECK en la base de datos garantizan la consistencia de la información.
* Las consultas dinámicas para filtrado y alertas están optimizadas para responder en tiempo real con volúmenes moderados de pedidos.

**Limitaciones y mejoras futuras**

* **Interfaz de usuario**: Falta un frontend (web o móvil) que permita interactuar de forma amigable con los endpoints.
* **Notificaciones**: No se integraron servicios de correo, SMS o push para alertas automáticas.
* **Seguridad avanzada**: Aunque se definieron roles, falta implementar autenticación (por ejemplo, JWT) y autorización granular.
* **Reportes y dashboards**: No se creó un mecanismo de visualización analítica de datos (gráficos, estadísticas).

**Direcciones de evolución**

Para la siguiente fase, se recomienda:

* Desarrollar un cliente web (React, Angular) que consuma la API REST.
* Incorporar un sistema de notificaciones automatizadas.
* Añadir autenticación JWT y control de acceso por roles.
* Implementar microservicios adicionales o escalado horizontal en contenedores (Docker/Kubernetes).

# LECCIONES APRENDIDAS

**Importancia de los Requerimientos Iniciales**:

* La claridad en la recopilación y análisis de requerimientos es fundamental para el éxito del proyecto.

**Impacto de la Documentación**:

* La documentación técnica detallada ayuda a garantizar la mantenibilidad del sistema y facilita futuras mejoras.

**Gestión de Alcances**:

* Definir un alcance realista desde el inicio evita sobrecarga de trabajo y garantiza entregables de calidad.

**Validaciones y Seguridad**:

* Incorporar validaciones a nivel de base de datos y backend refuerza la seguridad y coherencia del sistema.

**Aprendizaje Colaborativo**:

* El proyecto destaca la importancia del trabajo en equipo y la distribución efectiva de tareas.

# BIBLIOGRAFÍA

Microsoft. (2024). *SQL Server Management Studio* (versión 20) [Herramienta de gestión de bases de datos]. Microsoft. <https://learn.microsoft.com/en-us/sql/ssms/>

Project Lombok. (s.f.). *Features* [Sitio web]. Recuperado el 12 de junio de 2025, de<https://projectlombok.org/features/>

Spring Boot. (s.f.). Overview [Sitio web]. Recuperado el 12 de junio de 2025, de<https://spring.io/projects/spring-boot>