Arquitectura de Redis en Integrador

Documento Técnico - Febrero 2026

Introducción

Redis constituye el sistema nervioso central de comunicación en tiempo real de Integrador. Su implementación trasciende el uso convencional como almacén de datos en memoria para convertirse en la columna vertebral que habilita la sincronización instantánea entre todos los componentes del sistema. Sin esta infraestructura, la aplicación funcionaría como un sistema tradicional de solicitud-respuesta, obligando a cada cliente a refrescar manualmente su interfaz para visualizar actualizaciones. Con Redis, el ecosistema cobra vida: cuando un comensal realiza un pedido desde su dispositivo móvil, el mozo recibe instantáneamente una notificación, la cocina visualiza la orden en su pantalla, y el administrador observa en el Dashboard cómo la mesa cambia de estado. Este proceso se completa en milisegundos.

Este documento presenta un análisis exhaustivo de la arquitectura Redis implementada en Integrador, abarcando desde la gestión de conexiones hasta los patrones de resiliencia y seguridad que garantizan la operación continua del sistema.

Capítulo 1: El Paradigma de Eventos en Tiempo Real

La arquitectura de Integrador resuelve un desafío fundamental en sistemas distribuidos: la propagación instantánea de cambios de estado a múltiples clientes heterogéneos. Consideremos un escenario típico en un restaurante con veinte mesas, cinco mozos, tres cocineros y un administrador. Sin un sistema de eventos, cuando un cliente en la mesa siete realiza un pedido, el flujo sería deficiente: el pedido se almacena en PostgreSQL, pero el mozo debe refrescar constantemente su aplicación para detectar nuevos pedidos, la cocina desconoce que existe trabajo pendiente hasta consultar manualmente, y el administrador visualiza información desactualizada.

Con la implementación de Redis Pub/Sub, el flujo se transforma radicalmente. El pedido se persiste en PostgreSQL e inmediatamente se publica un evento ROUND_PENDING en Redis. El sistema distribuye este evento a todos los suscriptores relevantes de manera selectiva. El WebSocket Gateway recibe el evento y lo transmite exclusivamente a los mozos y administradores conectados de esa sucursal específica. En menos de cien milisegundos, todos los actores relevantes visualizan el nuevo pedido.

El flujo de comunicación sigue una trayectoria bien definida desde el REST API en el puerto ocho mil hacia Redis en el puerto seis mil trescientos ochenta, y desde allí hacia el WebSocket Gateway en el puerto ocho mil uno. El REST API genera eventos cuando ocurren cambios en el sistema. Redis actúa como intermediario que distribuye mensajes a los suscriptores. El WebSocket Gateway recibe estos eventos mediante pattern subscribe y los envía a los clientes WebSocket apropiados. Los frontends

Dashboard, pwaWaiter y pwaMenu reciben actualizaciones instantáneas según su rol y contexto.

Capítulo 2: Arquitectura de Pools de Conexiones

La implementación de Redis en Integrador no utiliza una conexión única, sino una arquitectura de pools diferenciados que optimiza el rendimiento bajo alta concurrencia. El módulo de pools de Redis implementa dos pools independientes con responsabilidades claramente definidas, ubicados en el archivo redis_pool.py del paquete de eventos de infraestructura compartida.

El pool asíncrono está diseñado para operaciones no bloqueantes, principalmente la publicación de eventos desde el REST API. Se configura con un máximo de cincuenta conexiones simultáneas, permitiendo que hasta cincuenta operaciones de publicación ocurran en paralelo sin bloquear el servidor. Cada conexión incorpora un timeout de cinco segundos para establecimiento y operaciones de lectura y escritura. Una característica crítica es el intervalo de health check de treinta segundos, que permite detectar automáticamente conexiones muertas en el pool.

La inicialización del pool implementa el patrón Singleton con doble verificación de bloqueo. Este patrón garantiza que solo se cree una instancia del pool incluso bajo acceso concurrente de múltiples corrutinas. La primera verificación ocurre sin adquisición de lock para optimizar el caso común donde el pool ya existe. Si no existe, se adquiere un

lock asíncrono y se verifica nuevamente antes de la creación, previniendo condiciones de carrera.

Ciertas operaciones requieren ejecución síncrona, particularmente aquellas que ocurren en contextos donde no existe un event loop de asyncio disponible o donde la simplicidad del código síncrono es preferible. Para estas situaciones, el sistema implementa un pool síncrono separado con veinte conexiones máximas. Las operaciones que utilizan el pool síncrono incluyen la verificación de rate limiting durante el login, la consulta de blacklist de tokens JWT durante la validación de middleware de autenticación, y operaciones de webhook de Mercado Pago que requieren sincronía con sistemas externos.

El pool síncrono utiliza threading.Lock en lugar de asyncio.Lock, ya que opera fuera del contexto asíncrono. Esta separación es fundamental: mezclar locks asíncronos y síncronos puede causar deadlocks sutiles y difíciles de diagnosticar.

El sistema implementa funciones de limpieza coordinadas que se ejecutan durante el shutdown de la aplicación. La función de cierre de pool cierra tanto el pool asíncrono como el síncrono, liberando todas las conexiones al sistema operativo. Esta coordinación es crítica para evitar conexiones huérfanas que podrían agotar los recursos de Redis.

Capítulo 3: Sistema de Canales y Enrutamiento Inteligente

Los canales en Redis siguen una convención de nombres jerárquica que permite el enrutamiento preciso de eventos según su naturaleza y audiencia objetivo. El módulo de canales define siete patrones organizados por dominio.

Los canales de nivel de sucursal incluyen el patrón branch seguido del identificador de sucursal y waiters para eventos destinados a todos los mozos de una sucursal específica, branch con identificador y kitchen para eventos del personal de cocina, y branch con identificador y admin para eventos de administradores y managers de la sucursal.

Los canales de nivel de sector utilizan el patrón sector seguido del identificador de sector y waiters para eventos filtrados por sector específico destinados a mozos asignados a ese sector particular.

Los canales de nivel de sesión emplean el patrón session seguido del identificador de sesión para eventos dirigidos a los comensales de una mesa específica activa, incluyendo actualizaciones de carrito compartido y notificaciones de estado de pedido.

Los canales de nivel de usuario siguen el patrón user con el identificador de usuario para notificaciones directas a un usuario específico. Los canales de nivel de tenant utilizan tenant con identificador y admin para eventos administrativos a nivel de restaurante completo.

El sistema no difunde todos los eventos a todos los canales indiscriminadamente. En su lugar, implementa

un enrutamiento inteligente basado en el tipo de evento, su estado en el ciclo de vida, y la relevancia para cada audiencia.

Cuando se crea un nuevo pedido con estado ROUND_PENDING, el sistema publica en el canal de mozos de la sucursal para que algún mozo lo verifique físicamente en la mesa, y simultáneamente en el canal de administradores para visibilidad en el Dashboard. Notablemente, no se envía a cocina en este punto porque el pedido debe ser verificado primero por el mozo. Tampoco se notifica a los comensales, ya que ellos mismos iniciaron el pedido.

Cuando el mozo verifica el pedido y el estado avanza a ROUND_CONFIRMED, se notifica a los administradores que ahora pueden enviarlo a cocina. Solo cuando el administrador ejecuta la acción de enviar a cocina con estado ROUND_SUBMITTED, el evento se publica en el canal de cocina donde aparece en la pantalla de nuevos pedidos.

Este enrutamiento selectivo minimiza el ruido de notificaciones y garantiza que cada rol visualice exclusivamente la información relevante para su función operativa.

Una característica avanzada del sistema es el filtrado por sector para restaurantes con múltiples zonas de servicio como Interior, Terraza o Barra. El sistema permite enviar eventos únicamente a los mozos asignados al sector correspondiente de la mesa.

Cuando un evento incluye identificador de sector, el sistema publica en dos canales complementarios: el

canal específico del sector para mozos asignados a ese sector, y el canal general de la sucursal como respaldo para garantizar que ningún evento se pierda.

El WebSocket Gateway mantiene un mapeo actualizado de asignaciones mozo-sector mediante el repositorio de sectores con caché TTL de sesenta segundos. Este caché reduce drásticamente las consultas a la base de datos durante períodos de alta actividad.

Capítulo 4: Sistema de Publicación de Eventos

El corazón del sistema de eventos reside en la función de publicación ubicada en el módulo publisher del paquete de eventos. Esta implementación trasciende un simple wrapper de publicación de Redis para incorporar múltiples capas de protección y resiliencia.

La validación de tamaño ocurre antes de cualquier publicación, verificando que el mensaje no exceda sesenta y cuatro kilobytes. Este límite previene problemas de memoria en los suscriptores y latencia excesiva en la red. Si un evento supera este umbral, se registra un warning y la publicación se rechaza con una excepción de valor inválido.

La serialización compacta convierte el evento a JSON sin espacios innecesarios para minimizar el tamaño de transmisión. La serialización incluye manejo especial para objetos datetime y otros tipos no serializables nativamente.

Si la publicación falla por una desconexión temporal de Redis, el sistema implementa reintentos con backoff exponencial decorrelacionado. El delay exponencial base comienza en medio segundo y se duplica en cada intento hasta un máximo de diez segundos. El jitter decorrelacionado previene el problema de thundering herd donde múltiples clientes que fallaron simultáneamente reintentan al mismo instante y sobrecargan Redis nuevamente. Con jitter, los reintentos se distribuyen aleatoriamente en el tiempo.

El sistema implementa el patrón Circuit Breaker en el módulo de circuit breaker para prevenir cascadas de fallos cuando Redis no está disponible. El circuit breaker opera en tres estados. El estado CLOSED representa operación normal donde las publicaciones proceden normalmente y se mantiene un conteo de fallos consecutivos. El estado OPEN representa fallo rápido: después de cinco fallos consecutivos, el circuit breaker se abre y las publicaciones fallan inmediatamente sin intentar contactar a Redis, retornando cero suscriptores. Esto previene que la aplicación se bloquee esperando timeouts de un Redis que claramente no está respondiendo. El estado HALF_OPEN representa recuperación: después de treinta segundos en estado OPEN, el circuit breaker permite un número limitado de pruebas de tres llamadas. Si estas pruebas tienen éxito, el circuit breaker retorna a CLOSED. Si fallan, vuelve a OPEN.

La implementación es thread-safe mediante threading.Lock, permitiendo su uso seguro desde múltiples corrutinas o hilos concurrentes.

Además del Pub/Sub tradicional, el sistema soporta publicación a Redis Streams mediante la función de publicación a stream. Los Streams proporcionan persistencia y garantía de entrega para eventos críticos que no deben perderse. El límite de cincuenta mil entradas proporciona aproximadamente dieciséis horas de buffer a carga máxima, utilizando trimming aproximado para prevenir crecimiento ilimitado.

Capítulo 5: Publicadores de Dominio Especializados

El módulo de publicadores de dominio implementa funciones especializadas para cada tipo de evento del sistema, encapsulando la lógica de routing compleja en interfaces simples.

La función de publicación de eventos de ronda maneja el ciclo completo de vida de pedidos. Cuando el tipo es ROUND_PENDING, publica al canal de mozos del sector si existe sector_id, al canal de mozos de la sucursal como respaldo, y al canal de administradores. Para ROUND_CONFIRMED publica solo a administradores ya que el mozo ya verificó. Para ROUND_SUBMITTED publica a administradores y cocina. Los estados posteriores IN_KITCHEN, READY y SERVED publican a administradores, cocina, mozos y sesión de diners.

La función de publicación de eventos de llamada de servicio maneja solicitudes de atención de clientes. El evento SERVICE_CALL_CREATED publica al canal de mozos del sector y administradores.

SERVICE_CALL_ACKED notifica a la sesión que un mozo reconoció la solicitud. SERVICE_CALL_CLOSED indica que la solicitud fue atendida.

La función de publicación de eventos de facturación maneja el flujo de pagos. CHECK_REQUESTED publica a mozos y administradores. PAYMENT_APPROVED y PAYMENT_REJECTED publican a la sesión de diners para actualizar su interfaz. CHECK_PAID cierra el ciclo de pago.

La función de publicación de eventos de mesa maneja cambios de estado. TABLE_SESSION_STARTED publica a mozos del sector, mozos de la sucursal y administradores cuando un cliente escanea el código QR. TABLE_CLEARED indica que la sesión terminó y la mesa está disponible. TABLE_STATUS_CHANGED notifica cambios de estado intermedios.

La función de publicación de eventos CRUD administrativos notifica cambios en entidades del sistema. ENTITY_CREATED, ENTITY_UPDATED, ENTITY_DELETED y CASCADE_DELETE publican al canal de administradores de la sucursal y opcionalmente al canal de administradores del tenant.

La función de publicación de eventos de carrito maneja la sincronización del carrito compartido entre dispositivos. CART_ITEM_ADDED, CART_ITEM_UPDATED, CART_ITEM_REMOVED y CART_CLEARED publican exclusivamente al canal de sesión, ya que solo afectan a los comensales de esa mesa específica. CART_SYNC proporciona el estado completo del carrito para reconexiones.

Capítulo 6: Suscripción y Procesamiento de Eventos

El WebSocket Gateway implementa el lado receptor del sistema Pub/Sub en el módulo de subscriber. Esta implementación sigue un patrón de orquestador delgado que delega la lógica específica a módulos especializados extraídos durante la refactorización arquitectónica.

El suscriptor utiliza pattern subscribe para escuchar múltiples canales con patrones glob. Los patrones incluyen branch asterisco waiters para todos los canales de mozos, branch asterisco kitchen para todos los canales de cocina, branch asterisco admin para todos los canales de administración, sector asterisco waiters para todos los canales de sector, y session asterisco para todas las sesiones de mesa.

Los eventos pueden llegar más rápido de lo que el sistema puede procesarlos, especialmente durante picos de actividad. Para manejar esta situación, el suscriptor implementa una cola de backpressure con capacidad configurable de quinientos eventos, optimizado para sucursales de aproximadamente cien mesas.

Si la cola alcanza su capacidad máxima, los eventos más antiguos se descartan automáticamente gracias al límite máximo del deque. El sistema monitorea activamente la tasa de descarte mediante el tracker de drop rate.

El módulo de tracking de drop rate implementa un tracker con ventana deslizante de sesenta segundos. Si más del cinco por ciento de los eventos están siendo descartados en la ventana, se genera una alerta crítica indicando que el sistema está sobrecargado y requiere atención inmediata. El cooldown de cinco minutos entre alertas previene tormentas de notificaciones.

Para optimizar el rendimiento, los eventos no se procesan individualmente sino en lotes de hasta cincuenta eventos. Este enfoque reduce la sobrecarga de cambios de contexto y mejora el throughput general del sistema.

El procesamiento de cada lote incluye validación del esquema del evento contra tipos conocidos, determinación de destinatarios identificando qué conexiones WebSocket deben recibir el evento, envío paralelo a todas las conexiones relevantes mediante asyncio.gather, y registro de métricas de éxito y fallo para monitoreo.

Si la conexión con Redis se pierde, el suscriptor implementa reconexión automática con backoff exponencial y jitter decorrelacionado. El proceso de reconexión incluye limpieza del pubsub anterior con timeouts de cinco segundos para unsubscribe y close, creación de un nuevo objeto pubsub, y re-suscripción a todos los canales. Después de veinte intentos fallidos, el suscriptor reporta un error fatal pero continúa intentando indefinidamente.

Capítulo 7: Seguridad con Redis

Cuando un usuario cierra sesión, su token JWT sigue siendo técnicamente válido hasta su expiración natural de quince minutos para access tokens. Para invalidar estos tokens inmediatamente, el sistema mantiene una blacklist en Redis implementada en el módulo de blacklist de tokens.

Para agregar a blacklist, cuando un usuario ejecuta logout, el identificador único JWT del token se almacena en Redis con clave compuesta del prefijo de blacklist de autenticación seguido del identificador. El TTL se calcula dinámicamente para coincidir con el tiempo restante de validez del token, permitiendo que Redis limpie automáticamente las entradas expiradas.

Para verificación, el sistema consulta si la clave existe en Redis. La optimización utiliza Redis Pipeline para combinar múltiples consultas en un solo round-trip, verificando simultáneamente la blacklist individual y la revocación a nivel de usuario.

La política de fallo cerrado es crítica: si Redis no está disponible durante la verificación de blacklist, el sistema asume que el token está en la blacklist y rechaza la solicitud. Esta política sigue el principio de seguridad de denegar acceso cuando hay duda.

Además de la blacklist individual, el sistema permite revocar todos los tokens de un usuario mediante un timestamp de revocación almacenado con el prefijo de revocación de usuario seguido del identificador de usuario. Cualquier token emitido antes de este

timestamp se considera inválido, incluso si no está explícitamente en la blacklist.

Esta funcionalidad es esencial cuando se detecta actividad sospechosa en la cuenta, cuando el usuario cambia su contraseña, o cuando un administrador fuerza el cierre de todas las sesiones de un usuario.

El sistema protege contra ataques de fuerza bruta limitando los intentos de login a cinco por minuto por dirección de email. La implementación utiliza un script Lua para garantizar atomicidad de la operación INCR más EXPIRE. El script incrementa el contador, establece el TTL si es el primer intento de la ventana, maneja el caso donde el TTL se perdió por alguna razón re-estableciéndolo, y retorna el conteo actual junto con el TTL restante.

Este script previene una condición de carrera sutil: sin atomicidad, si el servidor fallara entre INCR y EXPIRE, la clave quedaría sin TTL y el contador nunca se resetearía.

El SHA del script se cachea en memoria para evitar re-transmitirlo en cada llamada. Si Redis reinicia y pierde los scripts cargados, el sistema detecta el error NOSCRIPT y recarga automáticamente.

Cuando el límite se excede, el sistema responde con HTTP cuatrocientos veintinueve e incluye el header Retry-After con el tiempo restante en segundos.

Capítulo 8: Rate Limiting en WebSocket

El rate limiting de conexiones WebSocket se implementa en el módulo de rate limiter del Gateway. La clase WebSocketRateLimiter implementa un algoritmo de ventana deslizante con contador.

El algoritmo mantiene una lista de timestamps por conexión. En cada verificación, elimina los timestamps fuera de la ventana configurada y rechaza si el conteo excede el máximo de mensajes permitidos. La configuración por defecto permite veinte mensajes por segundo por conexión.

El sistema implementa límites de memoria con un máximo de cinco mil conexiones tracked. Cuando se alcanza este límite, se ejecuta evicción de las entradas más antiguas eliminando el diez por ciento para hacer espacio.

Una característica avanzada es el tracking de penalizaciones para conexiones eviccionadas. Esto previene que usuarios maliciosos evadan el rate limit simplemente reconectando. La penalización persiste por una hora después de la evicción.

Las métricas del rate limiter incluyen conteos de mensajes permitidos, mensajes rechazados y eviciones, proporcionando visibilidad sobre el comportamiento del sistema bajo carga.

Capítulo 9: Caché de Sectores

El WebSocket Gateway implementa caché de asignaciones de sectores para reducir consultas a la base de datos. La clase SectorCache en el repositorio de sectores mantiene un caché en memoria con TTL configurable.

El caché utiliza una tupla de identificador de usuario e identificador de tenant como clave, almacenando una lista de identificadores de sectores asignados. El TTL por defecto es de sesenta segundos, permitiendo que cambios de asignación se reflejen en un minuto máximo.

El límite máximo de mil entradas previene crecimiento descontrolado de memoria. Cuando se alcanza el límite, el sistema ejecuta evicción de las entradas más antiguas por timestamp de creación siguiendo el patrón LRU.

La implementación es thread-safe mediante threading.Lock, permitiendo acceso seguro desde múltiples corrutinas del event loop. Las métricas incluyen hits, misses y ratio de aciertos para monitorear la efectividad del caché.

El método de limpieza de expirados elimina entradas cuyo TTL ha vencido. El método de invalidación permite remover entradas específicas cuando se actualiza una asignación en la base de datos.

Capítulo 10: Patrón Outbox para Entrega Garantizada

Un desafío crítico en sistemas basados en eventos es garantizar que los datos de negocio y los eventos se guarden de forma atómica. Consideremos un escenario problemático: el backend guarda un pago en PostgreSQL, intenta publicar PAYMENT_APPROVED a Redis, pero Redis está temporalmente indisponible. El pago existe en la base de datos pero el evento nunca se publicó. El Dashboard y pwaWaiter nunca reciben la notificación.

El Patrón Outbox resuelve este problema escribiendo eventos a una tabla de la base de datos dentro de la misma transacción que los datos de negocio.

El flujo del patrón Outbox comienza en el endpoint REST API donde se crea el Payment y se escribe el OutboxEvent en la misma transacción. Al ejecutar commit, ambos se guardan atómicamente. El Outbox Processor, que corre como tarea de background cada segundo, selecciona eventos con estado PENDING ordenados por timestamp de creación. Actualiza el estado a PROCESSING para prevenir doble publicación. Publica a Redis. Actualiza el estado a PUBLISHED o incrementa el contador de reintentos si falla. Después de cinco reintentos fallidos, marca el evento como FAILED.

El modelo OutboxEvent incluye campos para identificador de tenant, tipo de evento, tipo de agregado como round o check o service_call, identificador de agregado, payload serializado como JSON, estado con valores posibles PENDING, PROCESSING, PUBLISHED o FAILED, contador de reintentos, último error, timestamp de creación y timestamp de procesamiento.

Los índices optimizados para polling eficiente incluyen un índice compuesto en estado y timestamp de creación para la query principal del processor, y un índice en tenant y estado para filtrado por tenant.

Los eventos cubiertos por Outbox incluyen CHECK_REQUESTED de alta criticidad porque el cliente espera confirmación de cuenta, PAYMENT_APPROVED de criticidad máxima porque afecta dinero y el Dashboard debe actualizar, PAYMENT_REJECTED de alta criticidad porque el cliente debe reintentar pago, CHECK_PAID de criticidad máxima porque cierra el flujo de pago, ROUND_SUBMITTED de alta criticidad porque cocina debe recibir pedido, ROUND_READY de alta criticidad porque el mozo debe servir plato listo, y SERVICE_CALL_CREATED de criticidad media porque representa una llamada de servicio al mozo.

Los eventos de menor criticidad como ROUND_CONFIRMED, ROUND_IN_KITCHEN y ROUND_SERVED usan publicación directa a Redis por eficiencia.

Capítulo 11: Redis Streams Consumer con Recuperación

Mientras Redis Pub/Sub es eficiente para notificaciones en tiempo real, no garantiza entrega. Si el WebSocket Gateway está reiniciando cuando llega un evento, el mensaje se pierde. Para eventos críticos, el sistema implementa un Consumer Group sobre Redis Streams.

El módulo de stream consumer implementa un consumidor completo con recuperación de mensajes pendientes mediante PEL (Pending Entries List), Dead Letter Queue para mensajes fallidos, y backoff exponencial con jitter para errores.

La configuración del Stream Consumer utiliza la clave events:critical como nombre del stream, ws_gateway_group como nombre del consumer group, y gateway-primary como nombre del consumidor. La verificación de PEL ocurre cada treinta ciclos, aproximadamente cada minuto. Los mensajes pendientes por más de treinta segundos se consideran para recuperación. Después de tres reintentos, los mensajes se mueven a la Dead Letter Queue.

El flujo del consumer comienza creando el Consumer Group si no existe. Luego recupera mensajes pendientes de sesiones anteriores. El loop principal lee mensajes nuevos con XREADGROUP bloqueando por dos segundos, procesa cada mensaje y ejecuta ACK en éxito. Cada treinta ciclos verifica la PEL para mensajes abandonados.

La recuperación de PEL utiliza XAUTOCLAIM para reclamar mensajes idle automáticamente. Obtiene mensajes pendientes por más de treinta segundos. Para cada mensaje, obtiene el conteo de reintentos del metadata. Si excede el máximo de reintentos, mueve el mensaje a la Dead Letter Queue con metadata completo incluyendo identificador original, stream de origen, datos, conteo de reintentos, timestamp de fallo y nombre del consumidor. Si no excede el máximo, reintenta el procesamiento.

El manejo del error NOGROUP ocurre si el Consumer Group se elimina externamente, por ejemplo cuando Redis reinicia sin persistencia. El consumer detecta el error y recrea el grupo automáticamente.

Capítulo 12: Scripts Lua para Atomicidad

El módulo de scripts Lua del WebSocket Gateway implementa rate limiting atómico para conexiones WebSocket. El script de rate limiting recibe la clave de rate limit, el máximo de mensajes permitidos, el tamaño de ventana en segundos y el timestamp actual. Retorna una tupla indicando si está permitido, el conteo actual y el TTL restante.

El script primero obtiene el contador actual. Verifica si excede el límite y retorna cero si lo hace, indicando rechazo. Si está permitido, incrementa el contador. Establece el TTL solo en el primer request de la ventana. Retorna uno con el nuevo conteo y TTL, indicando que la operación está permitida.

Las ventajas sobre implementación en Python son significativas. La atomicidad garantiza que todo ejecuta como operación atómica en Redis, eliminando race conditions. Un único round-trip reduce la latencia comparado con operaciones separadas de GET, INCR y EXPIRE. La imposibilidad de que dos requests incrementen simultáneamente elimina inconsistencias.

El caché de SHA almacena el identificador del script después de la primera carga para evitar re-

transmitirlo en cada llamada. Si Redis reinicia y pierde los scripts cargados, el sistema detecta el error NOSCRIPT y recarga automáticamente.

El ACK en lote con Pipeline agrupa acknowledgments para alto throughput. Procesa en lotes de cien para evitar pipelines excesivamente grandes. Reduce significativamente la latencia cuando el Gateway procesa lotes grandes de mensajes.

Capítulo 13: Schema de Eventos

El dataclass Event define la estructura de todos los eventos del sistema. Los campos requeridos incluyen type como tipo de evento, tenant_id como identificador de tenant que debe ser positivo, y branch_id como identificador de sucursal que debe ser mayor o igual a cero, donde cero indica eventos a nivel de tenant.

Los campos opcionales incluyen table_id para identificador de mesa, session_id para identificador de sesión de mesa, sector_id para routing específico de sector, entity como diccionario con datos específicos del evento, actor como diccionario identificando quién disparó el evento con user_id y role, ts como timestamp en formato ISO que se genera automáticamente, y v como versión del schema con valor por defecto de uno.

La validación en post-init verifica que todos los campos requeridos estén presentes, que los tipos sean correctos verificando strings, enteros y

diccionarios, y que los identificadores sean positivos. Lanza ValueError si la validación falla.

La serialización a JSON convierte el evento a string JSON con timestamp ISO. La deserialización desde JSON reconstruye el objeto Event desde un string JSON.

Capítulo 14: Monitoreo y Diagnóstico

El sistema expone múltiples métricas para monitorear la salud de la infraestructura Redis.

El estado del Circuit Breaker incluye el estado actual que puede ser closed, open o half_open, el conteo de fallos consecutivos actuales, el total de llamadas rechazadas históricamente, y el timestamp del último fallo.

El Drop Rate del Suscriptor incluye el total de eventos procesados, el total de eventos descartados, los procesados en la ventana actual, los descartados en la ventana actual, la tasa de descarte porcentual actual, el umbral de alerta configurado, el conteo de alertas emitidas, y el tamaño de ventana en segundos.

El endpoint de health detallado del WebSocket Gateway retorna estado de redis_async, información del pool síncrono, y estadísticas de conexiones activas.

Para diagnosticar el problema de eventos que no llegan a clientes, se debe verificar que Redis esté corriendo, verificar suscripción del WebSocket Gateway en logs buscando el mensaje de inicio del

subscriber, verificar publicación de eventos en logs del REST API buscando el mensaje de evento publicado a canal, verificar conexión WebSocket del cliente en consola del navegador, y verificar que el cliente está suscrito a los canales correctos.

Para diagnosticar alta latencia en eventos, se debe verificar el tamaño de la cola de eventos ya que si está cerca del máximo indica sistema sobrecargado, revisar drop rate ya que más del cinco por ciento indica problemas de capacidad, verificar latencia de Redis, verificar si el circuit breaker está en estado OPEN, y revisar CPU y memoria del servidor Redis.

Para diagnosticar conexión a Redis que falla intermitentemente, se debe verificar estado del circuit breaker en métricas, buscar errores de conexión en logs, verificar recursos del servidor Redis incluyendo memoria, CPU y conexiones, verificar el límite de conexiones de Redis en maxclients, y considerar aumentar timeouts si la red tiene latencia variable.

Para diagnosticar rate limiting que no funciona, se debe verificar que Redis esté accesible desde el backend, verificar que el script Lua esté cargado buscando el mensaje de cache miss en logs, y verificar que las claves de rate limit tengan TTL correcto consultando Redis directamente.

Capítulo 15: Configuración y Parámetros

La configuración centralizada en el módulo de settings define todos los parámetros de Redis con valores optimizados para restaurantes.

Los parámetros de conexión base incluyen redis_url con valor por defecto de redis://localhost:6380 usando puerto seis mil trescientos ochenta en desarrollo, y redis_socket_timeout de cinco segundos para timeout de conexión y operaciones I/O.

Los parámetros de pools de conexión incluyen redis_pool_max_connections de cincuenta para conexiones asíncronas máximas, y redis_sync_pool_max_connections de veinte para conexiones síncronas máximas utilizadas en rate limiting y blacklist.

Los parámetros de procesamiento de eventos incluyen redis_event_queue_size de quinientos como tamaño de cola de backpressure, redis_event_batch_size de cincuenta como eventos procesados por lote, redis_publish_max_retries de tres como reintentos de publicación, y redis_publish_retry_delay de cero punto uno segundos como delay base para reintentos.

Los parámetros de reconexión y timeouts incluyen redis_max_reconnect_attempts de veinte como intentos máximos de reconexión, redis_pubsub_cleanup_timeout de cinco segundos como timeout para limpieza de pubsub, y redis_pubsub_reconnect_total_timeout de quince segundos como timeout total de reconexión.

Los parámetros de control de calidad incluyen redis_event_strict_ordering como false por defecto indicando si eventos reintentados van al frente de la

cola, y redis_event_staleness_threshold de cinco segundos para advertencia si un evento espera más de ese tiempo.

Capítulo 16: Recomendaciones de Escalamiento

Para un restaurante pequeño con cinco a diez mesas y aproximadamente cincuenta conexiones WebSocket concurrentes, la configuración por defecto es adecuada.

Para restaurantes medianos con veinte a cincuenta mesas y aproximadamente doscientas conexiones concurrentes, se recomienda aumentar redis_pool_max_connections a cien, aumentar redis_event_queue_size a dos mil, y considerar Redis en hardware dedicado si comparte servidor con PostgreSQL.

Para cadenas con múltiples sucursales con más de cien mesas totales y más de quinientas conexiones concurrentes, se recomienda implementar Redis Cluster o Redis Sentinel para alta disponibilidad, aumentar redis_pool_max_connections a doscientos, aumentar redis_sync_pool_max_connections a cincuenta, utilizar servidor Redis dedicado con réplica para failover, y establecer monitoreo activo de métricas de latencia y memoria.

Conclusión

Redis en Integrador representa una implementación enterprise-grade que trasciende el uso convencional de almacén de datos en memoria. La arquitectura implementa múltiples capas de resiliencia y patrones avanzados que garantizan operación continua incluso bajo condiciones adversas.

Los patrones de resiliencia implementados incluyen Circuit Breaker que previene cascadas de fallos cuando Redis no responde, backoff exponencial con jitter que evita thundering herd en reconexiones, patrón Outbox que garantiza consistencia transaccional entre PostgreSQL y Redis, Consumer Groups con recuperación de PEL que aseguran entrega de eventos críticos, y Dead Letter Queue que preserva mensajes fallidos para análisis posterior.

Las optimizaciones de rendimiento incluyen pools separados async y sync que optimizan diferentes patrones de acceso, scripts Lua atómicos que eliminan race conditions en rate limiting, pipeline batching que reduce latencia en ACKs masivos, y constantes centralizadas que garantizan consistencia entre publisher y consumer.

Las políticas de seguridad siguen el principio fundamental de fail-closed: ante cualquier duda o error, el sistema deniega acceso en lugar de permitirlo. El rate limiting atómico mediante Lua previene ataques de fuerza bruta, y la blacklist de tokens proporciona revocación instantánea de sesiones.

La arquitectura dual de Pub/Sub más Streams proporciona el balance óptimo. Pub/Sub maneja eventos

de baja latencia donde pérdida ocasional es aceptable. Streams maneja eventos críticos como pagos y pedidos a cocina que requieren entrega garantizada.

El monitoreo integral mediante métricas de circuit breaker, drop rate y estadísticas de conexión permite identificar problemas antes de que afecten a los usuarios. La configuración flexible soporta desde pequeños restaurantes hasta cadenas con múltiples sucursales.

Esta arquitectura representa las mejores prácticas de la industria para sistemas de eventos en tiempo real, proporcionando la base sólida sobre la cual Integrador construye su experiencia de usuario diferenciada.

Documento generado: Febrero 2026
Versión: 2.0 - Reescrito en prosa narrativa

WebSocket Gateway: Arquitectura Modular y Funcionamiento Interno

Versión 3.0 - Febrero 2026

Introducción

El WebSocket Gateway constituye el sistema nervioso central de comunicación en tiempo real dentro del ecosistema Integrador. Este servicio, ejecutándose en el puerto 8001, desempeña un rol fundamental como intermediario entre los eventos de dominio generados por la API REST y los múltiples clientes conectados

que requieren notificaciones instantáneas. A diferencia del modelo tradicional de polling, donde los clientes consultan repetidamente al servidor en busca de actualizaciones, el Gateway mantiene conexiones bidireccionales persistentes que permiten la transmisión inmediata de eventos sin latencia perceptible para el usuario final.

La arquitectura del Gateway fue diseñada desde sus cimientos para soportar operaciones de alta concurrencia en entornos de restauración exigentes. Con capacidad para manejar simultáneamente entre cuatrocientas y mil conexiones WebSocket, el sistema implementa patrones avanzados de gestión de recursos que incluyen locks fragmentados organizados por sucursal y usuario, un worker pool dedicado para broadcasting paralelo con diez workers, y pools de conexiones Redis optimizados para diferentes patrones de uso. Esta infraestructura robusta permite que un restaurante en hora pico pueda operar con docenas de mozos activos, múltiples terminales de cocina y cientos de comensales consultando el estado de sus pedidos, todos recibiendo actualizaciones en tiempo real sin degradación del servicio.

El proyecto experimentó una refactorización arquitectónica significativa, identificada internamente como ARCH-MODULAR, que transformó archivos monolíticos de casi mil líneas de código en orquestadores delgados que delegan responsabilidades específicas a módulos especializados. El connection manager se redujo de novecientas ochenta y siete líneas a aproximadamente cuatrocientas sesenta y tres, mientras que el redis subscriber pasó de seiscientas sesenta y seis líneas a cerca de

trescientas veintiséis. Esta transformación no solo mejoró dramáticamente la mantenibilidad del código base sino que también habilitó testing unitario granular de cada componente y permitió la evolución independiente de subsistemas sin afectar otras partes del Gateway.

Capítulo 1: Topología del Proyecto y Organización de Módulos

El WebSocket Gateway reside en la carpeta ws_gateway ubicada en la raíz del proyecto Integrador. Su organización interna refleja una arquitectura de componentes donde cada módulo posee una responsabilidad única y claramente definida, siguiendo el principio de responsabilidad única que constituye la S en los principios SOLID.

El archivo main.py representa el punto de entrada de la aplicación FastAPI con aproximadamente doscientas ochenta líneas de código. Este módulo configura la aplicación, define el lifespan manager que orquesta el ciclo de vida completo del servicio, y registra los endpoints WebSocket junto con los endpoints de salud y métricas. Aquí se inicializan los workers de broadcasting, se establecen las suscripciones a Redis, se configuran las tareas de mantenimiento periódico como la limpieza de conexiones inactivas, y se define el consumidor de Redis Streams para eventos críticos que requieren entrega garantizada.

Los dos orquestadores principales, connection_manager.py y redis_subscriber.py, fueron

refactorizados desde implementaciones monolíticas a coordinadores delgados que actúan como fachadas componiendo módulos especializados mediante inyección de dependencias. El connection manager ahora coordina cuatro módulos extraídos: ConnectionLifecycle para la aceptación y desconexión de clientes, ConnectionBroadcaster para envío paralelo mediante worker pools, ConnectionCleanup para eliminación de conexiones muertas o inactivas, y ConnectionStats para agregación de estadísticas de rendimiento. De manera análoga, el redis subscriber coordina módulos para validación de eventos, tracking de eventos perdidos, y procesamiento por lotes.

El directorio core contiene los módulos extraídos de los orquestadores originales, organizados por dominio funcional. El subdirectorio connection agrupa cuatro módulos esenciales. El archivo lifecycle.py maneja la aceptación y desconexión de clientes WebSocket, implementando la verificación de límites de conexión, la validación de parámetros, y el registro en índices. El archivo broadcaster.py implementa el envío paralelo mediante worker pools con diez workers que procesan envíos concurrentemente, incluyendo la estrategia híbrida que selecciona automáticamente entre modo legacy para audiencias pequeñas y modo worker pool para broadcasts masivos. El archivo cleanup.py gestiona la identificación y eliminación de conexiones muertas o stale mediante el patrón de dos fases que evita errores de modificación de diccionarios durante iteración. El archivo stats.py agrega estadísticas de rendimiento incluyendo conteos de conexiones, broadcasts y eventos.

El subdirectorio subscriber dentro de core contiene módulos para el procesamiento de eventos Redis. El archivo drop_tracker.py monitorea y genera alertas cuando la tasa de eventos perdidos supera umbrales configurados, implementando una ventana deslizante de observación con cooldown entre alertas para evitar saturación del sistema de logging. El archivo validator.py valida esquemas de eventos entrantes verificando la presencia de campos requeridos como type, tenant_id y branch_id, y que el tipo de evento sea uno de los tipos conocidos y válidos. El archivo processor.py procesa lotes de eventos eficientemente, desacoplando la recepción de Redis del envío a conexiones WebSocket para mantener baja latencia.

El directorio components organiza funcionalidad transversal por dominio de responsabilidad, proporcionando una capa de abstracción adicional sobre los módulos core. Esta separación refleja dos niveles de abstracción complementarios: mientras core contiene los módulos extraídos directamente de los orquestadores monolíticos originales conservando su lógica fundamental, components organiza funcionalidad transversal facilitando la composición y reutilización de comportamientos comunes.

Capítulo 2: El Punto de Entrada y Gestión del Ciclo de Vida

El archivo main.py configura la aplicación FastAPI y orquesta el ciclo de vida completo del Gateway mediante una función de lifespan asíncrona decorada con el context manager de asynccontextmanager. Esta

función constituye el corazón de la orquestación del servicio, iniciando y deteniendo múltiples subsistemas de forma coordinada y ordenada.

Durante la fase de startup, el sistema inicializa cinco componentes críticos que operarán en paralelo durante la vida del servicio. Primero, se inicializa el broadcaster del connection manager llamando a manager.broadcaster.initialize(), lo que arranca el worker pool con diez workers que procesan envíos de mensajes concurrentemente mediante una cola asíncrona con capacidad máxima de cinco mil tareas pendientes, proporcionando backpressure cuando el sistema experimenta picos de carga. Segundo, se crea una instancia singleton del EventRouter mediante init_event_router() que centraliza la validación y enrutamiento de eventos, evitando la creación de múltiples instancias que podrían causar inconsistencias o condiciones de carrera en el routing.

Tercero, se inicia la tarea del suscriptor Redis Pub/Sub creando una coroutine que ejecuta subscriber.start() como background task. Esta tarea escucha eventos en canales de sucursal, sector y sesión mediante suscripciones por patrón que permiten capturar dinámicamente todos los canales relevantes sin conocerlos de antemano. Cuarto, se lanza el consumidor de Redis Streams para eventos críticos que requieren entrega garantizada, complementando el sistema Pub/Sub con semánticas de at-least-once delivery. Quinto, se activa la tarea de limpieza de heartbeat mediante create_task(heartbeat_cleanup_task()) que ejecuta periódicamente cada treinta segundos para remover

conexiones que han permanecido inactivas por más del timeout configurado.

La fase de shutdown procede de manera ordenada y segura para evitar pérdida de datos o corrupción de estado. Primero se cancelan las tareas de suscripción y consumo de eventos verificando que existan antes de llamar cancel() y esperando su finalización con suppress de CancelledError, asegurando que no se procesen eventos nuevos durante el cierre. Luego se detiene el worker pool de broadcasting llamando a manager.broadcaster.shutdown() con un timeout de cinco segundos, permitiendo que los envíos en progreso finalicen graciosamente antes de forzar la terminación de workers que no respondan. A continuación se espera la finalización de cualquier operación de limpieza de locks pendiente. Finalmente se liberan los recursos del repositorio de sectores mediante cleanup_sector_repository() y se cierran los pools de conexiones Redis para liberar todos los recursos del sistema.

El Gateway expone diferentes endpoints WebSocket diferenciados por rol y método de autenticación, cada uno manejado por una clase de endpoint especializada que hereda de las clases base definidas en components/endpoints. El endpoint /ws/waiter acepta tokens JWT y permite conexiones de usuarios con roles WAITER, MANAGER o ADMIN, entregando notificaciones filtradas por los sectores asignados al mozo según su configuración del día actual. El endpoint /ws/kitchen también utiliza autenticación JWT pero está destinado a personal con roles KITCHEN, MANAGER o ADMIN, recibiendo únicamente las comandas que han sido enviadas a cocina y requieren preparación. El

endpoint /ws/admin proporciona visibilidad completa de sucursal para usuarios MANAGER o ADMIN mediante autenticación JWT, recibiendo todos los eventos de la sucursal sin filtrado. El endpoint /ws/diner utiliza table tokens HMAC en lugar de JWT, permitiendo que los comensales reciban actualizaciones sobre el estado de sus pedidos personales sin necesidad de cuentas de usuario en el sistema.

Adicionalmente, el Gateway expone tres endpoints de observabilidad esenciales para el monitoreo operacional. El endpoint /ws/health responde a solicitudes GET con una verificación básica de disponibilidad del servicio retornando status healthy. El endpoint /ws/health/detailed ofrece información extendida ejecutando verificaciones asíncronas de Redis, reportando el estado de los pools de conexión, estadísticas de conexiones activas desglosadas por tipo, y el estado del circuit breaker. El endpoint /ws/metrics expone métricas en formato Prometheus para integración con sistemas de monitoreo y alerting, formateando los contadores del MetricsCollector según la especificación de Prometheus con líneas HELP y TYPE.

Capítulo 3: El Connection Manager como Orquestador Central

El ConnectionManager representa la fachada principal para todas las operaciones relacionadas con conexiones WebSocket, residiendo en el archivo connection_manager.py con aproximadamente cuatrocientas sesenta y tres líneas de código tras la

refactorización ARCH-MODULAR-08. Esta clase actúa como punto de entrada unificado que coordina múltiples subsistemas especializados, proporcionando una interfaz coherente mientras delega la complejidad a módulos internos que pueden ser testeados y mantenidos independientemente.

El constructor del ConnectionManager inicializa los componentes core necesarios mediante inyección de dependencias a través de la clase ConnectionManagerDependencies definida en components/core/dependencies.py. Esta clase contenedora agrupa todas las dependencias como singletons que pueden ser reemplazados durante testing. El LockManager proporciona coordinación de concurrencia mediante locks fragmentados por sucursal y usuario. El MetricsCollector centraliza estadísticas con operaciones thread-safe para contextos tanto async como sync. El HeartbeatTracker detecta conexiones inactivas manteniendo timestamps de última actividad. El WebSocketRateLimiter previene abuso limitando mensajes por conexión. El ConnectionIndex proporciona búsqueda rápida mediante múltiples índices.

La verdadera potencia del diseño radica en la composición de módulos especializados del directorio core/connection. El objeto _lifecycle de tipo ConnectionLifecycle encapsula toda la lógica de registro y desregistro de conexiones, incluyendo verificación de límites, validación de parámetros, aceptación del WebSocket con timeout, y actualización de índices. El objeto _cleanup de tipo ConnectionCleanup maneja la identificación y eliminación de conexiones muertas o stale mediante el

patrón de dos fases. El objeto _broadcaster de tipo ConnectionBroadcaster implementa el envío paralelo de mensajes mediante worker pools con estrategia híbrida de selección. El objeto _stats de tipo ConnectionStats agrega y expone estadísticas de rendimiento consultando los índices y el metrics collector.

Esta composición permite que cada aspecto de la gestión de conexiones sea probado, mantenido y evolucionado independientemente sin afectar otros componentes. Los métodos públicos del manager simplemente delegan al módulo apropiado con mínima lógica de coordinación: una llamada a connect() invoca internamente _lifecycle.connect(), mientras que send_to_branch() delega a _broadcaster.send_to_branch(). Este patrón de delegación mantiene el código del manager legible y facilita la sustitución de implementaciones durante testing mediante el contenedor de dependencias.

El manager también expone propiedades de conveniencia que proporcionan acceso a submódulos para código que necesita funcionalidad específica. La propiedad broadcaster retorna el objeto _broadcaster permitiendo acceso directo a métodos de envío especializados. La propiedad lifecycle retorna el objeto _lifecycle para operaciones de registro avanzadas. Estas propiedades mantienen la encapsulación mientras permiten acceso controlado cuando es necesario.

Capítulo 4: Sistema de Índices para Búsqueda de Conexiones

El sistema mantiene múltiples índices para localizar conexiones en tiempo constante O(1) según diferentes criterios de búsqueda, implementados en la clase ConnectionIndex ubicada en components/connection/index.py. Esta clase encapsula las estructuras de datos junto con las operaciones de registro, búsqueda y eliminación, actuando como un value object que representa el estado actual de todas las conexiones del sistema.

Los índices primarios organizan conexiones por atributo clave utilizando diccionarios que mapean identificadores a conjuntos de WebSockets. El diccionario by_user mapea identificadores de usuario a conjuntos de sus conexiones activas, permitiendo enviar mensajes a todas las sesiones de un usuario específico cuando este tiene múltiples dispositivos conectados. El diccionario by_branch agrupa conexiones por sucursal, fundamental para broadcasts de eventos de negocio que deben llegar a todo el personal de una ubicación específica. El diccionario by_sector mantiene conexiones de mozos organizadas por sector asignado, habilitando notificaciones dirigidas solo al personal responsable de ciertas mesas cuando un evento solo es relevante para el área específica del restaurante. El diccionario by_session agrupa conexiones de comensales por sesión de mesa, permitiendo que todos los participantes de una mesa reciban actualizaciones de su pedido compartido en tiempo real.

Los índices por rol proporcionan acceso rápido a conexiones con permisos específicos que requieren tratamiento diferenciado. El diccionario admins_by_branch contiene conexiones de administradores y managers por sucursal, utilizado para eventos que requieren visibilidad de gestión como confirmaciones de pedidos o alertas operativas. El diccionario kitchen_by_branch agrupa personal de cocina por sucursal, destinatarios de comandas y actualizaciones de preparación que solo son relevantes para el área de producción.

Los mappings inversos permiten obtener información sobre una conexión específica sin iterar sobre todos los índices, operación que sería costosa en un sistema con cientos de conexiones. El diccionario _ws_to_user mapea cada WebSocket a su user_id asociado, permitiendo identificar al propietario de una conexión durante operaciones de limpieza. El diccionario _ws_to_tenant mapea cada conexión a su tenant_id para filtrado multi-tenant que garantiza aislamiento entre restaurantes. El diccionario _ws_to_branches almacena los branch_ids a los que cada conexión tiene acceso, utilizado durante verificaciones de autorización.

El diseño de índices múltiples permite broadcasts selectivos sin necesidad de iterar sobre todas las conexiones activas del sistema, lo cual degradaría el rendimiento bajo carga. Cuando un evento debe llegar únicamente a los mozos del sector Terraza, el sistema consulta directamente by_sector con el identificador del sector y obtiene inmediatamente el conjunto exacto de conexiones destinatarias, sin examinar las cientos de otras conexiones que podrían estar activas

simultáneamente en otros sectores, sucursales o roles.

Capítulo 5: Locks Fragmentados y Prevención de Deadlocks

Para manejar alta concurrencia sin contención excesiva que degradaría el rendimiento, el Gateway implementa un sistema de locks fragmentados coordinado por el LockManager ubicado en components/connection/locks.py. Esta estrategia divide la sincronización en múltiples locks independientes, permitiendo que operaciones no relacionadas procedan en paralelo sin bloquearse mutuamente.

El sistema define diferentes tipos de locks según su alcance y el recurso que protegen. El connection_counter_lock protege el contador global de conexiones activas como un recurso compartido crítico, asegurando que los límites de capacidad se respeten atómicamente cuando múltiples conexiones intentan registrarse simultáneamente. Los branch_locks proporcionan un lock por cada sucursal almacenados en un defaultdict, de modo que operaciones en sucursales diferentes no compiten por el mismo recurso y pueden ejecutarse en paralelo completo. Los user_locks ofrecen un lock por cada usuario, permitiendo que conexiones de diferentes usuarios se manejen concurrentemente sin interferencia.

Locks adicionales protegen estructuras de datos específicas que cruzan los límites de usuario o sucursal. El sector_lock protege el índice by_sector durante actualizaciones que afectan múltiples sectores. El session_lock protege el índice by_session para conexiones de comensales. El dead_connections_lock protege el conjunto de conexiones marcadas para limpieza diferida.

Cuando un mozo se conecta a la sucursal número cinco, el sistema solo adquiere el lock de esa sucursal específica mediante get_branch_lock(5). Esto permite que conexiones simultáneas a otras sucursales procedan sin bloqueo alguno. Esta estrategia de fragmentación reduce la contención en aproximadamente un noventa por ciento comparado con un lock global único, mejora crítica para sostener cientos de conexiones concurrentes durante las horas pico de operación.

El orden de adquisición de locks está estrictamente definido para prevenir deadlocks, siguiendo una jerarquía clara documentada en el código. Primero siempre se adquiere el connection_counter_lock como lock global de máxima prioridad que controla el recurso más fundamental. Luego los user_locks en orden ascendente de user_id cuando se necesitan múltiples locks de usuario simultáneamente. Después los branch_locks también en orden ascendente de branch_id para mantener consistencia. Finalmente los locks secundarios sector_lock, session_lock y dead_connections_lock que protegen estructuras auxiliares.

Para garantizar el cumplimiento del orden de locks, el módulo lock_sequence.py implementa un context manager especializado que valida la secuencia de adquisición en tiempo de ejecución, desarrollado bajo la iniciativa ARCH-AUDIT-02. La clase LockSequence utiliza contextvars.ContextVar para rastrear el lock actualmente mantenido en cada contexto de ejecución asíncrona. Cuando se intenta adquirir un nuevo lock, el context manager compara su orden numérico definido en la enumeración LockOrder con el del lock actual. Si el nuevo lock tiene un orden menor o igual al actual, se lanza DeadlockRiskError inmediatamente con información detallada sobre ambos locks, antes de que pueda ocurrir el bloqueo que sería imposible de diagnosticar. Esta detección convierte bugs potencialmente catastróficos de deadlock en excepciones claras durante desarrollo.

El LockManager incluye además lógica de limpieza automática para evitar la acumulación infinita de locks en memoria que ocurriría si cada nueva sucursal o usuario creara un lock permanente. Se definen constantes que establecen un máximo de quinientos locks cacheados como MAX_CACHED_LOCKS, con un umbral de limpieza LOCK_CLEANUP_THRESHOLD al ochenta por ciento de capacidad que dispara la limpieza cuando se alcanza. Un ratio de histéresis LOCK_CLEANUP_HYSTERESIS_RATIO del ochenta por ciento previene el fenómeno de thrashing donde la limpieza ocurriría demasiado frecuentemente alternando entre estados de limpiar y llenar.

Capítulo 6: Ciclo de Vida de Conexiones

El módulo ConnectionLifecycle ubicado en core/connection/lifecycle.py encapsula toda la lógica de aceptación y limpieza de conexiones, proporcionando una interfaz clara para el registro y desregistro de clientes WebSocket con manejo robusto de errores y condiciones excepcionales.

El proceso de conexión sigue una secuencia cuidadosamente ordenada de verificaciones y registros diseñada para fallar rápidamente ante condiciones inválidas. Primero se verifica si el servidor está en proceso de shutdown consultando una bandera interna, rechazando conexiones nuevas durante el cierre gracioso para evitar registrar conexiones que inmediatamente serían cerradas. Luego se validan los parámetros proporcionados incluyendo los branch_ids del usuario que deben ser una lista no vacía, y para mozos, los sector_ids asignados que determinan qué eventos recibirán. La validación incluye advertencias de logging para patrones sospechosos como mozos con más de diez sectores asignados o identificadores duplicados en las listas, que podrían indicar configuración incorrecta en el sistema de administración o intentos de manipulación.

La verificación del límite global de conexiones ocurre de manera atómica bajo el connection_counter_lock para prevenir condiciones de carrera donde múltiples conexiones simultáneas podrían exceder el límite. Si el servidor está a capacidad máxima de mil conexiones, se incrementa el contador de conexiones rechazadas en las métricas para monitoreo, se registra un warning con detalles de la conexión rechazada, y se lanza una excepción

indicando la condición que el endpoint traducirá en un cierre con código apropiado. Si hay capacidad disponible, se incrementa el contador de conexiones activas antes de liberar el lock, reservando el espacio para esta conexión.

La aceptación del WebSocket se realiza llamando a websocket.accept() con un timeout configurable mediante asyncio.wait_for para prevenir conexiones colgadas que consumirían recursos indefinidamente esperando un handshake que nunca completa. El timeout por defecto es de diez segundos, suficiente para conexiones lentas pero no tanto como para desperdiciar recursos en clientes problemáticos. Si la aceptación falla por timeout, excepción de WebSocket, o cualquier otra excepción, se decrementa el contador de conexiones que fue incrementado previamente antes de propagar el error, manteniendo la consistencia del estado interno del contador.

Finalmente, el registro en índices añade la conexión a todas las estructuras de datos relevantes según los atributos del usuario autenticado. Se registra en el índice by_user con el user_id, en by_branch para cada branch_id autorizado iterando sobre la lista, en los mappings inversos _ws_to_user y _ws_to_tenant, y para mozos en by_sector para cada sector_id asignado. Para administradores se registra adicionalmente en admins_by_branch, y para personal de cocina en kitchen_by_branch.

El proceso de desconexión realiza las operaciones inversas de manera robusta, tolerando el caso donde algunos índices ya fueron limpiados por operaciones concurrentes. Se remueve la conexión de cada índice

donde fue registrada utilizando discard() en lugar de remove() para que la ausencia del elemento no cause excepción. Los mappings inversos se eliminan con pop() usando default None. El contador global se decrementa bajo el lock correspondiente. Esta robustez es necesaria porque una conexión puede ser limpiada tanto por un cierre normal iniciado por el cliente como por la tarea de limpieza de conexiones muertas, y ambos paths deben funcionar correctamente sin importar cuál ejecuta primero.

Capítulo 7: Broadcasting Paralelo y Worker Pool

El módulo ConnectionBroadcaster ubicado en core/connection/broadcaster.py representa una de las optimizaciones más significativas del Gateway, implementando un sistema de envío paralelo que reduce dramáticamente el tiempo de distribución de mensajes a grandes audiencias de cuatro segundos a aproximadamente ciento sesenta milisegundos para cuatrocientas conexiones.

El broadcaster mantiene un pool de diez workers configurables que procesan envíos de mensajes concurrentemente, implementados como tareas asíncronas que ejecutan loops infinitos consumiendo de una cola compartida. Durante el startup de la aplicación llamando al método initialize(), se crea una cola asíncrona asyncio.Queue con capacidad máxima de cinco mil tareas pendientes. Esta capacidad acotada proporciona backpressure cuando el sistema experimenta picos de carga: si los workers no pueden procesar tan rápido como llegan los envíos, la cola

eventualmente se llena y los intentos de encolar nuevas tareas esperan, propagando la presión hacia atrás en lugar de consumir memoria infinitamente.

Cada worker ejecuta un loop infinito en el método _worker que consume tareas de la cola compartida. El loop espera una tarea llamando a queue.get() con un timeout de un segundo mediante wait_for, permitiendo verificar periódicamente la bandera _running para determinar si el worker debe detenerse durante el shutdown. Cuando una tarea llega, el worker extrae los tres elementos de la tupla: el WebSocket destinatario, el payload serializado a enviar, y un asyncio.Future opcional para reportar el resultado. El envío se realiza llamando a websocket.send_text() capturando cualquier excepción. El resultado booleano indicando éxito o fallo se registra en el Future si fue proporcionado mediante set_result(), permitiendo al código llamante contar cuántos envíos fueron exitosos versus fallidos para métricas.

El shutdown del worker pool procede de manera ordenada para permitir que envíos en progreso completen. Se establece la bandera _running en False indicando que los workers deben detenerse. Luego se espera la finalización de todos los workers con asyncio.gather usando un timeout configurable de cinco segundos. Los workers que no terminen dentro del timeout porque están procesando envíos lentos se cancelan forzosamente mediante cancel() en cada tarea, asegurando que el shutdown no se bloquee indefinidamente ante conexiones problemáticas.

El broadcaster implementa una estrategia híbrida que selecciona automáticamente el método de envío óptimo

según el tamaño del broadcast, balanceando entre overhead de coordinación del worker pool y paralelismo efectivo. Para broadcasts pequeños con cincuenta conexiones o menos, el sistema utiliza el modo legacy que procesa en lotes secuenciales mediante asyncio.gather() con return_exceptions=True. Este enfoque tiene menor overhead de coordinación ya que no requiere encolar tareas y esperar Futures, siendo más eficiente para audiencias reducidas donde el paralelismo adicional no justifica el costo.

Para broadcasts grandes que superan las cincuenta conexiones, el sistema activa el modo de worker pool encolando cada envío como una tupla en la cola compartida. Cada tupla contiene el WebSocket destinatario, el payload ya serializado, y un Future creado con loop.create_future() para rastrear el resultado. Los diez workers procesan estas tareas en paralelo verdadero, distribuyendo el trabajo de envío entre múltiples contextos de ejecución simultáneos. Al finalizar, se recolectan todos los Futures para contar éxitos y fallos que se registran en métricas.

Las mediciones de rendimiento demuestran el impacto transformador de esta optimización. Un broadcast a cuatrocientas conexiones que tomaría aproximadamente cuatro segundos en modo secuencial puro se completa en alrededor de ciento sesenta milisegundos usando el worker pool, una mejora de veinticinco veces que resulta crítica para mantener la sensación de tiempo real en la interfaz de usuario donde delays perceptibles destruirían la experiencia.

Capítulo 8: Aislamiento Multi-Tenant

El TenantFilter ubicado en components/broadcast/tenant_filter.py garantiza aislamiento estricto entre diferentes restaurantes que comparten la infraestructura del Gateway. Este componente actúa como guardián que previene fugas de información entre organizaciones, un requerimiento crítico de seguridad y privacidad en un sistema multi-tenant.

El filtrado opera verificando que cada conexión candidata pertenezca al mismo tenant_id que el evento siendo distribuido. Cuando un evento especifica un tenant_id en su payload, el filtro examina cada conexión consultando el mapping inverso _ws_to_tenant y verifica que el valor coincida exactamente. Las conexiones sin tenant_id asociado en el mapping o con un tenant_id diferente se excluyen silenciosamente del broadcast sin generar errores, simplemente no reciben el mensaje. Si el evento no especifica tenant_id con valor null o ausente, el filtro permite todas las conexiones pasar, comportamiento útil para mensajes del sistema, broadcasts globales autorizados, o eventos de infraestructura que no contienen datos de negocio sensibles.

Un aspecto crítico del diseño identificado como CRIT-DEEP-02 FIX es que el filtrado por tenant ocurre dentro del lock de rama, no después de liberarlo. Esta decisión previene una condición de carrera sutil pero peligrosa que sería difícil de detectar en testing pero catastrófica en producción. Si el filtrado ocurriera después de liberar el lock, nuevas conexiones de otros tenants podrían registrarse entre

el momento de liberar el lock y el momento del envío, potencialmente recibiendo mensajes destinados a otro tenant. Al filtrar dentro del lock, se garantiza una vista consistente de las conexiones durante toda la operación de broadcast, eliminando la ventana de vulnerabilidad.

El método filter_connections implementa el filtrado recibiendo una lista de conexiones y el tenant_id objetivo. Si tenant_id es None, retorna la lista sin modificación. Si tiene valor, itera sobre las conexiones construyendo una nueva lista solo con aquellas cuyo tenant_id en el mapping coincide. El resultado se retorna para que el broadcaster proceda con el envío solo a las conexiones autorizadas.

Capítulo 9: Suscripción a Eventos Redis

El RedisSubscriber ubicado en redis_subscriber.py establece una conexión persistente con Redis y escucha eventos publicados por la API REST, actuando como puente entre el backend que genera eventos y las conexiones WebSocket que los consumen. Tras la refactorización ARCH-MODULAR-09, se transformó en un orquestador delgado de aproximadamente trescientas veintiséis líneas que coordina módulos especializados para validación, tracking de eventos perdidos, y procesamiento.

El suscriptor utiliza el patrón Pub/Sub de Redis con suscripciones por patrón mediante psubscribe, permitiendo escuchar dinámicamente todos los canales que coincidan con ciertos patrones sin necesidad de

conocer de antemano los identificadores específicos de sucursales, sectores y sesiones que varían entre instalaciones y cambian en tiempo de ejecución cuando se crean o eliminan entidades.

El sistema define cinco patrones de canales principales registrados en WSConstants. El patrón branch:*:*waiters captura eventos destinados a mozos de cualquier sucursal, como notificaciones de nuevos pedidos o llamadas de servicio. El patrón branch:*:*kitchen captura eventos para personal de cocina, incluyendo comandas nuevas y actualizaciones de estado de preparación. El patrón branch:*:*admin captura eventos para administradores y managers con visibilidad completa de la operación. El patrón sector:*:waiters permite notificaciones dirigidas a mozos de sectores específicos, útil cuando un evento solo es relevante para el personal de cierta área del restaurante. El patrón session:* captura eventos para comensales de sesiones específicas, como actualizaciones del estado de su pedido individual.

El loop principal del suscriptor mantiene una cola interna de eventos pendientes implementada como collections.deque con capacidad máxima de cinco mil elementos que proporciona backpressure natural. Cuando llega un mensaje de Redis mediante get_message(), se valida usando el módulo validator y se encola si es válido. Durante períodos de inactividad cuando get_message() retorna None porque no hay mensajes pendientes, se procesa la cola acumulada en lotes de hasta cincuenta eventos, distribuyendo la carga de procesamiento de manera eficiente sin bloquear la recepción de nuevos mensajes.

El módulo EventDropRateTracker ubicado en core/subscriber/drop_tracker.py monitorea la salud del pipeline de eventos como singleton global, generando alertas cuando la tasa de eventos perdidos supera umbrales configurados. El tracker mantiene una ventana deslizante implementada como deque de tuplas con timestamp, eventos procesados y eventos perdidos. Cuando se registra un evento perdido mediante record_drop(), el tracker calcula la tasa de pérdida sobre la ventana actual de sesenta segundos. Si esta tasa supera el umbral configurado del cinco por ciento, se genera una alerta de nivel CRITICAL con detalles sobre la tasa actual y el umbral. Un cooldown de cinco minutos entre alertas previene la saturación del sistema de logging durante problemas sostenidos que generarían alertas cada pocos segundos.

Capítulo 10: Validación de Eventos y Enrutamiento

El módulo validator.py en core/subscriber verifica que los eventos contengan los campos requeridos antes del procesamiento, actuando como primera línea de defensa contra eventos malformados, corruptos, o maliciosamente construidos que podrían causar errores en el procesamiento posterior.

El sistema define conjuntos inmutables frozensets de campos requeridos y opcionales que no pueden modificarse en runtime. Los campos requeridos incluyen type que indica el tipo de evento como ROUND_SUBMITTED o TABLE_SESSION_STARTED, tenant_id

que identifica el restaurante origen, y branch_id que identifica la sucursal específica. Los campos opcionales incluyen session_id para eventos relacionados con sesiones de mesa, sector_id para eventos que deben filtrarse por sector, table_id para eventos de mesa específica, entity para el payload de datos del evento, y otros específicos de ciertos tipos.

La función de validación validate_event_schema implementa verificación pura sin efectos secundarios, retornando una tupla de booleano indicando validez y string con información de diagnóstico. Primero verifica que el dato sea un diccionario usando isinstance, rechazando listas, strings u otros tipos con razón not_dict. Segundo verifica que todos los campos requeridos estén presentes iterando sobre REQUIRED_FIELDS y construyendo lista de faltantes, rechazando con razón missing_fields:type,branch_id si hay campos ausentes. Tercero verifica que el tipo de evento sea uno de los tipos conocidos consultando el conjunto VALID_EVENT_TYPES, rechazando con razón unknown_type:INVALID_EVENT si es desconocido.

El EventRouter ubicado en components/events/router.py centraliza la lógica de enrutamiento determinando qué conexiones deben recibir cada tipo de evento basándose en el canal de origen y el tipo de evento. El router mantiene conjuntos inmutables que clasifican eventos por su audiencia objetivo. SESSION_EVENTS contiene eventos que deben llegar a comensales de sesiones específicas como ROUND_READY o CART_ITEM_ADDED. BRANCH_WIDE_EVENTS contiene eventos que deben llegar a toda la sucursal sin filtrado de sector como ROUND_PENDING y TABLE_SESSION_STARTED,

necesarios para que cualquier mozo cercano pueda atender.

El método route_event recibe el evento validado y el canal de origen, retornando una lista de conexiones destinatarias. Extrae el tipo de evento y los identificadores relevantes del payload. Consulta el ConnectionIndex para obtener las conexiones candidatas según el tipo de canal. Aplica el TenantFilter para garantizar aislamiento multi-tenant. Para eventos de mozo, verifica si el evento está en BRANCH_WIDE_EVENTS para determinar si debe enviarse a todos los mozos o solo a los del sector especificado.

Capítulo 11: Autenticación mediante Estrategias

El Gateway implementa el patrón Strategy para soportar diferentes métodos de autenticación de manera extensible y mantenible. Las clases de estrategia residen en el módulo components/auth/strategies.py y encapsulan toda la lógica de verificación de credenciales, permitiendo agregar nuevos métodos de autenticación sin modificar código existente.

El resultado de autenticación se representa mediante la dataclass inmutable AuthResult con frozen=True y slots=True para optimización de memoria. La dataclass encapsula el éxito o fallo de la autenticación junto con información contextual. El campo success indica si la autenticación fue exitosa. El campo data contiene los datos extraídos del token para éxitos,

como claims del JWT o datos del table token. El campo error contiene el mensaje de error para fallos. El campo close_code contiene el código de cierre WebSocket apropiado para fallos. El campo audit_reason proporciona información para logging de seguridad. Los métodos de clase ok(), fail() y forbidden() proporcionan constructores convenientes para los casos comunes con valores predeterminados apropiados.

La estrategia JWTAuthStrategy maneja autenticación mediante tokens JWT para personal del restaurante incluyendo mozos, cocina y administradores. El constructor recibe la lista de roles requeridos para el endpoint específico. El método authenticate ejecuta una secuencia de verificaciones. Primero valida el origen de la conexión WebSocket contra la lista de orígenes permitidos llamando a validate_websocket_origin, rechazando conexiones de orígenes desconocidos con código FORBIDDEN y razón invalid_origin. Segundo verifica la firma y validez del JWT llamando a verify_jwt del módulo de seguridad compartido, rechazando tokens expirados o malformados con código AUTH_FAILED. La estrategia rechaza específicamente tokens que contienen el claim token_type con valor refresh, que son refresh tokens que solo deben usarse para obtener nuevos access tokens y no para autenticación directa de WebSocket. Finalmente verifica que el usuario tenga al menos uno de los roles requeridos comparando la lista de roles del token con required_roles.

La estrategia TableTokenAuthStrategy maneja autenticación mediante tokens HMAC para comensales conectados a sesiones de mesa. Estos tokens no

contienen roles ya que los comensales no tienen identidad de usuario en el sistema tradicional de gestión de personal. El método authenticate primero valida el origen de la conexión. Luego valida criptográficamente el token HMAC llamando a verify_table_token que verifica la firma usando la clave secreta configurada. El token contiene información sobre session_id, branch_id y tenant_id que se extraen y retornan en el AuthResult para uso posterior durante el registro de la conexión.

Capítulo 12: Endpoints WebSocket y Jerarquía de Clases

Los endpoints WebSocket heredan de clases base que encapsulan comportamiento común, eliminando aproximadamente trescientas líneas de código duplicado que existían cuando cada endpoint implementaba su propio ciclo de vida completo con lógica repetida de autenticación, registro, loop de mensajes y limpieza.

La clase abstracta WebSocketEndpointBase ubicada en components/endpoints/base.py define el ciclo de vida completo de un endpoint como template method. El constructor acepta el WebSocket de la conexión, el ConnectionManager para registro, y un nombre de endpoint utilizado para logging contextual y métricas. El método run() ejecuta la secuencia completa del ciclo de vida: primero llama validate_auth() para verificar credenciales, luego create_context() para construir el contexto de usuario, después register_connection() para añadir a

índices, ejecuta el loop de mensajes llamando a _message_loop(), y finalmente en bloque finally llama unregister_connection() para garantizar limpieza incluso ante excepciones inesperadas.

La clase define cuatro métodos abstractos mediante @abstractmethod que las subclases deben implementar proporcionando la lógica específica de cada tipo de endpoint. El método validate_auth() realiza la autenticación específica y retorna los datos extraídos del token como diccionario, o None si la autenticación falla indicando que la conexión debe cerrarse. El método create_context() construye un objeto WebSocketContext con la información del usuario autenticado para uso durante el ciclo de vida. El método register_connection() añade la conexión a los índices apropiados del ConnectionManager según el tipo de usuario. El método unregister_connection() realiza la limpieza inversa removiendo de índices cuando la conexión termina.

La clase JWTWebSocketEndpoint extiende la base añadiendo autenticación JWT y revalidación periódica para conexiones de larga duración. El constructor adicional acepta el token JWT como string y la lista de roles requeridos. La implementación de validate_auth() instancia JWTAuthStrategy con los roles requeridos e invoca authenticate(). El método _message_loop() sobrescrito incluye verificación periódica cada cinco minutos del intervalo JWT_REVALIDATION_INTERVAL, verificando que el token siga siendo válido consultando la lista de revocación en Redis y comprobando que no haya expirado. Si la revalidación falla porque el token fue revocado o expiró, la conexión se cierra con código AUTH_FAILED.

El WaiterEndpoint extiende JWTWebSocketEndpoint con funcionalidad específica para personal de servicio. Durante create_context() extrae información del JWT y consulta las asignaciones de sector actuales llamando a get_waiter_sector_ids() del SectorAssignmentRepository. Durante register_connection() invoca al manager indicando is_admin basado en el rol y proporcionando sector_ids. El endpoint implementa el comando especial refresh_sectors que permite actualizar asignaciones de sector sin desconectarse, útil cuando un supervisor reasigna sectores durante el turno. Al recibir el comando, revalida el JWT, consulta nuevas asignaciones, actualiza índices del manager, y envía confirmación al cliente.

El KitchenEndpoint es más simple, aceptando roles KITCHEN, MANAGER y ADMIN, registrando con is_kitchen=True para indexación en kitchen_by_branch. El AdminEndpoint proporciona visibilidad completa de sucursal registrando con is_admin=True. El DinerEndpoint difiere significativamente usando TableTokenAuthStrategy en lugar de JWT, registrando con user_id pseudo-negativo derivado de session_id para evitar colisiones, y en el índice de sesiones para recibir eventos de mesa.

Capítulo 13: Rate Limiting y Protección

El WebSocketRateLimiter ubicado en components/connection/rate_limiter.py previene que conexiones individuales abusen del sistema enviando

mensajes a tasas excesivas, ya sea por error de programación en el cliente como un loop infinito, o por intento malicioso de agotar recursos del servidor.

El limiter mantiene un diccionario que mapea cada WebSocket a una deque de timestamps de mensajes recientes con maxlen igual al límite para auto-evicción de timestamps antiguos. El método is_allowed implementa verificación en dos pasos. Primero obtiene o crea la deque para la conexión, limpiando timestamps que han salido de la ventana de tiempo de un segundo comparando con el tiempo actual. Segundo cuenta cuántos timestamps permanecen en la deque. Si el conteo es menor al límite de veinte mensajes por defecto, añade el timestamp actual y retorna True permitiendo el mensaje. Si el conteo alcanza o supera el límite, retorna False indicando rechazo.

Las conexiones que exceden el límite de veinte mensajes por segundo reciben código de cierre 4029 asignado en WSCloseCode.RATE_LIMITED, un código personalizado en el rango 4000+ que permite a los clientes distinguir entre un cierre por rate limiting versus otros tipos de cierre como autenticación o política. Un cliente bien implementado podría reconectarse después de un breve delay de backoff, mientras que uno defectuoso debería investigar por qué está generando tantos mensajes.

El módulo components/redis/lua_scripts.py implementa rate limiting atómico adicional usando scripts Lua ejecutados directamente en Redis para operaciones que requieren atomicidad garantizada entre múltiples claves o comandos. El problema con operaciones Redis

separadas como GET seguido de INCREMENT es que entre leer el contador actual y escribir el nuevo valor, otra request podría leer el mismo valor original, resultando en un conteo incorrecto que permite más requests de las autorizadas. Los scripts Lua se ejecutan atómicamente en el servidor Redis como una única operación indivisible, garantizando que toda la secuencia ocurre sin interrupciones de otras operaciones.

El script RATE_LIMIT_SCRIPT recibe como argumentos el key de Redis, el límite máximo, el tamaño de ventana en segundos, y el timestamp actual. Primero obtiene el contador actual con GET. Si el contador existe y está en el límite, retorna inmediatamente una tupla indicando rechazo junto con el conteo actual y el TTL restante para que el cliente sepa cuánto esperar. Si hay espacio disponible, incrementa el contador con INCR y establece el TTL con EXPIRE si es el primer incremento de la ventana. Finalmente retorna éxito con el nuevo conteo y TTL.

Capítulo 14: Heartbeat y Limpieza de Conexiones

El HeartbeatTracker ubicado en components/connection/heartbeat.py mantiene un registro preciso de la última actividad de cada conexión, permitiendo identificar y cerrar conexiones que han quedado inactivas por problemas de red, clientes congelados, o desconexiones no detectadas.

El tracker mantiene un diccionario _timestamps que mapea cada WebSocket al timestamp flotante de su

último heartbeat registrado. Un threading.Lock protege esta estructura para operaciones thread-safe, necesario porque algunas operaciones de limpieza pueden ejecutarse desde contextos síncronos mientras el registro de heartbeats ocurre en contextos async. El timeout por defecto se configura en sesenta segundos, significando que conexiones sin actividad reportada por más de un minuto se consideran stale y candidatas para cierre.

El protocolo de heartbeat funciona como un intercambio simple de mensajes de vida entre cliente y servidor. El cliente envía un mensaje con type igual a ping cada treinta segundos como keep-alive. El servidor al recibir este mensaje responde inmediatamente con un mensaje type pong y llama a tracker.record(websocket) para actualizar el timestamp de la conexión en el diccionario interno. La tarea heartbeat_cleanup_task ejecuta cada treinta segundos en background, llamando a tracker.get_stale_connections() para identificar conexiones cuyo último heartbeat ocurrió hace más del timeout. Las conexiones identificadas como stale se cierran con código 1001 GOING_AWAY, indicando que el servidor está terminando la conexión debido a inactividad detectada.

El ConnectionCleanup implementa un patrón de dos fases para la limpieza que evita el error RuntimeError de dictionary changed size during iteration que ocurriría si se modificara el diccionario de conexiones mientras se itera sobre él durante la limpieza. En la primera fase de snapshot, se toma una copia de las conexiones stale bajo el lock del tracker mediante list(). En la segunda fase

de cierre, se procede a cerrar cada conexión fuera del lock ya que las operaciones de I/O de red no deben bloquear otras operaciones del tracker que podrían estar registrando heartbeats de otras conexiones. En la tercera fase de desregistro, se desregistra cada conexión del ConnectionIndex con una verificación adicional double-check consultando si la conexión aún existe antes de intentar removerla, manejando el caso donde la conexión ya fue desregistrada por otra operación concurrente como un cierre normal iniciado por el cliente.

Capítulo 15: Circuit Breaker para Resiliencia Redis

El CircuitBreaker ubicado en components/resilience/circuit_breaker.py protege al Gateway contra cascadas de fallos cuando Redis experimenta problemas de conectividad, saturación, o rendimiento degradado que causarían timeouts acumulados.

El circuit breaker implementa tres estados distintos representados en la enumeración CircuitState. En estado CLOSED las operaciones proceden normalmente y el breaker simplemente monitorea contando éxitos y fallos consecutivos. Cuando el contador de fallos consecutivos alcanza el umbral configurable de cinco por defecto, el breaker transiciona a estado OPEN donde todas las operaciones se rechazan inmediatamente sin intentar contactar Redis, previniendo la acumulación de timeouts que degradarían el rendimiento general y potencialmente

agotarían recursos como threads o conexiones de pool. Después del timeout de recuperación de treinta segundos por defecto, el breaker transiciona automáticamente a estado HALF_OPEN donde permite un número limitado de operaciones de prueba para verificar si Redis se ha recuperado. Si estas pruebas tienen éxito registrado mediante record_success(), el breaker regresa a CLOSED; si fallan con record_failure(), vuelve a OPEN por otro período de timeout.

Un aspecto crítico de la implementación identificado como CRIT-AUD-01 FIX es el uso de un único threading.Lock para todas las operaciones de sincronización que modifican o leen estado interno. Las implementaciones anteriores que mezclaban asyncio.Lock para operaciones async y threading.Lock para operaciones sync experimentaban race conditions cuando ambos tipos de operaciones modificaban el mismo estado del breaker simultáneamente. El lock unificado basado en threading funciona correctamente en ambos contextos de ejecución, garantizando consistencia del estado interno sin importar desde qué tipo de código se invoque.

El método get_stats proporciona información sobre el estado actual del breaker para health checks y métricas. Este método también adquiere el lock antes de leer estado para garantizar un snapshot consistente sin lecturas parciales, identificado como MED-DEEP-01 FIX. Retorna un diccionario con el estado actual como string, el contador de fallos recientes, el número de operaciones rechazadas mientras estuvo abierto, y el timestamp de la última transición de estado.

Capítulo 16: Redis Streams para Eventos Críticos

Además del sistema Pub/Sub tradicional que proporciona entrega best-effort, el Gateway implementa un consumidor completo de Redis Streams para eventos críticos que requieren entrega garantizada como pagos y estados de pedido importantes. Esta funcionalidad desarrollada bajo la iniciativa ARCH-STREAM-01 complementa Pub/Sub proporcionando semánticas de at-least-once delivery donde los mensajes nunca se pierden aunque puedan entregarse más de una vez.

Redis Streams difiere fundamentalmente de Pub/Sub en su modelo de persistencia y entrega. Mientras Pub/Sub descarta mensajes inmediatamente si no hay suscriptores activos en el momento exacto de publicación, Streams persiste los mensajes en el servidor Redis y permite a los consumidores procesarlos a su propio ritmo, retomando desde donde quedaron después de desconexiones. Los Consumer Groups permiten que múltiples instancias del Gateway compartan el procesamiento de un stream distribuyendo mensajes entre ellas, con Redis rastreando internamente qué mensajes ha procesado cada consumidor mediante la Pending Entries List.

El StreamConsumer implementa el patrón de Consumer Group con varias características robustas de producción. Durante la inicialización en el método start(), crea el Consumer Group si no existe usando XGROUP CREATE con la opción MKSTREAM para crear

también el stream subyacente si es necesario, evitando errores en instalaciones nuevas donde los streams aún no existen. El loop principal lee lotes de mensajes usando XREADGROUP con el especificador mayor-que para recibir solo mensajes nuevos, con un timeout de bloqueo BLOCK de mil milisegundos que permite verificar periódicamente si el consumidor debe detenerse durante shutdown.

La recuperación de mensajes pendientes constituye una característica distintiva del consumidor que garantiza que ningún mensaje se pierda por fallos de instancias. Redis mantiene una Pending Entries List con mensajes que fueron entregados a consumidores pero nunca confirmados mediante XACK, típicamente porque el consumidor murió antes de completar el procesamiento exitoso. Cada treinta ciclos del loop principal, el consumidor ejecuta XAUTOCLAIM para reclamar mensajes que han estado pendientes por más de treinta segundos, presumiblemente abandonados por consumidores fallidos de otras instancias. Cada mensaje recuperado incrementa un contador de reintentos almacenado en los metadatos; aquellos que exceden tres reintentos se mueven a la Dead Letter Queue ya que probablemente tienen un problema inherente que impide su procesamiento exitoso.

La Dead Letter Queue implementada bajo RES-LOW-01 proporciona un destino para mensajes que no pueden procesarse exitosamente después de múltiples intentos. En lugar de descartar estos mensajes perdiendo datos potencialmente valiosos, el sistema los preserva en un stream separado con sufijo :dlq junto con metadatos ricos. Se almacena el stream de origen, el ID del mensaje original, el payload

serializado como JSON, la razón del fallo o excepción capturada, el contador de reintentos alcanzado, y el timestamp del momento del fallo. Este registro permite análisis posterior para identificar patrones de fallo, debugging de problemas de integración, y recuperación manual de datos si es necesario.

El backoff exponencial con jitter protege contra el thundering herd cuando múltiples consumidores intentan reconectarse simultáneamente después de una falla de Redis que afectó a todas las instancias. El delay base de un segundo se multiplica exponencialmente por dos elevado al número de intento hasta un máximo de sesenta segundos. El jitter decorrelacionado implementado en DecorrelatedJitter añade aleatoriedad eligiendo un valor uniforme entre el delay base y el delay calculado, distribuyendo los reintentos en el tiempo para evitar oleadas sincronizadas que sobrecargarían el servidor Redis en recuperación.

Capítulo 17: Métricas y Observabilidad

El MetricsCollector ubicado en components/metrics/collector.py centraliza estadísticas del Gateway con operaciones diseñadas para funcionar correctamente tanto en contextos asíncronos como síncronos sin causar deadlocks o inconsistencias.

El collector agrupa métricas en categorías representadas como dataclasses internas. BroadcastMetrics contiene total de broadcasts

enviados, cantidad con fallos parciales o totales, y cantidad rechazados por rate limiting global. ConnectionMetrics contiene rechazos por límite de capacidad alcanzado y rechazos por rate limiting individual. EventMetrics contiene total procesados exitosamente y perdidos desglosados por razón incluyendo esquema inválido, sin destinatarios, y circuit breaker abierto.

La implementación utiliza dos locks separados identificado como CRIT-WS-08 FIX. Un asyncio.Lock para operaciones asíncronas normales que constituyen la mayoría del código del Gateway. Un threading.Lock para operaciones síncronas que ocurren en ciertos paths como callbacks de bibliotecas externas o código de limpieza. Esta dualidad resuelve problemas que ocurrían cuando operaciones del mismo contador se ejecutaban desde diferentes contextos de ejecución que requerían diferentes tipos de lock. Los métodos vienen en pares: increment_broadcast_total() para uso async adquiriendo el lock async, y increment_broadcast_total_sync() para uso sync adquiriendo el lock de threading.

El método get_snapshot_sync() proporciona una vista consistente de todas las métricas en un momento dado, utilizado para health checks detallados y el endpoint de métricas Prometheus. El snapshot se captura bajo el lock sync para asegurar que todos los valores corresponden al mismo instante sin actualizaciones parciales que darían una vista inconsistente.

El PrometheusFormatter ubicado en components/metrics/prometheus.py formatea las métricas para exposición en el endpoint /ws/metrics

siguiendo la especificación de Prometheus. Cada métrica incluye líneas de comentario con HELP describiendo qué mide en lenguaje humano, y TYPE indicando si es counter para valores siempre crecientes, gauge para valores que pueden subir y bajar, o histogram para distribuciones. El formato permite labels entre llaves para dimensionar métricas, por ejemplo wsgateway_events_dropped_total con label reason igual a invalid_schema versus reason igual a no_recipients. Las métricas clave expuestas incluyen conexiones totales desde startup, conexiones activas actuales, broadcasts totales y fallidos, y estado del circuit breaker como gauge numérico.

Capítulo 18: Flujo Completo de un Evento en Práctica

Cuando un comensal confirma su pedido en la aplicación pwaMenu, se desencadena una secuencia de eventos que ilustra el funcionamiento integrado de todo el sistema desde la acción del usuario hasta la notificación en todas las pantallas relevantes.

El proceso comienza en la API REST donde el endpoint de creación de rondas valida el pedido contra el menú vigente, lo persiste en PostgreSQL con estado PENDING, y procede a notificar a las partes interesadas. La función publish_round_event construye el payload del evento y publica a múltiples canales Redis usando el pool async. Primero al canal branch seguido del branch_id y :waiters para que los mozos sepan que hay un nuevo pedido por verificar en la mesa. Luego al canal branch seguido del branch_id y

:admin para visibilidad en el Dashboard de gestión donde los administradores monitorizan la operación.

El RedisSubscriber del Gateway recibe ambos mensajes a través de su suscripción por patrón que captura todos los canales que coinciden con branch:*:waiters y branch:*:admin. Cada mensaje pasa por validación de esquema en el módulo validator que verifica la presencia de los campos requeridos type, tenant_id y branch_id, y que el tipo de evento ROUND_PENDING sea uno de los tipos válidos registrados en el conjunto VALID_EVENT_TYPES.

El EventRouter determina los destinatarios apropiados según el canal de origen parseando el nombre del canal. El mensaje del canal branch:5:waiters se rutea consultando el índice by_branch para obtener conexiones de mozos de la sucursal cinco, verificando que ROUND_PENDING está en BRANCH_WIDE_EVENTS para enviarlo a todos los mozos sin filtrado de sector. El mensaje del canal branch:5:admin se rutea a las conexiones en el índice admins_by_branch con clave cinco.

Antes del envío, el TenantFilter verifica dentro del lock de rama que cada conexión candidata pertenezca al mismo tenant_id que el evento consultando el mapping inverso, garantizando que restaurantes diferentes que comparten la infraestructura no reciban eventos ajenos. El ConnectionBroadcaster evalúa el número de destinatarios y selecciona el método de envío apropiado. Para unas pocas conexiones usa el modo legacy con asyncio.gather. Para muchas conexiones encola en el worker pool para envío paralelo.

En el Dashboard, la recepción del evento ROUND_PENDING activa la lógica de actualización visual en el store de mesas. La mesa correspondiente muestra una animación de pulso amarillo indicando orden pendiente, el badge de estado cambia a mostrar Pendiente en amarillo, y si el administrador tiene habilitadas las notificaciones de audio, suena un sonido discreto. En pwaWaiter, los mozos ven una notificación de nuevo pedido que requiere verificación física en la mesa antes de poder avanzar el estado.

Cuando el mozo verifica el pedido y el administrador lo envía a cocina con estado SUBMITTED, el KitchenEndpoint recibe el evento ROUND_SUBMITTED y lo entrega a las conexiones en kitchen_by_branch. Las terminales de cocina muestran la nueva comanda en la columna Nuevos, listando los productos a preparar con sus modificaciones y notas especiales del comensal. Este flujo completo desde que el comensal toca el botón hasta que ve la confirmación y la cocina recibe la comanda típicamente toma menos de doscientos milisegundos de transmisión a través del Gateway, creando la ilusión de actualizaciones instantáneas.

Conclusión

El WebSocket Gateway representa el sistema nervioso central del ecosistema Integrador, transmitiendo información en tiempo real entre todos los actores de un restaurante desde comensales hasta personal de cocina. Su arquitectura modular, resultado de la

refactorización ARCH-MODULAR, separó responsabilidades en componentes especializados ubicados en los directorios core y components que pueden evolucionar independientemente sin afectar la estabilidad del sistema.

Los patrones implementados reflejan decisiones arquitectónicas orientadas tanto a la mantenibilidad como al rendimiento bajo carga. El sistema de locks fragmentados mediante el LockManager reduce la contención en aproximadamente noventa por ciento durante operaciones concurrentes, permitiendo que múltiples sucursales operen con total independencia. El mecanismo de LockSequence previene deadlocks validando el orden de adquisición en tiempo de ejecución, convirtiendo bugs potencialmente catastróficos en excepciones claras durante desarrollo.

El worker pool de broadcasting con diez workers reduce el tiempo de entrega de mensajes de cuatro segundos a ciento sesenta milisegundos para broadcasts de cuatrocientas conexiones, una mejora de veinticinco veces esencial para mantener la sensación de tiempo real. El circuit breaker con lock unificado previene cascadas de fallos cuando Redis experimenta problemas, permitiendo degradación graceful en lugar de colapso total del servicio.

El patrón Strategy para autenticación mediante JWTAuthStrategy y TableTokenAuthStrategy permite extensibilidad sin modificar código existente. El patrón Template Method en las clases de endpoint elimina trescientas líneas de código duplicado. El TenantFilter garantiza aislamiento multi-tenant

filtrando dentro del lock para prevenir condiciones de carrera.

Los scripts Lua atómicos eliminan race conditions en rate limiting. El Stream Consumer con recuperación de mensajes pendientes garantiza entrega de eventos críticos incluso cuando instancias fallan. La Dead Letter Queue preserva mensajes fallidos para análisis en lugar de perderlos silenciosamente.

Esta arquitectura sustenta la experiencia fluida que los usuarios perciben al ver actualizaciones instantáneas en sus dispositivos, desde el momento que un comensal confirma su pedido hasta que el plato aparece marcado como listo en todas las pantallas del ecosistema, creando la sensación de un sistema vivo y reactivo que responde instantáneamente a cada acción.

El Dashboard: Centro de Comando del Ecosistema Gastronómico

Versión 3.1 - Febrero 2026

Prólogo: La Naturaleza del Panel de Administración

El Dashboard constituye el cerebro operativo de todo el sistema de gestión gastronómica. No se trata simplemente de una interfaz gráfica que permite visualizar datos, sino de un organismo digital vivo que respira al ritmo de las operaciones del restaurante. Cada click del administrador, cada actualización en tiempo real, cada decisión tomada desde este panel repercute instantáneamente en la cocina, en las mesas de los comensales y en los dispositivos de los mozos que recorren el salón.

Para comprender verdaderamente la magnitud de esta aplicación, debemos pensar en ella como el puente entre dos mundos: el mundo físico del restaurante con sus ollas humeantes, sus mozos apresurados y sus comensales hambrientos, y el mundo digital donde toda esa información se traduce en bits que viajan a la velocidad de la luz entre servidores, bases de datos y dispositivos móviles.

El Dashboard está construido sobre React 19.2.0, la versión más reciente del framework que introduce cambios paradigmáticos en la forma de manejar formularios mediante useActionState, hooks de optimismo con useOptimistic, y el React Compiler que proporciona memoización automática eliminando la necesidad de useMemo y useCallback en la mayoría de los casos. Utiliza Zustand 5.0.9 como gestor de estado, una biblioteca minimalista pero tremendamente poderosa que evita la complejidad ceremonial de Redux mientras mantiene la predictibilidad que las aplicaciones empresariales demandan. El bundler Vite 7.2.4 proporciona tiempos de desarrollo instantáneos y builds de producción optimizados. Y todo esto se orquesta sobre TypeScript 5.9.3 en modo estricto, convirtiendo errores de tiempo de ejecución en errores de compilación que se detectan antes de que el código llegue a producción.

El puerto de desarrollo es el 5177, la interfaz está completamente en español, y el tema visual utiliza naranja (#f97316) como color de acento sobre un fondo claro, siguiendo la identidad visual del sistema Sabor.

Capítulo 1: La Estructura Modular del Proyecto

La carpeta Dashboard organiza su código fuente en subdirectorios claramente diferenciados por responsabilidad. Esta organización refleja los principios de separación de preocupaciones y permite que diferentes aspectos de la aplicación evolucionen independientemente sin interferir entre sí.

El directorio `components` contiene treinta y dos componentes reutilizables organizados en subdirectorios temáticos. El subdirectorio `auth` contiene ProtectedRoute, un wrapper que verifica autenticación antes de renderizar rutas protegidas. El subdirectorio `layout` contiene Layout como contenedor principal con skip link para accesibilidad, Sidebar con navegación jerárquica colapsable, Header con información del usuario y logo, y PageContainer como wrapper reutilizable para páginas que garantiza consistencia de padding y márgenes.

El subdirectorio `tables` contiene componentes especializados para gestión de mesas incluyendo TableSessionModal para ver detalles de sesiones activas con comensales, rondas agrupadas por categoría e íconos de despacho, BulkTableModal para creación masiva de mesas, WaiterAssignmentModal para asignar mozos a sectores diariamente, y AddSectorDialog para agregar nuevos sectores.

El subdirectorio `ui` contiene la biblioteca de componentes de interfaz: Modal con focus trap y soporte para modales anidados, Button con estados de carga y aria-busy, Input con validación y generación automática de identificadores, Select con opciones tipadas, Textarea para texto

multilínea, Table con navegación por teclado, Badge y Card memoizados explícitamente para alto rendimiento, Toggle como switch accesible, AllergenSelect y AllergenPresenceEditor para edición de alérgenos con tipos de presencia (contains, may_contain, free_from), BranchPriceInput para precios diferenciados por sucursal, ProductSelect para selección múltiple de productos con cantidades, ImageUpload con preview y validación SSRF, Pagination con goto directo a página, ConfirmDialog con callback tipado, HelpButton para ayuda contextual, ErrorBoundary para manejo de errores de UI, Toast para notificaciones temporales limitadas a cinco simultáneas, LazyModal con imports dinámicos, CascadePreviewList para preview de eliminaciones en cascada, y TableSkeleton como placeholder de carga.

El directorio `pages` contiene veintiséis páginas con rutas definidas en el router, todas completamente funcionales. Dashboard sirve como landing con selector de sucursal y tarjetas con estadísticas. Restaurant permite configurar el tenant. Branches gestiona sucursales con horarios y zonas horarias. Tables implementa el grid de mesas con cinco estados y workflow completo incluyendo el nuevo TableSessionModal. Staff administra personal con roles por sucursal. Roles define los cuatro roles del sistema: ADMIN, MANAGER, KITCHEN, WAITER.

Orders presenta la gestión de rondas con el flujo completo de estados. Kitchen implementa la vista de cocina con dos columnas para nuevos (SUBMITTED) y en preparación (IN_KITCHEN), permitiendo que el personal de cocina vea solo los pedidos relevantes. Recipes gestiona fichas técnicas con ingredientes, pasos de preparación, y conexión con el sistema RAG para alimentar el chatbot inteligente. Ingredients administra el catálogo de ingredientes con grupos y sub-ingredientes para trazabilidad de alérgenos.

Categories y Subcategories manejan la jerarquía del catálogo scoped por sucursal. Products es la página más compleja con gestión de productos incluyendo precios por rama en centavos, alérgenos con presencia, ingredientes, métodos de cocción, perfiles de sabor y textura, y múltiples atributos dietéticos. Prices permite actualización masiva de precios por sucursal. Allergens administra alérgenos globales con reacciones cruzadas. Badges gestiona insignias promocionales. Seals administra sellos de certificación. PromotionTypes define tipos de promoción. Promotions gestiona promociones multi-sucursal con rangos de fecha y hora.

ProductExclusions configura exclusiones de categorías y subcategorías por sucursal. Settings ofrece configuración de la aplicación. Sales muestra estadísticas de ventas. HistoryBranches presenta historial por sucursal. Y la nueva HistoryCustomers implementa el tracking de fidelización de clientes de la Phase 4, mostrando dispositivos reconocidos, preferencias implícitas, y métricas de personalización.

El directorio `stores` contiene veintidós stores de Zustand con middleware de persistencia, cada uno responsable de un dominio específico de la lógica de negocio.

El directorio `hooks` contiene trece hooks personalizados que encapsulan lógica reutilizable. useFormModal elimina tres useState repetitivos gestionando estado de modal más formulario en una sola abstracción. useConfirmDialog simplifica diálogos de confirmación de eliminación. useAdminWebSocket escucha eventos ENTITY_CREATED, ENTITY_UPDATED, ENTITY_DELETED del backend para sincronizar stores. useTableWebSocket escucha eventos TABLE_*, ROUND_*, CHECK_* para actualizar mesas en tiempo real con gestión de animaciones. useWebSocketConnection gestiona la conexión global al WebSocket. usePagination implementa paginación estándar con diez items por página. useInitializeData dispara fetch inicial de todos los stores cuando Layout monta. useInitializeStaffRoles carga roles durante startup. useDocumentTitle actualiza el título de la página dinámicamente. useFocusTrap implementa focus trap para modales usando AbortController para cleanup. useOptimisticMutation facilita mutaciones optimistas con rollback automático. useKeyboardShortcuts gestiona atajos de teclado globales. useSystemTheme detecta el tema del sistema operativo.

El directorio `services` contiene tres servicios críticos. `api.ts` con más de mil cien líneas implementa el cliente REST completo con manejo de JWT, mutex para refresh de tokens que previene race conditions, AbortController para timeouts de treinta segundos, y retry automático con backoff para errores de red. `websocket.ts` con más de cuatrocientas líneas gestiona la conexión WebSocket con reconexión exponencial hasta cincuenta intentos, heartbeat bidireccional cada treinta segundos, y throttling de eventos de mesa con cien milisegundos. `cascadeService.ts` implementa eliminación en cascada con patrón snapshot/restore que permite rollback si algo falla y muestra preview de entidades afectadas antes de proceder.

El directorio `types` define contratos TypeScript para entidades, formularios, staff y roles con interfaces exhaustivas que garantizan type safety en toda la aplicación.

El directorio `utils` contiene utilidades críticas: `constants.ts` con patrones regex, límites de validación, y claves de storage con versionado para migraciones; `validation.ts` con ochocientas treinta y cinco líneas de validadores centralizados para todas las entidades; `sanitization.ts` con funciones de prevención XSS y SSRF; `permissions.ts` con treinta y dos funciones helper RBAC; `logger.ts` para logging centralizado reemplazando console directos; `helpContent.tsx` con contenido de ayuda contextual en ReactNode; `form.ts` con utilidades de formularios; `webVitals.ts` y `analytics.ts` para métricas Core Web Vitals; `performance.ts` con debounce y throttle; y `exportCsv.ts` para exportación de datos.

El directorio `config` contiene `env.ts` que valida variables de entorno. El directorio `test` contiene `setup.ts` con configuración de Vitest incluyendo mocks para matchMedia, IntersectionObserver, y cleanup automático después de cada test.

Capítulo 2: La Arquitectura del Estado Global

Los stores de Zustand forman el sistema circulatorio del Dashboard, bombeando datos hacia todos los rincones de la aplicación. Sin estos stores, cada componente sería una isla aislada, incapaz de comunicarse con los demás, repitiendo información que ya existe en otro lugar.

El sistema implementa veintidós stores especializados. Este número refleja la complejidad inherente de gestionar un restaurante moderno con múltiples sucursales, cientos de productos, decenas de empleados y miles de transacciones diarias.

Cada store sigue un patrón arquitectónico consistente. El estado incluye un array de items tipados, un booleano isLoading para indicar operaciones en progreso, un string error para mensajes de fallo, y un flag _isSubscribed para prevenir suscripciones duplicadas a WebSocket. Las acciones síncronas incluyen setItems para reemplazar el array completo, addItem para agregar un elemento, updateItem para modificar, y deleteItem para eliminar. Las acciones asíncronas incluyen fetchItems que obtiene datos del backend, createItemAsync, updateItemAsync, y deleteItemAsync que comunican con la API. Los handlers WebSocket incluyen handleWSEvent que procesa eventos entrantes y subscribeToEvents que retorna una función de cleanup para cancelar la suscripción. Finalmente, reset limpia el store durante logout.

El `authStore` merece atención especial por su complejidad. No solo almacena si el usuario está autenticado. Guarda el token JWT de acceso que expira cada quince minutos, la información del usuario incluyendo sus roles y las sucursales a las que tiene acceso, e implementa un sistema de refresco proactivo con jitter aleatorio para evitar que todos los usuarios del sistema intenten refrescar sus tokens exactamente al mismo momento. El refresh token ya no vive en el store ni en localStorage; siguiendo el fix SEC-09, se almacena exclusivamente como cookie HttpOnly que JavaScript no puede acceder, protegiéndolo de ataques XSS.

El `productStore` maneja el catálogo completo de productos con todas sus características. Cada producto tiene nombre, descripción, precio base, precios diferenciados por sucursal expresados en centavos para evitar errores de punto flotante, imágenes validadas contra ataques SSRF, alérgenos con diferentes niveles de presencia siguiendo el estándar EU 1169/2011, perfiles dietéticos con flags para vegetariano, vegano, sin gluten, sin lácteos, apto para celíacos, keto, y bajo en sodio, métodos de cocción con tiempos de preparación, perfiles de sabor y textura. El store debe mapear entre el formato de la API que usa identificadores numéricos y centavos, y el formato del frontend que usa strings y valores decimales para display.

El `tableStore` es particularmente complejo porque debe rastrear no solo el estado estático de cada mesa sino también su estado dinámico en tiempo real y gestionar animaciones visuales que alertan al operador de cambios. Una mesa puede estar libre mostrada en verde, ocupada en rojo, con pedido en diferentes estados, o con la cuenta solicitada en morado. Pero además debe saber cuántas rondas de pedidos tiene, en qué estado está cada ronda desde pending hasta served, si hay pedidos listos que necesitan entregarse mientras otros siguen cocinándose. Todo esto cambia constantemente a través de eventos WebSocket que llegan desde el gateway.

El `recipeStore` gestiona las fichas técnicas de cocina, una adición crítica que permite documentar el conocimiento culinario del establecimiento. Cada receta contiene ingredientes con cantidades y unidades, pasos de preparación ordenados, tiempos estimados, porciones, alérgenos derivados de ingredientes, y puede vincularse a productos del catálogo. Las recetas también pueden ingerirse al sistema RAG mediante el endpoint `/api/recipes/{id}/ingest`, generando embeddings vectoriales que alimentan el chatbot inteligente.

Capítulo 3: El Patrón de Selectores Estables

React 19 introdujo mejoras en la detección de cambios que, paradójicamente, pueden causar problemas si no se manejan correctamente. Cuando un componente usa un store de Zustand, React necesita saber si el valor devuelto ha cambiado para decidir si debe re-renderizar el componente. Si el store devuelve un array vacío nuevo cada vez que se llama al selector, React interpreta esto como un cambio, aunque semánticamente el valor sea el mismo.

La solución implementada es elegante en su simplicidad. En lugar de crear un array vacío nuevo cada vez, el sistema define constantes inmutables como EMPTY_PRODUCTS, EMPTY_TABLES, EMPTY_ROLES que se reutilizan. De esta manera, cuando no hay productos, el selector siempre devuelve exactamente la misma referencia en memoria, y React puede determinar que no hay cambio sin necesidad de comparar el contenido.

Este patrón se extiende a selectores más complejos. Cuando se necesita filtrar o transformar datos, se implementan cachés manuales que recuerdan el resultado anterior y solo recalculan cuando los datos de entrada realmente han cambiado. Los selectores parametrizados como selectById usan un Map para cachear funciones selectoras por parámetro, evitando crear nuevas funciones en cada render. Los selectores filtrados como selectByBranch retornan funciones que React puede optimizar con useShallow cuando es necesario prevenir re-renders por nuevas referencias de array.

La regla cardinal que todo el código respeta es nunca desestructurar el resultado de useStore. En lugar de escribir destructuración directa del store, siempre se usan selectores explícitos que extraen exactamente los datos necesarios. Esto garantiza que el componente solo re-renderice cuando los datos específicos que usa realmente cambian.

```typescript
// CORRECTO: Usar selectores
const products = useProductStore(selectProducts)
const addProduct = useProductStore((s) => s.addProduct)

// INCORRECTO: Nunca desestructurar (causa re-renders infinitos)
// const { products } = useProductStore()
```

Capítulo 4: Persistencia y Migraciones de Stores

Los stores críticos implementan persistencia en localStorage mediante el middleware persist de Zustand. Esto permite que el usuario cierre la pestaña y al volver encuentre sus datos locales intactos, mejorando la experiencia especialmente en conexiones inestables.

El sistema de migraciones maneja la evolución del schema de datos entre versiones. Cada store tiene una versión numérica definida en STORE_VERSIONS. Cuando el código evoluciona y necesita un nuevo campo o cambio de estructura, se incrementa la versión. La función migrate recibe el estado persistido y la versión anterior, aplicando transformaciones necesarias para actualizar al formato actual.

Las migraciones típicas incluyen limpiar datos mock que tenían IDs con cierto patrón, agregar nuevos campos con valores por defecto a entidades existentes, y transformar estructuras cuando el modelo cambia. Por ejemplo, cuando se agregó el sistema de presencia de alérgenos, la migración convirtió el array simple de allergen_ids al nuevo formato con objetos que incluyen allergen_id y presence_type.

La versión 4 del authStore representa un cambio significativo: el campo refreshToken fue eliminado de la persistencia siguiendo SEC-09. Las pestañas existentes que tenían

refreshToken en localStorage lo ignoran silenciosamente, y el sistema ahora depende exclusivamente de la cookie HttpOnly que el backend establece durante login.

Las versiones actuales reflejan la madurez de cada store. branchStore está en versión 5, categoryStore y subcategoryStore en versión 4, productStore en versión 6 siendo el más evolucionado por su complejidad, tableStore en versión 7 por los múltiples refinamientos del sistema de estados y animaciones, mientras stores más simples como badgeStore y sealStore permanecen en versión 1 por su estabilidad.

Capítulo 5: El Sistema de Autenticación con Cookies HttpOnly

La seguridad en el Dashboard no es un añadido superficial sino un pilar fundamental que permea cada capa de la aplicación. El sistema utiliza JSON Web Tokens para autenticar usuarios, pero la implementación ha evolucionado significativamente con el fix SEC-09 para maximizar la protección contra ataques XSS.

Cuando un usuario ingresa sus credenciales, el servidor devuelve un access token de corta vida que expira en quince minutos, diseñado para minimizar el daño si es interceptado. Pero el refresh token ya no viaja en el cuerpo de la respuesta. El backend lo establece mediante el header Set-Cookie con los siguientes atributos: HttpOnly que impide que JavaScript acceda a la cookie, Secure que en producción requiere HTTPS, SameSite=Lax que previene ataques CSRF mientras permite navegación de primer nivel, y Path=/api/auth que limita el envío de la cookie solo a endpoints de autenticación.

El access token se guarda en memoria para peticiones, pero el refresh token permanece exclusivamente en la cookie segura donde ningún script malicioso puede tocarlo. Cuando el cliente necesita refrescar el token, simplemente hace una petición POST a `/api/auth/refresh` con `credentials: 'include'`, y el navegador envía automáticamente la cookie. El servidor valida la cookie, emite un nuevo access token, y opcionalmente rota el refresh token estableciendo una nueva cookie.

El sistema implementa refresco proactivo, lo que significa que no espera a que el token expire para intentar renovarlo. Aproximadamente catorce minutos después del último refresco, el cliente inicia una solicitud de renovación. Pero si todos los usuarios refrescaran exactamente a los catorce minutos, podrían crear picos de carga en el servidor. Por eso se añade jitter, una variación aleatoria de más o menos dos minutos, distribuyendo las solicitudes de refresco en el tiempo. La función getRefreshIntervalWithJitter calcula este intervalo tomando el base de

catorce minutos, generando un número aleatorio, y sumando la variación para obtener un valor entre doce y dieciséis minutos diferente para cada usuario y cada ciclo.

Cuando el refresco falla, el sistema tiene hasta tres intentos antes de decidir que la sesión debe terminar. Esto maneja casos donde una interrupción momentánea de red podría causar un fallo temporal. Solo después de agotar los reintentos el sistema cierra la sesión y redirige al usuario a la página de login.

El mutex de refresh, implementado como fix CRIT-01, previene caos cuando múltiples peticiones fallan con 401 simultáneamente. Una variable global refreshPromise comienza como null. Cuando la primera petición detecta que necesita refresh, verifica si refreshPromise es null. Si lo es, crea una nueva Promise que representa la operación de refresh, la asigna a refreshPromise síncronamente antes de cualquier operación asíncrona, y comienza el trabajo real. Las otras peticiones, cuando detectan que necesitan refresh, ven que refreshPromise ya existe y simplemente esperan esa misma Promise en lugar de iniciar su propio refresh.

Capítulo 6: La Sincronización entre Pestañas con BroadcastChannel

Un usuario moderno frecuentemente tiene múltiples pestañas del mismo sitio abiertas simultáneamente. Esto crea desafíos: si el usuario cierra sesión en una pestaña, las otras deberían cerrar sesión también. Si se refresca el token en una pestaña, las otras no deberían intentar refrescar innecesariamente.

El Dashboard implementa esta coordinación mediante la API BroadcastChannel, un mecanismo del navegador que permite que diferentes contextos de la misma aplicación se comuniquen entre sí sin pasar por un servidor. El canal se llama `dashboard-auth-sync` y transporta tres tipos de mensajes específicos.

El mensaje `TOKEN_REFRESHED` se emite cuando una pestaña completa un refresco exitoso del access token. Las otras pestañas, al recibirlo, actualizan su token local con el nuevo valor sin necesidad de hacer su propia solicitud de refresh. Esto reduce la carga en el servidor y evita múltiples refreshes redundantes.

El mensaje `LOGOUT` se emite cuando una pestaña cierra sesión. Todas las demás pestañas ejecutan su propia limpieza local sin hacer llamadas adicionales al servidor. La función `performLocalLogout` existe específicamente para este escenario: limpia el estado local,

desconecta el WebSocket, y redirige al login sin transmitir otro LOGOUT que crearía un loop infinito.

El mensaje `LOGIN` se emite cuando una pestaña completa el proceso de login. Otras pestañas que pudieran estar mostrando la pantalla de login detectan esto y pueden actualizar su estado para reflejar que el usuario ya está autenticado.

La inicialización del BroadcastChannel incluye manejo de errores para navegadores que no soportan la API, particularmente versiones antiguas de Safari. En esos casos, el sistema funciona normalmente pero sin sincronización entre pestañas, lo cual es un degradado graceful aceptable.

Capítulo 7: La Comunicación en Tiempo Real

Si los stores son el sistema circulatorio del Dashboard, el WebSocket es su sistema nervioso. Los datos pueden fluir lentamente a través de peticiones HTTP, pero las señales de tiempo real necesitan velocidad instantánea. Cuando un comensal escanea un código QR y se sienta en una mesa, el administrador debe ver ese cambio reflejado inmediatamente.

La conexión WebSocket se establece hacia el gateway en el puerto 8001, autenticándose con el mismo token JWT que se usa para las peticiones HTTP. El servicio `websocket.ts` encapsula toda la complejidad de mantener esta conexión viva. Expone métodos simples: connect para establecer conexión pasando el token, disconnect para cerrarla limpiamente, softDisconnect para cerrarla temporalmente preservando listeners, on para registrar un callback para un tipo de evento específico, y updateToken para reconectar con un token nuevo después de un refresh.

El heartbeat bidireccional detecta conexiones muertas. Cada treinta segundos, el cliente envía un mensaje `{"type":"ping"}` y espera un `{"type":"pong"}` de respuesta del servidor dentro de diez segundos. Si el pong no llega, asume que la conexión está muerta, cierra el WebSocket forzosamente, y programa una reconexión. Esto detecta el caso donde ni siquiera llega un evento de cierre porque la conexión simplemente dejó de funcionar sin notificarlo.

Cuando la conexión se cierra inesperadamente, el servicio no intenta reconectar inmediatamente. Implementa backoff exponencial: espera un segundo para el primer intento, dos para el segundo, cuatro para el tercero, ocho para el cuarto, hasta un máximo de treinta segundos. El jitter añade hasta treinta por ciento de variación aleatoria a cada intervalo,

evitando el efecto manada donde miles de clientes reconectan exactamente al mismo momento después de una caída del servidor. El límite es de cincuenta intentos, un número alto que garantiza persistencia incluso en condiciones de red muy inestables.

No todos los cierres deben resultar en reconexión. El protocolo define códigos de cierre que el servidor puede usar para indicar la razón. El servicio mantiene un Set de códigos no recuperables definidos en NON_RECOVERABLE_CLOSE_CODES: 4001 indica AUTH_FAILED y significa token inválido o expirado, 4003 indica FORBIDDEN y significa permisos insuficientes, 4029 indica RATE_LIMITED. Cuando el servidor cierra con uno de estos códigos, el servicio no programa reconexión porque sería inútil sin intervención del usuario.

Los eventos se distribuyen mediante un sistema de listeners tipado. Los tipos incluyen TABLE_SESSION_STARTED cuando un comensal escanea el QR, TABLE_STATUS_CHANGED para cambios de estado de mesa con debounce de cien milisegundos, TABLE_CLEARED cuando la sesión termina. Para pedidos: ROUND_PENDING cuando el comensal envía un pedido, ROUND_CONFIRMED cuando el mozo lo verifica, ROUND_SUBMITTED cuando el admin lo envía a cocina, ROUND_IN_KITCHEN cuando cocina lo toma, ROUND_READY cuando está listo, ROUND_SERVED cuando el mozo lo entrega, ROUND_CANCELED si se cancela, y ROUND_ITEM_DELETED cuando se elimina un item. Para llamados de servicio: SERVICE_CALL_CREATED, SERVICE_CALL_ACKED, SERVICE_CALL_CLOSED. Para facturación: CHECK_REQUESTED, CHECK_PAID, PAYMENT_APPROVED, PAYMENT_REJECTED.

El patrón de ref resuelve un problema sutil de closures. Si el callback del listener captura estado del componente, ese estado queda congelado al momento del registro. La solución es mantener el handler actual en un ref con useRef, registrar una función que llama handlerRef.current, y actualizar el ref en cada render. El listener siempre llama al handler más reciente sin necesidad de re-registrarse.

Capítulo 8: Los Formularios con React 19

React 19 introdujo useActionState, un hook que cambia fundamentalmente cómo se manejan los formularios. En versiones anteriores, manejar un formulario requería múltiples estados separados para datos, errores, y estado de carga. Con useActionState, el formulario se trata como una máquina de estados finitos.

El hook recibe una función de acción y un estado inicial, y devuelve tres cosas: el estado actual que contiene errores y flag de éxito, una función de acción para pasar al elemento form, y un booleano isPending que indica si la acción está en progreso.

La función de acción recibe dos parámetros: el estado previo que permite implementar lógica acumulativa, y un objeto FormData que contiene todos los valores del formulario. FormData es una API nativa del navegador que el form element construye automáticamente al hacer submit, incluyendo todos los inputs con atributo name.

En el Dashboard, cada página CRUD implementa este patrón. La función submitAction es un useCallback que extrae los campos del FormData, los valida según las reglas de negocio usando los validadores centralizados de validation.ts, y retorna un nuevo estado. Si hay errores de validación, retorna un objeto con la propiedad errors mapeando nombres de campo a mensajes de error y isSuccess en false. Si la validación pasa, intenta crear o actualizar la entidad en el backend. Si tiene éxito, retorna isSuccess en true. Si falla, captura el error, lo registra con el logger centralizado, y retorna un mensaje de error general.

El componente observa el estado retornado. Cuando isSuccess se vuelve verdadero y el modal está abierto, ejecuta código para cerrar el modal. Los errores se muestran junto a cada campo correspondiente mediante la prop error que cada Input y Select acepta.

FormData es excelente para campos simples pero tiene limitaciones con estructuras complejas. La solución es híbrida. Los campos simples se leen directamente de FormData usando formData.get. Los campos complejos como arrays de alérgenos con presencia, listas de precios por sucursal, o ingredientes de recetas se mantienen en estado local del componente usando useState, y la función submitAction los lee de ese estado.

Capítulo 9: El Hook useFormModal y Abstracciones de Formulario

Manejar un modal con un formulario implica coordinar múltiples piezas de estado: si el modal está abierto, los datos del formulario, si estamos creando o editando, y cuál es el elemento siendo editado. Repetir esta lógica en cada una de las veintiséis páginas sería tedioso y propenso a errores.

El hook useFormModal encapsula toda esta lógica en una abstracción reutilizable de aproximadamente ciento treinta líneas. Acepta los datos iniciales del formulario como parámetro y retorna un objeto con todas las piezas necesarias: isOpen indica si el modal está abierto, formData contiene los datos actuales del formulario, selectedItem es el elemento siendo editado o null si estamos creando, setFormData permite actualizar campos individuales, openCreate abre el modal en modo creación con los datos iniciales, openEdit abre el modal en

modo edición con los datos del elemento existente, close cierra el modal y resetea el estado después de la animación de cierre, y reset restaura los datos iniciales sin cerrar el modal.

Las funciones openCreate y openEdit aceptan opcionalmente datos custom. openCreate puede recibir un objeto parcial que se mergeará con los datos iniciales, útil cuando queremos pre-seleccionar una categoría basándonos en el filtro actual. openEdit puede recibir datos de formulario custom en lugar de usar el elemento directamente, útil cuando el formato del elemento no coincide exactamente con el formato del formulario y necesita transformación.

La función close implementa un detalle sutil pero importante. Primero setea isOpen a false, lo que hace que el modal comience su animación de cierre. Luego programa un setTimeout de doscientos milisegundos, el tiempo típico de una animación de fade out, y solo entonces resetea formData y selectedItem. Si reseteáramos inmediatamente, el usuario vería el contenido del modal cambiar durante la animación de cierre.

El hook useConfirmDialog sigue un patrón similar más simple para diálogos de confirmación de eliminación, eliminando dos useState por cada página que lo usa y garantizando comportamiento consistente en toda la aplicación.

Capítulo 10: La Gestión de Productos

Un producto en el sistema gastronómico no es simplemente un nombre con un precio. Es una entidad rica que encapsula información nutricional, dietética, alérgenos, métodos de preparación, costos, márgenes, y disponibilidad que puede variar entre sucursales. La página de Productos refleja esta complejidad mientras mantiene una interfaz manejable.

El formulario de producto tiene más de cuarenta campos organizados en secciones lógicas. La sección básica incluye nombre obligatorio, descripción opcional, imagen validada contra SSRF mediante sanitizeImageUrl, categoría y subcategoría que definen dónde aparece en el menú. La sección de precio permite configurar un precio base único o precios diferenciados por sucursal mediante el toggle use_branch_prices. La sección de atributos incluye badges promocionales y sellos de certificación. La sección de alérgenos permite seleccionar múltiples alérgenos con diferentes tipos de presencia: contains indica que definitivamente contiene el alérgeno, may_contain indica posibilidad de contaminación cruzada, free_from certifica ausencia con precauciones en la preparación.

Debajo hay una sección colapsable de campos avanzados. El perfil dietético permite marcar vegetariano, vegano, sin gluten, sin lácteos, apto para celíacos, keto, o bajo en sodio. Los ingredientes se seleccionan del catálogo centralizado y pueden marcarse como principales o secundarios. Los métodos de cocción incluyen horneado, frito, a la parrilla, crudo, hervido, al vapor, salteado, braseado. Los tiempos de preparación y cocción se expresan en minutos. Los perfiles de sabor describen suave, intenso, dulce, salado, ácido, amargo, umami, picante. Los perfiles de textura describen crocante, cremoso, tierno, firme, esponjoso, gelatinoso, granular.

El componente BranchPriceInput maneja precios por sucursal. Por defecto, el producto tiene un precio base único. Al activar el toggle aparece una lista de todas las sucursales activas donde el administrador puede establecer precios individuales y marcar disponibilidad. Los precios se almacenan en centavos para evitar errores de punto flotante. La conversión ocurre en la frontera entre frontend y backend, multiplicando por cien al enviar y dividiendo al recibir.

La tabla de productos muestra columnas optimizadas para escaneo visual rápido. La columna de imagen muestra un thumbnail o placeholder. La columna de producto muestra nombre en negrita con descripción truncada e íconos de estrella y trending si está destacado o es popular. La columna de precio es inteligente: si usa precio único muestra ese valor, si usa precios por sucursal calcula y muestra el rango mínimo-máximo evitando redundancia cuando todos son iguales. La columna de categoría muestra badge con categoría y subcategoría. La columna de alérgenos muestra hasta tres íconos emoji con indicador de más si hay adicionales. Las columnas de acciones muestran botones condicionales según permisos.

Capítulo 11: La Gestión de Mesas en Tiempo Real

Una mesa transita por múltiples estados durante su jornada de servicio. Comienza libre disponible para nuevos comensales mostrada en verde. Cuando alguien escanea el código QR y abre una sesión pasa a ocupada mostrada en rojo. Cuando los comensales ordenan, el estado de pedido indica si hay pendiente, en cocina, o listo. Finalmente, cuando piden la cuenta entra en cuenta_solicitada mostrada en morado.

Pero el estado de la mesa es solo parte de la historia. Una mesa ocupada puede tener múltiples rondas de pedidos en diferentes estados. La primera ronda puede estar lista mientras la segunda sigue cocinándose y la tercera acaba de enviarse. El sistema necesita un estado agregado que resuma la situación para que el operador sepa qué acción tomar.

El cálculo de estado agregado en tableStore sigue reglas de prioridad cuidadosamente diseñadas. La prioridad más alta es el estado combinado `ready_with_kitchen`: si hay alguna

ronda lista y alguna que no está lista (pending, confirmed, submitted, in_kitchen), se muestra este estado especial con badge naranja que indica al mozo que hay items para llevar pero que debe volver por más. Esta situación es operativamente crítica y recibe una animación de blink naranja de cinco segundos para asegurar que no pase desapercibida.

La siguiente prioridad es `pending` si alguna ronda está pendiente de confirmación del mozo, mostrado con badge amarillo. Luego `confirmed` si todas están confirmadas pero no enviadas a cocina. Luego `submitted` o `in_kitchen` que se tratan igual como "En Cocina" con badge azul. Finalmente `ready` si todas están listas con badge verde, y `served` si todas fueron servidas con badge gris.

El Sistema de Animaciones con Maps

Las animaciones visuales que alertan al operador de cambios recientes requieren gestión cuidadosa de timeouts para evitar memory leaks y comportamiento errático. El tableStore implementa un sistema sofisticado usando Maps para rastrear timeouts activos por mesa.

El Map `blinkTimeouts` rastrea timeouts de animación de cambio de estado (blink azul de 1.5 segundos). Cuando llega un evento que cambia el estado de una mesa, el código primero verifica si ya existe un timeout activo para esa mesa. Si existe, lo cancela con clearTimeout. Luego programa un nuevo timeout, almacena su ID en el Map con el tableId como clave, y activa el flag `statusChanged` en el estado de la mesa. Cuando el timeout expira, el callback elimina la entrada del Map y desactiva el flag.

El Map `pendingStatusFetches` implementa debounce para eventos TABLE_STATUS_CHANGED. Cuando llega este evento, en lugar de hacer fetch inmediato, el código verifica si ya hay un fetch pendiente para esa mesa. Si lo hay, lo cancela. Luego programa uno nuevo con delay de cien milisegundos. Esto evita ráfagas de API calls cuando llegan múltiples eventos en rápida sucesión.

El TableSessionModal

Al clickear una mesa ocupada se abre el nuevo componente TableSessionModal que muestra los detalles completos de la sesión activa. El header muestra número de mesa, sector, y estado. La primera sección muestra los comensales registrados con sus nombres y colores asignados. La sección principal muestra las rondas de pedidos con header por ronda incluyendo número, estado con badge, y tiempo transcurrido.

Los items dentro de cada ronda se agrupan por categoría siguiendo el orden natural del servicio: primero bebidas, luego entradas, luego principales, finalmente postres. Cada categoría tiene un header colapsable y un ícono de despacho que el operador puede clickear para tracking visual de qué categorías ya fueron despachadas a cocina. Cada item muestra cantidad, nombre, precio, y el nombre del comensal que lo ordenó con cualquier nota especial en texto pequeño.

Para rondas en estado CONFIRMED, aparece un botón "Enviar a Cocina" que solo el admin o manager puede usar para transicionar la ronda a SUBMITTED. Este paso existe para dar control adicional al management sobre cuándo se liberan pedidos a cocina.

El modal se actualiza en tiempo real mediante listeners de eventos WebSocket que recargan los datos cuando llegan eventos relevantes. Al cerrar el modal, se restaura el foco al elemento que lo abrió para mantener accesibilidad de teclado.

Capítulo 12: El Personal y Sus Roles

El sistema de personal implementa un modelo de control de acceso basado en roles que determina qué puede ver y hacer cada usuario. Cuatro roles forman la jerarquía: WAITER para mozos, KITCHEN para personal de cocina, MANAGER para gerentes de sucursal, y ADMIN para administradores del sistema.

Un ADMIN tiene acceso total. Puede crear, editar y eliminar cualquier entidad en cualquier sucursal. Puede asignar cualquier rol a cualquier empleado, incluyendo crear otros administradores. Ve todas las estadísticas, todas las sucursales, todos los empleados.

Un MANAGER tiene acceso amplio pero restringido a sus sucursales asignadas. Puede crear y editar empleados, productos, categorías, mesas, pero solo en las sucursales donde tiene rol de gerente. Crucialmente, no puede asignar el rol ADMIN a nadie, preservando la jerarquía de que solo administradores crean administradores. Tampoco puede eliminar entidades, una restricción que previene pérdida accidental de datos valiosos.

Un KITCHEN puede acceder a la página de cocina donde ve los pedidos que debe preparar. Puede gestionar recetas porque ese es conocimiento de su dominio. Pero no tiene acceso a gestión de personal, mesas, ventas, ni otras áreas administrativas.

Un WAITER tiene acceso mínimo al Dashboard porque su herramienta principal es pwaWaiter. Si accede al Dashboard, puede ver información básica pero no modificar configuraciones.

La página de Personal muestra una tabla con empleados de la sucursal seleccionada. La tabla muestra nombre completo, rol traducido al español, email, teléfono, DNI, fecha de ingreso formateada según locale argentina, y estado con badge de color. El campo de búsqueda implementa debounce usando useDeferredValue de React 19 para evitar re-renders excesivos mientras el usuario escribe. La búsqueda filtra por nombre, apellido, email, o DNI.

El formulario incluye campos para nombre, apellido, email, teléfono, DNI, fecha de ingreso, estado activo, sucursal, y rol. El selector de sucursal muestra diferentes opciones según quien lo usa: para administradores todas las sucursales, para gerentes solo las sucursales donde tienen acceso. El selector de rol no muestra ADMIN si el usuario actual es gerente, previniendo escalación de privilegios desde la UI.

Las restricciones de seguridad real están en el backend. Si un gerente intentara asignar rol ADMIN manipulando el request HTTP, el backend lo rechazaría. El frontend oculta opciones que el usuario no debería usar mejorando experiencia, pero confiar solo en el frontend sería inseguro.

Capítulo 13: La Vista de Cocina

La página de Cocina presenta un diseño de dos columnas que refleja el flujo de trabajo real de una cocina profesional. La columna izquierda titulada "Nuevos" muestra pedidos en estado SUBMITTED que son pedidos que el administrador envió a cocina y están esperando que un cocinero los tome. La columna derecha titulada "En Cocina" muestra pedidos en estado IN_KITCHEN que son pedidos que algún cocinero ya está preparando.

Un detalle importante es que la cocina no ve todos los estados del pedido. Los estados PENDING y CONFIRMED son invisibles para la cocina. Esto es intencional: esos estados representan pedidos que aún no deberían prepararse porque no fueron autorizados para envío a cocina. El mozo debe verificar PENDING en la mesa, transicionando a CONFIRMED. Luego el admin o manager decide cuándo enviarlo a cocina.

Cada pedido se representa como una tarjeta compacta diseñada para escaneo rápido en el ambiente de alta presión de una cocina. El código de mesa domina visualmente porque es el identificador que el cocinero usa para saber dónde irá el pedido. Debajo aparece la cantidad

de items y el tiempo transcurrido. Un badge indica estado. Un anillo rojo envuelve la tarjeta si el pedido es urgente: más de quince minutos en SUBMITTED o más de veinte en IN_KITCHEN.

Las tarjetas son clickeables y accesibles por teclado. Al hacer click se abre un modal con el detalle completo. El modal muestra código de mesa, estado, tiempo, y cada item con cantidad, nombre, y crucialmente las notas especiales que aparecen prominentemente en rojo para que no se pasen por alto.

Al pie del modal hay un botón de acción que depende del estado. Si está SUBMITTED dice "Marcar como En Cocina". Si está IN_KITCHEN dice "Marcar como Listo". Después de transicionar a READY el pedido desaparece de la vista de cocina porque ya no es responsabilidad del cocinero; pasa a ser responsabilidad del mozo recogerlo.

Antes de renderizar cualquier contenido, la página verifica que el usuario tenga rol apropiado. La verificación usa los selectores del authStore y las funciones de permissions.ts: el usuario debe tener rol KITCHEN, MANAGER, o ADMIN. Si tiene solo WAITER se muestra una página de acceso denegado.

Capítulo 14: Las Promociones y el Tiempo

Las promociones añaden una dimensión temporal a los productos. Un Happy Hour no es simplemente un descuento; es un descuento que aplica solo entre las 17:00 y las 20:00, solo en ciertos días, solo hasta cierta fecha. El sistema debe capturar estas restricciones y aplicarlas correctamente.

Cada promoción tiene fecha de inicio y fecha de fin que delimitan su vigencia en días calendario. Pero además tiene hora de inicio y hora de fin que delimitan las horas del día en que aplica. Una promoción puede estar configurada del 1 al 31 de enero, pero solo de 18:00 a 21:00 cada día.

El estado de una promoción se calcula dinámicamente. Si is_active es false, está desactivada manualmente independientemente de fechas y horas. Si la fecha actual está fuera del rango, está inactiva por tiempo. Si la fecha está dentro pero la hora está fuera, también está inactiva. Solo cuando todas las condiciones se cumplen la promoción está activa.

Una promoción agrupa varios productos en un combo con precio especial. El componente ProductSelect permite seleccionar productos del catálogo y especificar la cantidad de cada uno. El precio de la promoción es independiente de la suma de precios de los productos individuales, permitiendo tanto ofertas donde el combo es más barato como bundles premium.

Las promociones pueden aplicar a todas las sucursales o solo a algunas. El componente BranchCheckboxes muestra todas las sucursales activas como checkboxes individuales. Esta flexibilidad permite estrategias de marketing diferenciadas por ubicación.

La validación incluye reglas temporales especiales. Al crear una promoción nueva, la fecha de inicio debe ser hoy o futura. La fecha de fin debe ser igual o posterior a la de inicio. Si inicio y fin son el mismo día, la hora de fin debe ser posterior a la de inicio. Al editar una promoción existente, las validaciones son más permisivas para permitir correcciones.

Capítulo 15: El Catálogo Jerárquico

El menú se organiza en una jerarquía de tres niveles. Las categorías son agrupaciones amplias: Bebidas, Entradas, Platos Principales, Postres. Las subcategorías son divisiones más finas dentro de cada categoría: dentro de Bebidas podríamos tener Gaseosas, Cervezas Nacionales, Cervezas Importadas, Vinos Tintos. Los productos son los items individuales que los comensales pueden ordenar.

Las categorías pertenecen a sucursales. Esto significa que cada sucursal puede tener su propia estructura de menú si es necesario, aunque en la práctica la mayoría usa la misma estructura con variaciones solo en disponibilidad de productos.

Existe una categoría especial llamada HOME que el menú público usa internamente para productos destacados en la página principal, pero que no debe aparecer en las listas administrativas. Los selectores y filtros excluyen HOME_CATEGORY_NAME de todas las vistas.

Las páginas de Categorías y Subcategorías siguen el patrón CRUD establecido. Una tabla lista las entidades existentes. Un botón Nueva abre el modal de creación. Íconos de editar y eliminar en cada fila permiten modificar o borrar. Cada categoría tiene nombre obligatorio con validación de duplicados por sucursal, orden numérico para controlar posición en el menú, ícono emoji opcional, imagen opcional, y estado activo o inactivo.

Eliminar una categoría tiene efectos en cascada. Si tiene subcategorías, eliminarla dejaría esas subcategorías huérfanas. Si esas subcategorías tienen productos, los productos también quedarían huérfanos. Antes de proceder, el componente CascadePreviewList analiza qué se eliminará y muestra un resumen al usuario con conteos por tipo de entidad. La eliminación es soft, marcando is_active como false en lugar de borrar físicamente los registros, preservando la trazabilidad para auditoría.

Capítulo 16: Ingredientes y Recetas

El sistema de ingredientes implementa una estructura de dos niveles. Los ingredientes principales son componentes directos como Tomate, Pollo, Harina. Pero algunos ingredientes marcados como procesados contienen sub-ingredientes. Salsa BBQ es un ingrediente procesado que contiene Tomate, Vinagre, Azúcar Morena, Especias, Salsa de Soja. Esta descomposición es crucial para trazabilidad de alérgenos: la soja en la salsa BBQ debe propagarse a cualquier producto que use esa salsa.

Los ingredientes se organizan en grupos para facilitar navegación: Carnes, Verduras, Condimentos, Lácteos, Granos. La página de Ingredientes muestra una lista donde cada ingrediente principal puede expandirse para revelar sus sub-ingredientes. El filtro por grupo permite encontrar ingredientes rápidamente.

Las recetas van más allá de listar ingredientes. Son documentos estructurados que capturan todo el conocimiento necesario para preparar un plato consistentemente: ingredientes con cantidades exactas y unidades, pasos de preparación con instrucciones detalladas, tiempos de preparación y cocción, porciones esperadas, notas del chef con tips profesionales, información de almacenamiento, sugerencias de presentación.

Una receta tiene más de cuarenta campos opcionales organizados en secciones. La sección de información básica incluye nombre, descripción, sucursal, categoría, subcategoría, tipo de cocina, nivel de dificultad, tiempos, y porciones. La sección de ingredientes permite agregar componentes del catálogo o escribir ingredientes ad-hoc con cantidad, unidad, y notas. La sección de preparación contiene los pasos como lista ordenada que se puede reordenar arrastrando. Las notas del chef son texto libre para trucos del oficio.

Una receta puede derivarse a un producto del catálogo. Este proceso crea un producto nuevo vinculado a la receta original. El producto hereda automáticamente los alérgenos calculados de los ingredientes de la receta, asegurando consistencia.

Las recetas también pueden ingerirse al sistema RAG mediante el botón "Ingerir" que llama al endpoint `/api/recipes/{id}/ingest`. Este proceso genera embeddings vectoriales del contenido de la receta que se almacenan en la base de datos con soporte pgvector. Estos embeddings alimentan el chatbot inteligente de pwaMenu, permitiendo que los comensales hagan preguntas sobre ingredientes, preparación, y alérgenos y reciban respuestas informadas.

Capítulo 17: La Validación Centralizada

El archivo validation.ts con ochocientas treinta y cinco líneas contiene validadores centralizados para todas las entidades del sistema. Esta centralización tiene beneficios multiplicativos: la lógica de validación está en un solo lugar, los errores se formatean consistentemente, y las reglas de negocio se documentan implícitamente.

Los validadores de números verifican diferentes condiciones: isValidNumber verifica que sea número válido no NaN, isPositiveNumber verifica que sea mayor que cero, isNonNegativeNumber verifica que sea cero o positivo. Estos helpers se usan extensivamente en validadores de entidades.

Cada validador de entidad retorna un objeto con isValid booleano y errors como mapa de campo a mensaje de error. validateProduct por ejemplo verifica que nombre tenga longitud entre 2 y 100 caracteres, que descripción no exceda 500 caracteres, que precio sea positivo si no usa precios por sucursal, que cada precio por sucursal sea positivo, que category_id exista, que URLs de imagen pasen validación SSRF. El fix DASH-008 añadió verificación de duplicados con exclusión del item siendo editado para evitar falsos positivos.

Los límites de validación están definidos en VALIDATION_LIMITS: MIN_NAME_LENGTH es 2, MAX_NAME_LENGTH es 100, MAX_DESCRIPTION_LENGTH es 500, MAX_ADDRESS_LENGTH es 200, MAX_ORDER_VALUE es 9999, MAX_CAPACITY es 999 para mesas, MAX_PRICE es 999999999 en centavos, MAX_TOASTS es 5 simultáneas.

Los validadores de tiempo verifican formato HH:mm mediante expresión regular PATTERNS.TIME. Los validadores de email verifican formato básico con PATTERNS.EMAIL. Los validadores de teléfono permiten varios formatos argentinos e internacionales con PATTERNS.PHONE. Los validadores de DNI verifican longitud y que contenga solo dígitos.

Capítulo 18: Seguridad y Sanitización

El archivo sanitization.ts contiene funciones de prevención XSS y SSRF. sanitizeImageUrl valida URLs de imagen verificando protocolo https o http, bloqueando IPs internas y rangos reservados como localhost, 127.0.0.1, rangos 10.x.x.x, 172.16-31.x.x, 192.168.x.x, y el endpoint de metadata de cloud 169.254.169.254 que atacantes usan para robar credenciales de instancias cloud. Si la URL es inválida, retorna un fallback proporcionado en lugar de lanzar error.

sanitizeHtml escapa caracteres especiales HTML como menor que, mayor que, ampersand, comillas simples y dobles. Esto previene inyección de HTML en cualquier lugar donde se muestre contenido del usuario. React ya hace esto automáticamente en JSX, pero esta función existe para casos donde se manipula HTML manualmente.

isSafeFilename verifica que nombres de archivo no contengan caracteres peligrosos como barras o puntos dobles que podrían permitir directory traversal. Un atacante que sube un archivo con nombre "../../../etc/passwd" podría escapar del directorio de uploads si el servidor no valida.

El archivo permissions.ts exporta treinta y dos funciones declarativas que encapsulan la lógica de autorización. isAdmin verifica rol ADMIN, isManager verifica MANAGER, isAdminOrManager verifica cualquiera, isKitchen verifica KITCHEN. canDelete retorna true solo si isAdmin porque managers no pueden eliminar. Las funciones específicas verifican permisos para operaciones concretas: canCreateProduct, canEditProduct, canCreateStaff, canEditStaff verifican admin o manager; canCreateBranch requiere admin. Las funciones de acceso a páginas verifican si el usuario debería poder ver ciertas secciones: canAccessKitchenPage retorna true si tiene rol KITCHEN, MANAGER, o ADMIN; canAccessRecipesPage verifica lo mismo.

Capítulo 19: Optimización de Rendimiento

El Dashboard implementa lazy loading usando React.lazy y Suspense para cada una de sus veintiséis páginas. El archivo App.tsx define cada página como un import dinámico con sintaxis `const Page = lazy(() => import('./pages/Page'))`. Esta sintaxis indica al bundler Vite que cree un chunk separado para cada página. El resultado son múltiples archivos JavaScript pequeños en lugar de uno gigante. Cuando el usuario navega a una ruta, React detecta que la página no está cargada, muestra el fallback de Suspense con un PageLoader que muestra "Cargando..." y un spinner, y una vez descargado el chunk renderiza la página.

El React Compiler está habilitado en vite.config.ts mediante el plugin babel-plugin-react-compiler con target 19. Esto proporciona memoización automática de componentes y callbacks, reduciendo la necesidad de useMemo y useCallback explícitos. Sin embargo, componentes de alto re-render como Modal usado en todas las páginas CRUD, Card usado más de cien veces, Badge usado extensivamente, y Table usado en cada página de listado mantienen memoización explícita con React.memo como medida de seguridad adicional, logrando reducciones de treinta a treinta y cinco por ciento en renders.

La optimización LCP trata la primera imagen de sucursal con atributos loading="eager", fetchPriority="high", y decoding="async" para que se descargue inmediatamente con alta prioridad. Las imágenes subsecuentes reciben loading="lazy" para que no se descarguen hasta que estén cerca del viewport.

La configuración PWA con vite-plugin-pwa implementa service worker con autoUpdate, manifest con tema naranja para instalación, y workbox con patrones de caché. Los fonts de Google se cachean con estrategia CacheFirst por un año. El build final resulta en aproximadamente 246 kilobytes totales que se comprimen a menos de 80 kilobytes gzipped.

Los chunks manuales en rollupOptions separan react-vendor con React, ReactDOM y React Router; icons con lucide-react; y state con Zustand. Esto permite caché de navegador más efectivo cuando se actualizan partes del código sin invalidar todo el bundle.

El tracking de Web Vitals mediante webVitals.ts y analytics.ts mide LCP, FID, CLS, y TTFB en producción, reportando métricas que ayudan a identificar problemas de rendimiento.

Capítulo 20: El Flujo Completo de un Pedido

Para entender cómo todas las piezas del Dashboard encajan, sigamos el viaje completo de un pedido desde que el comensal se sienta hasta que paga.

Un grupo de cuatro amigos llega al restaurante y el hostess les asigna la mesa 5 del sector Interior. Uno de ellos saca su teléfono y escanea el código QR. El código contiene una URL que abre pwaMenu con parámetros que identifican la mesa y sucursal.

pwaMenu envía una petición al backend para crear sesión. El backend crea un registro de TableSession, genera un table token con HMAC, y publica el evento TABLE_SESSION_STARTED al canal WebSocket de la sucursal.

El Dashboard, conectado vía WebSocket al endpoint /ws/admin, recibe el evento. El handler en tableStore procesa el evento, encuentra la mesa 5, y actualiza su status a ocupada. El componente de mesas detecta el cambio y re-renderiza la tarjeta con fondo rojo y animación de destello azul de 1.5 segundos.

Los cuatro amigos exploran el menú en sus teléfonos, cada uno conectado a la misma sesión compartiendo el mismo carrito. Añaden items a sus carritos que se sincronizan en tiempo real via WebSocket. Cuando están listos, uno propone enviar el pedido. Los otros confirman mediante el panel de confirmación grupal. El pedido se envía al backend.

El backend crea un Round con estado PENDING registrando qué items pidió cada comensal. Publica ROUND_PENDING a los canales de admin y waiters.

El Dashboard recibe ROUND_PENDING. El tableStore actualiza la mesa 5 agregando la ronda al diccionario roundStatuses con estado pending, recalcula orderStatus como pending, y activa el flag hasNewOrder. La tarjeta re-renderiza con pulso amarillo indicando nuevo pedido pendiente de verificación.

El mozo Alberto recibe la notificación en pwaWaiter. Se acerca a la mesa 5 y verifica el pedido con los comensales. Confirma desde pwaWaiter. El backend transiciona a CONFIRMED y publica ROUND_CONFIRMED.

El Dashboard recibe ROUND_CONFIRMED. El tableStore actualiza roundStatuses a confirmed, desactiva hasNewOrder. La tarjeta deja de pulsar amarillo y muestra badge azul "Confirmado" con destello azul.

El gerente ve que la mesa 5 tiene pedido confirmado. Hace click en la tarjeta, se abre TableSessionModal mostrando comensales, rondas agrupadas por categoría con íconos de despacho. Hace click en "Enviar a Cocina". El Dashboard envía petición POST al backend para transicionar a SUBMITTED. El backend actualiza y publica ROUND_SUBMITTED a los canales de admin, waiters, y kitchen.

El Dashboard recibe ROUND_SUBMITTED. La tarjeta muestra badge azul "En Cocina". La página de Cocina recibe el evento y el pedido aparece en la columna "Nuevos".

María la cocinera ve el nuevo pedido. Hace click, ve el detalle con todos los items y notas especiales en rojo. Hace click en "Marcar como En Cocina". El backend transiciona a IN_KITCHEN y publica ROUND_IN_KITCHEN.

El Dashboard actualiza. La página de Cocina mueve el pedido a la columna "En Cocina". Quince minutos después María hace click en "Marcar como Listo". El backend transiciona a READY y publica ROUND_READY.

El Dashboard recibe ROUND_READY. La tarjeta de mesa 5 muestra badge verde "Listo". En la página de Cocina el pedido desaparece porque ya no es responsabilidad del cocinero. Alberto recibe notificación sonora, recoge los platos de cocina, y los lleva a la mesa. Marca como servido desde pwaWaiter. El backend transiciona a SERVED y publica ROUND_SERVED.

Los amigos ordenan otra ronda de postres. El flujo se repite. Mientras la segunda ronda está en cocina, si la primera ya fue servida y hay otra pendiente, la tarjeta mostraría el estado combinado ready_with_kitchen con badge naranja y animación de cinco segundos.

Después de disfrutar su comida, los amigos piden la cuenta desde pwaMenu. El backend publica CHECK_REQUESTED. El Dashboard recibe el evento. La tarjeta cambia a fondo morado y empieza a pulsar con animate-pulse-urgent.

Alberto lleva la cuenta. Los amigos pagan en efectivo. Alberto registra el pago manual desde pwaWaiter. El backend procesa el pago y eventualmente publica TABLE_CLEARED.

El Dashboard recibe TABLE_CLEARED. El tableStore ejecuta reset completo de la mesa 5: status vuelve a libre, roundStatuses se vacía, orderStatus vuelve a none, todos los timeouts de animación se limpian. La tarjeta vuelve a verde, disponible para los próximos comensales.

El ciclo completo fue visible en tiempo real en el Dashboard. El gerente pudo monitorear el progreso sin moverse de su escritorio.

Capítulo 21: Testing y Calidad

La configuración de Vitest en vitest.config.ts establece el entorno de pruebas. El plugin de React con Babel incluye el React Compiler. El entorno es jsdom para simular browser. Los archivos de setup en test/setup.ts configuran mocks necesarios.

El setup extiende expect con matchers de jest-dom para assertions de DOM. Mockea window.matchMedia para componentes que usan media queries. Mockea IntersectionObserver para lazy loading. Ejecuta cleanup después de cada test para limpiar el DOM y prevenir fugas de estado.

Los tests verifican funcionamiento de hooks personalizados. useFormModal tiene tests que verifican inicialización con valores por defecto, apertura en modo creación, apertura en modo edición con datos del item, cierre con reset diferido, y manejo de datos custom. useConfirmDialog tiene tests que verifican apertura, cierre, y ejecución de callback.

Los tests de validación en validation.test.ts verifican que los validadores de entidades funcionen correctamente: campos requeridos, límites de longitud, formatos de email y teléfono, precios positivos, duplicados.

Los tests de utilidades de formulario en form.test.ts verifican las funciones helper que extraen y procesan errores de formulario.

El coverage reporta en texto, JSON y HTML mediante el provider v8. Excluye node_modules, directorio dist, archivos de definición de tipos .d.ts, archivos de configuración, y el directorio test. El proyecto tiene más de cien tests con tiempo de ejecución de aproximadamente 3.5 segundos.

Capítulo 22: Fidelización de Clientes y la Fase 4

La nueva página HistoryCustomers implementa el tracking de fidelización de clientes que constituye la Fase 4 del sistema de personalización. Esta funcionalidad permite que el restaurante reconozca clientes que regresan y les ofrezca experiencias personalizadas.

El sistema opera en cuatro fases progresivas. La Fase 1 implementa tracking de dispositivo: cuando un comensal escanea el QR por primera vez, pwaMenu genera un deviceId único y un deviceFingerprint que persisten en localStorage. Estas identificaciones se envían al backend con cada visita.

La Fase 2 implementa preferencias implícitas: los filtros de alérgenos, preferencias dietéticas, y métodos de cocción que el comensal selecciona se sincronizan automáticamente al backend después de dos segundos de debounce. Cuando el mismo dispositivo regresa, estas preferencias se cargan automáticamente.

La Fase 3 implementa reconocimiento: el backend puede detectar cuando un deviceId coincide con visitas anteriores y cargar el historial de preferencias. La Fase 4 implementa customer opt-in: el comensal puede registrarse voluntariamente proporcionando nombre, email, y aceptando políticas de privacidad GDPR. Una vez registrado, su historial se vincula a una entidad Customer permanente.

La página HistoryCustomers en el Dashboard muestra métricas de este sistema. Lista dispositivos reconocidos con fecha de primera y última visita. Muestra preferencias implícitas almacenadas. Para clientes registrados, muestra perfil completo, historial de visitas, y sugerencias personalizadas generadas por el sistema.

Esta funcionalidad permite estrategias de marketing segmentado: ofertas especiales para clientes frecuentes, recomendaciones basadas en pedidos anteriores, y notificaciones de nuevos platos que coinciden con sus preferencias registradas.

Epílogo: El Dashboard como Espejo del Negocio

Habiendo recorrido cada rincón del Dashboard con el nivel de detalle que merece, podemos apreciar cómo esta aplicación no es simplemente código que muestra datos. Es un modelo digital del restaurante físico, diseñado para que los operadores puedan gestionar su negocio efectivamente desde cualquier lugar con conexión a internet.

El catálogo de productos modela la oferta culinaria con toda su complejidad: precios diferenciados por sucursal que reflejan realidades económicas distintas, alérgenos con niveles de presencia que protegen la salud de los comensales, perfiles dietéticos que permiten filtrado avanzado, vinculación con recetas que preserva conocimiento institucional.

El sistema de mesas modela el servicio de salón con estados que reflejan el flujo real de una comida: mesa libre esperando comensales, ocupada cuando llegan, múltiples rondas de pedidos que pueden estar en distintos estados simultáneamente, cuenta solicitada cuando terminan. Las animaciones comunican urgencia sin requerir atención constante. El estado

combinado ready_with_kitchen asegura que situaciones operativamente críticas no pasen desapercibidas.

El sistema de personal modela la organización humana con roles que reflejan jerarquías reales y permisos que implementan políticas de acceso coherentes. La asignación de mozos a sectores permite que cada empleado se enfoque en su zona sin ruido de otras áreas.

La comunicación en tiempo real modela la inmediatez que el servicio de restaurante requiere. Cuando algo cambia, todos los actores relevantes lo saben instantáneamente. El mozo sabe que llegó un pedido. La cocina sabe qué preparar. El gerente tiene visibilidad total. El sistema de fidelización reconoce clientes que regresan.

La autenticación con cookies HttpOnly, el BroadcastChannel para sincronización entre pestañas, el mutex de refresh para prevenir race conditions: estos detalles técnicos modelan la confianza y seguridad que una organización requiere de sus sistemas.

Y quizás esa es la lección más importante de esta exploración: el mejor software no es el que tiene la arquitectura más elegante o el código más conciso, sino el que modela fielmente el dominio para el que fue creado y sirve genuinamente a las personas que lo usan. El Dashboard del sistema Integrador es software bien hecho no porque use React 19.2 o Zustand 5.0.9, sino porque entiende profundamente qué es un restaurante, cómo funciona, y qué necesitan las personas que lo operan.

Cada store existe porque hay un aspecto del negocio que necesita rastrearse. Cada página existe porque hay una tarea que necesita realizarse. Cada evento WebSocket existe porque hay información que necesita fluir en tiempo real. Cada permiso existe porque hay decisiones sobre quién puede hacer qué. La tecnología está al servicio del negocio, no al revés.

En última instancia, cuando el gerente mira el Dashboard y ve un mosaico de mesas verdes, rojas, y moradas con badges de colores y animaciones pulsantes, está viendo el pulso de su restaurante traducido a luz. Cuando hace click para enviar un pedido a cocina, está moviendo átomos en el mundo físico a través de bits en el mundo digital. Cuando revisa las métricas de fidelización, está convirtiendo miles de interacciones individuales en entendimiento accionable.

El Dashboard es, en el sentido más profundo, un puente entre mundos. Y construir puentes que soporten el peso del uso real, día tras día, es de lo que se trata la ingeniería de software profesional.

Documento técnico narrativo del proyecto Dashboard. Última actualización: Febrero 2026.

pwaWaiter: El Corazón Móvil del Servicio de Mesa

Introducción: El Problema que Resuelve pwaWaiter

En el ecosistema de un restaurante moderno, el personal de servicio enfrenta un desafío fundamental: mantener consciencia situacional constante de múltiples mesas simultáneamente, cada una en diferentes etapas del ciclo de servicio. Un mozo experimentado desarrolla una especie de radar mental que le permite saber cuándo una mesa necesita atención, cuándo un pedido está listo en cocina, y cuándo un cliente está esperando la cuenta. Sin embargo, este "radar" tiene limitaciones físicas evidentes: el mozo no puede estar en todas partes al mismo tiempo, y la información que percibe está limitada por su campo visual y auditivo.

pwaWaiter nace como una extensión digital de este radar natural. La aplicación transforma el dispositivo móvil del mozo en un centro de comando personal que concentra toda la información relevante de sus mesas asignadas en tiempo real. Cuando un cliente escanea un código QR y realiza un pedido desde pwaMenu, esa información viaja instantáneamente al dispositivo del mozo correspondiente. Cuando la cocina termina de preparar un plato, el mozo recibe una notificación sonora incluso si está en el extremo opuesto del salón. Cuando un comensal presiona el

botón de llamado en su mesa, el mozo lo sabe de inmediato.

La aplicación está construida como una Progressive Web App utilizando React 19.2.0, Zustand 5 para gestión de estado, TypeScript estricto, y Vite 7 como bundler. Esta arquitectura permite que la aplicación se instale en cualquier dispositivo móvil sin necesidad de pasar por tiendas de aplicaciones. Un nuevo mozo puede comenzar a usar la aplicación en segundos, simplemente navegando a una URL y añadiendo el acceso directo a su pantalla de inicio. La decisión de construir una PWA no es casual: los restaurantes rotan personal frecuentemente, y la fricción de instalación debe ser mínima.

El puerto de desarrollo es el 5178, la interfaz está completamente en español, y el tema visual utiliza naranja (#f97316) como color de acento sobre un fondo claro, con botones rectangulares que evitan deliberadamente las esquinas redondeadas para proyectar una estética profesional y directa.

Capítulo 1: La Cascada de Autenticación en Cinco Pasos

El Paradigma de Estado Derivado

La arquitectura de pwaWaiter se fundamenta en un principio que permea todo el diseño de la aplicación: el estado de la interfaz debe derivarse del estado de los datos, nunca sincronizarse manualmente con él. Este principio, conocido en la comunidad React como

"derived state", elimina una categoría completa de bugs relacionados con inconsistencias entre lo que el usuario ve y lo que realmente está sucediendo en el sistema.

El archivo `App.tsx` no contiene lógica de navegación tradicional con rutas y redirects. En su lugar, implementa una función `renderContent()` que examina el estado actual de la aplicación y retorna el componente apropiado. Esta cascada de condiciones puede parecer verbosa comparada con un router tradicional, pero ofrece garantías que ningún sistema de rutas puede proporcionar. Es imposible que el usuario vea el grid de mesas sin haber pasado por cada uno de los pasos previos. Es imposible que quede atrapado en un estado inconsistente donde la UI muestra una cosa pero los datos dicen otra.

Paso 1: Selección de Sucursal Previa al Login

El flujo de autenticación comienza antes siquiera de pedir credenciales. El componente `PreLoginBranchSelect.tsx` presenta una lista de sucursales activas obtenida del endpoint público `/api/public/branches`, que no requiere autenticación. Esta decisión arquitectónica permite que el mozo seleccione dónde trabajará hoy sin necesidad de identificarse primero.

La lista de sucursales se carga al montar el componente mediante un efecto que maneja estados de carga y error con indicadores visuales apropiados. Cada sucursal se presenta como un botón grande con el nombre prominente, optimizado para selección táctil con una sola mano. Al seleccionar una sucursal, el

sistema almacena tanto el ID como el nombre en el `authStore` mediante las propiedades `preLoginBranchId` y `preLoginBranchName`, y la interfaz transiciona automáticamente al paso siguiente.

Un botón "Cambiar" permanece visible durante todo el flujo posterior, permitiendo que el mozo que seleccionó la sucursal equivocada pueda corregir sin necesidad de cerrar sesión. Este botón limpia las propiedades de sucursal pre-login y retorna al paso inicial.

Paso 2: Autenticación con Credenciales

Con la sucursal ya seleccionada, aparece el componente `Login.tsx` con el formulario tradicional de email y contraseña. La pantalla muestra prominentemente el nombre de la sucursal seleccionada para que el mozo confirme que está ingresando al lugar correcto.

El envío del formulario invoca `authAPI.login()`, que realiza una petición POST al endpoint `/api/auth/login`. Si las credenciales son válidas, el servidor retorna un token JWT de acceso con quince minutos de vida útil. El refresh token, siguiendo la especificación SEC-09 del proyecto, ya no se almacena en localStorage como en versiones anteriores, sino que el servidor lo configura automáticamente como una cookie HttpOnly con el atributo `secure` en producción y `samesite=lax` para protección contra CSRF.

Esta migración a cookies HttpOnly representa una mejora significativa de seguridad. Las cookies HttpOnly no son accesibles desde JavaScript, lo que significa que incluso si un atacante logra inyectar código malicioso en la página mediante un ataque XSS, no puede robar el refresh token. Desde la perspectiva del código frontend, esto simplifica las cosas: la aplicación simplemente incluye `credentials: 'include'` en las solicitudes de refresh, y el navegador se encarga de enviar la cookie automáticamente.

Tras el login exitoso, el store verifica que el usuario tenga el rol WAITER o ADMIN. Un usuario con solo rol KITCHEN, por ejemplo, sería rechazado con un mensaje explicativo. Simultáneamente, se inicia el intervalo de renovación de token y se establece la conexión WebSocket con el servidor.

Paso 3: Verificación de Asignación Diaria

El login exitoso no otorga acceso inmediato a la aplicación. En el código de `App.tsx`, después de confirmar que existe un token válido y una sucursal seleccionada, el sistema realiza una verificación de asignación llamando al endpoint `/api/waiter/verify-branch-assignment?branch_id={id}`.

Este endpoint consulta la tabla de asignaciones y verifica que el mozo autenticado tenga una asignación activa para la sucursal seleccionada en la fecha actual. La respuesta incluye un booleano `is_assigned`, un mensaje descriptivo, el nombre de la sucursal, y la lista de sectores asignados si corresponde.

Si el mozo no tiene asignación para esa sucursal hoy, la aplicación muestra una pantalla de "Sin Asignación para Hoy" con el mensaje del servidor y un botón "Elegir otra sucursal" que retorna al paso 1. Este diseño tiene una consecuencia importante: un mozo con credenciales válidas puede ser rechazado si intenta acceder a una sucursal donde no está asignado. Esto no es un bug sino una característica. Garantiza que el administrador del restaurante mantiene control total sobre quién ve qué mesas, y que ese control se refleja en tiempo real en la aplicación.

Si la verificación es exitosa, el sistema establece `assignmentVerified: true` en el store y copia el ID de sucursal pre-login a `selectedBranchId`, que es el campo canónico usado por el resto de la aplicación.

Pasos 4 y 5: La Interfaz Principal con Dos Pestañas

Una vez verificada la asignación, el componente `MainPage.tsx` toma el control. Esta página presenta una interfaz con dos pestañas en el header: "Comensales" (la vista por defecto) y "Autogestión".

La pestaña Comensales renderiza el componente `TableGrid.tsx`, que muestra todas las mesas asignadas al mozo organizadas por sector. La pestaña Autogestión abre el componente `AutogestionModal.tsx`, un modal de pantalla completa para el flujo de toma de pedidos gestionados directamente por el mozo, útil para clientes que prefieren no usar sus teléfonos.

La página principal también inicializa varios sistemas críticos: el listener de eventos WebSocket mediante `subscribeToEvents()`, el intervalo de actualización periódica de mesas configurado en `UI_CONFIG.TABLE_REFRESH_INTERVAL`, y el gesto de pull-to-refresh para actualización manual. Un banner de conexión en la parte superior indica el estado del WebSocket, y un banner de offline aparece cuando el dispositivo pierde conectividad de red.

Capítulo 2: La Jerarquía de Stores y el Patrón de Selectores

La Gestión de Estado con Zustand 5

La gestión de estado en pwaWaiter utiliza Zustand en su versión 5.0.9, una biblioteca que se distingue por su simplicidad conceptual. A diferencia de Redux, que requiere acciones, reducers y middleware, Zustand permite definir el estado y sus mutaciones en un solo lugar, con tipado completo de TypeScript y sin boilerplate.

La aplicación organiza su estado en cuatro stores especializados, cada uno responsable de un dominio específico y con su propio archivo de tests en Vitest. Estos stores no operan en aislamiento: el `tablesStore` consume el token del `authStore` para autenticar sus llamadas API, cuando una operación falla puede delegarla al `retryQueueStore`, y cuando una acción se completa exitosamente puede registrarla en el `historyStore`. Esta orquestación refleja la

realidad de que el estado de una aplicación no es monolítico sino interconectado.

El Patrón Crítico de Selectores en React 19

El patrón de selectores es fundamental y su violación constituye uno de los errores más comunes que causan loops infinitos de re-renderizado. La regla es simple pero inflexible: nunca desestructurar el store, siempre usar selectores.

```typescript
// CORRECTO: Usar selectores
const user = useAuthStore(selectUser)
const tables = useTablesStore(selectTables)
const fetchTables = useTablesStore((s) =>
s.fetchTables)

// INCORRECTO: Nunca desestructurar (causa re-renders
infinitos)
// const { user } = useAuthStore()
// const { tables } = useTablesStore()
```

Esta restricción existe porque la desestructuración crea nuevas referencias en cada render, lo que Zustand interpreta como un cambio de estado, disparando un nuevo render, que crea nuevas referencias, ad infinitum. Los selectores, en cambio, retornan referencias estables que solo cambian cuando el dato subyacente realmente cambia.

Para selectores que retornan arrays potencialmente vacíos, el código implementa un patrón adicional de arrays estables. En lugar de retornar un nuevo array

vacío cada vez (que sería una nueva referencia), los selectores mantienen una constante `EMPTY_TABLES` o `EMPTY_ARRAY` definida fuera del store que se retorna siempre que el resultado estaría vacío. Este detalle aparentemente menor previene cientos de re-renders innecesarios en aplicaciones con listas dinámicas.

Los selectores que filtran datos implementan además un patrón de caché manual. El selector `selectTablesWithPendingRounds`, por ejemplo, mantiene una variable de caché en el scope del módulo que almacena tanto el array de entrada como el resultado filtrado. Si el array de entrada no ha cambiado (comparación por referencia), retorna el resultado cacheado sin recalcular. Este patrón, documentado como fix WAITER-STORE-CRIT-01, eliminó una fuente significativa de problemas de rendimiento.

El authStore: Identidad y Seguridad

El `authStore.ts` con sus más de 450 líneas gestiona todo lo relacionado con la identidad del usuario: credenciales, tokens de acceso, sucursal seleccionada, y estado de verificación de asignación. Su estado incluye `user`, `token`, `selectedBranchId`, `selectedBranchName`, las variantes `preLogin` de sucursal, `assignmentVerified`, y flags de control como `isLoading`, `error`, `isRefreshing`, y `refreshAttempts`.

La renovación de tokens implementa un mecanismo proactivo: cada catorce minutos, un minuto antes de que el token expire, la aplicación solicita automáticamente un nuevo token al servidor. Esta

renovación ocurre en segundo plano, invisible para el mozo, que puede continuar trabajando sin interrupciones. El flag `isRefreshing`, introducido como fix HIGH-29-18, previene una condición de carrera donde múltiples componentes podrían disparar refresh simultáneos. Si un refresh ya está en progreso, las solicitudes subsiguientes esperan su resultado en lugar de iniciar nuevos intentos.

El contador `refreshAttempts` implementa el fix WAITER-CRIT-01: después de tres intentos fallidos consecutivos de refresh, el sistema ejecuta logout automático bajo la premisa de que algo está fundamentalmente mal con la sesión y continuar reintentando sería inútil. Este límite previene loops infinitos de refresh que consumirían recursos y confundirían al usuario.

El store también expone `setTokenRefreshCallback()`, un mecanismo para que el servicio WebSocket pueda registrar un callback que será invocado cuando el token se renueve. Esto permite que la conexión WebSocket actualice su token sin necesidad de reconexión completa.

La persistencia utiliza el middleware `persist` de Zustand con localStorage y un campo `partialize` que selecciona explícitamente qué campos persistir: `token`, `user`, `selectedBranchId`, y `availableBranches`. El refresh token ya no se persiste en localStorage siguiendo SEC-09.

El tablesStore: El Corazón de la Aplicación

El `tablesStore.ts` con más de 900 líneas es el verdadero centro neurálgico de la aplicación. Mantiene el estado de todas las mesas visibles para el mozo y procesa los eventos en tiempo real que llegan desde el servidor.

El estado principal es el array `tables` de tipo `TableCard[]`, obtenido del endpoint `/api/waiter/tables?branch_id={id}`. Cada `TableCard` contiene información esencial: `table_id`, `code` (el identificador alfanumérico como "INT-01"), `status` (FREE, ACTIVE, PAYING, OUT_OF_SERVICE), `session_id` si hay sesión activa, `open_rounds` contando rondas pendientes de servir, `pending_calls` contando llamados de servicio activos, `check_status` para el estado de la cuenta, y campos de sector (`sector_id`, `sector_name`) para agrupación visual.

Además de los campos que vienen del servidor, cada tabla mantiene estado local para animaciones: `orderStatus` (el estado agregado de todas las rondas), `roundStatuses` (un Record que mapea IDs de ronda a sus estados individuales), y flags booleanos `statusChanged`, `hasNewOrder`, y `hasServiceCall` que controlan las animaciones visuales.

El store también gestiona `activeSession` de tipo `WaiterSessionSummary`, que almacena el resumen de la sesión activa cuando el mozo está en modo de gestión directa (Autogestión). Este campo se puebla mediante `fetchSessionSummary()` y se usa para mostrar información en el modal de pedidos.

El Cálculo del Estado Agregado de Pedidos

Una mesa puede tener múltiples rondas de pedidos simultáneamente, cada una en un estado diferente del ciclo de vida. La primera ronda podría estar siendo servida mientras la segunda aún está en cocina y la tercera acaba de ser confirmada. Mostrar el estado de cada ronda individualmente en la tarjeta de mesa sería confuso, así que el store calcula un estado agregado mediante una lógica de priorización específica.

El estado `ready_with_kitchen` es particularmente importante. Aparece cuando al menos una ronda está lista para servir (`READY`) pero otras rondas aún están en proceso (pendientes, confirmadas, o en cocina). Este estado combinado, renderizado con un badge naranja y una animación de blink de cinco segundos, alerta al mozo de que hay items para recoger de cocina pero que habrá más por venir.

La prioridad completa de estados es: `ready_with_kitchen` (naranja) > `pending` (amarillo, requiere confirmación del mozo) > `confirmed` (azul, verificado pero no enviado a cocina) > `submitted`/`in_kitchen` (azul, en preparación) > `ready` (verde, todo listo) > `served` (gris, completado). El estado `none` indica ausencia de rondas activas.

El retryQueueStore: Resiliencia Offline

El `retryQueueStore.ts` con aproximadamente 200 líneas implementa un patrón crucial para aplicaciones móviles: la cola de trabajo persistente. Cuando una operación falla debido a problemas de conectividad,

se encola para reintento automático cuando la conexión se restablezca.

La cola soporta acciones específicas: `MARK_ROUND_SERVED`, `ACK_SERVICE_CALL`, `RESOLVE_SERVICE_CALL`, `CLEAR_TABLE`, y `SUBMIT_ROUND`. Cada acción encolada tiene un identificador único, el tipo de acción, el payload de datos, un timestamp de creación, y un contador de reintentos.

La deduplicación opera por combinación de tipo y ID de entidad. Si el mozo, frustrado por la falta de respuesta, pulsa el mismo botón varias veces, solo la primera acción se encola. Las subsiguientes se descartan silenciosamente al detectar que ya existe una acción del mismo tipo para la misma entidad.

El procesamiento de la cola ocurre automáticamente cuando el dispositivo detecta reconexión mediante el evento `online` del navegador. Las acciones se procesan en orden FIFO con un debounce de 100ms para evitar ráfagas. Si una acción falla con error de red, se incrementa su contador de reintentos y se mantiene para el próximo intento. Si falla con error de negocio (404, 403), se elimina porque reintentar sería inútil. Después de tres reintentos fallidos, la acción se descarta bajo la premisa de que algo está fundamentalmente mal.

El fix WAITER-STORE-CRIT-02 asegura que el listener del evento `online` se limpie correctamente cuando el store se desmonta, exportando una función `cleanupOnlineListener()` que los tests pueden invocar para evitar listeners huérfanos.

El historyStore: Auditoría y Sincronización

El `historyStore.ts` con aproximadamente 190 líneas mantiene un registro de las últimas acciones realizadas por el mozo, útil tanto para auditoría como para debugging. Las entradas incluyen tipos como `ROUND_MARKED_SERVED`, `SERVICE_CALL_ACCEPTED`, y `SERVICE_CALL_COMPLETED`, junto con timestamps y metadatos relevantes.

La característica distintiva de este store es su sincronización entre pestañas mediante la API BroadcastChannel. Cuando el mozo tiene la aplicación abierta en múltiples pestañas o dispositivos, cada acción registrada se transmite a través del canal. Cualquier otra pestaña escuchando en el mismo canal recibe el mensaje y actualiza su estado.

El nombre del canal está definido en la constante `BROADCAST_CHANNEL_NAME = 'waiter-history-sync'`. La implementación incluye salvaguardas: verifica disponibilidad de la API antes de crear el canal, cierra el canal explícitamente en logout mediante `closeBroadcastChannel()` (fix CRIT-10 y QA-WAITER-HIGH-01), y mantiene la referencia al canal si el cierre falla para poder reintentar después (fix LOW-29-15). Un mount guard previene actualizaciones de estado después de que el componente se desmonta (fix WAITER-STORE-HIGH-02).

El historial está limitado a 100 entradas en modo FIFO para prevenir crecimiento ilimitado de memoria.

Capítulo 3: La Capa de Servicios y sus Protecciones

El Cliente API y la Defensa contra SSRF

El archivo `api.ts` con más de 400 líneas es mucho más que un simple wrapper alrededor de fetch. Es una capa de abstracción que implementa múltiples patrones de seguridad y resiliencia.

La primera línea de defensa es contra ataques SSRF (Server-Side Request Forgery). Aunque SSRF es típicamente un problema de servidores, un cliente web maliciosamente manipulado podría intentar hacer que el navegador del mozo realice requests a URLs internas. La configuración `API_CONFIG` define `ALLOWED_HOSTS` (localhost, 127.0.0.1) y `ALLOWED_PORTS` (80, 443, 8000, 8001, 5176, 5177, 5178), y toda URL base es validada contra estos valores antes de ejecutar cualquier request.

En producción, la validación es más estricta: bloquea acceso directo a direcciones IP (tanto IPv4 como IPv6), requiere match exacto de hostname sin wildcards de subdominios, y rechaza cualquier intento de alcanzar rangos de IP privados o servicios de metadata de cloud como 169.254.169.254. Si la validación falla, el sistema lanza una excepción inmediatamente, antes de que el request siquiera se intente.

Timeouts y Control de Señales

El fix WAITER-SVC-MED-01 introdujo un sistema de timeout robusto basado en AbortController. Cada request tiene un timeout configurable (por defecto 30 segundos en `API_CONFIG.TIMEOUT`) después del cual es abortado automáticamente. La implementación es sofisticada: combina una señal externa opcional (para que el código que llama pueda cancelar) con una señal interna de timeout, usando `AbortSignal.any()` cuando está disponible o una implementación manual de fallback.

Esto previene que la aplicación quede colgada indefinidamente esperando respuesta de un servidor que no responde, y también permite que navegación o desmontaje de componentes cancelen requests en vuelo que ya no son necesarios.

Las APIs Especializadas por Dominio

El archivo exporta múltiples objetos API agrupados por responsabilidad. El objeto `authAPI` agrupa login, getMe, refresh, y logout. Notablemente, el método `logout` pasa `false` como tercer argumento a `fetchAPI`, deshabilitando el retry automático en 401. Este detalle es crucial: si el token ya expiró y logout retorna 401, reintentar dispararía el callback de token expirado que llamaría logout nuevamente, creando un loop infinito. El fix está documentado como patrón anti-loop de logout.

El objeto `tablesAPI` agrupa las operaciones de consulta: `getTables(branchId)` retorna todas las mesas filtradas por los sectores asignados al mozo, `getTable(id)` retorna una mesa específica, y `getTableSessionDetail(id)` retorna el detalle

completo de la sesión activa incluyendo diners, rondas con items, y estado de cuenta.

El objeto `roundsAPI` agrupa las operaciones sobre pedidos: `confirmRound(roundId)` para verificación del mozo, `markAsServed(roundId)` para marcar como entregado, `deleteItem(roundId, itemId)` para eliminar items de rondas no enviadas a cocina.

El objeto `serviceCallsAPI` proporciona `acknowledge(callId)` para indicar que el mozo vio el llamado, y `resolve(callId)` para marcarlo como atendido.

El objeto `waiterTableAPI` agrupa el flujo de mesa gestionada: `activateTable(tableId, {diner_count})` crea una sesión para una mesa libre, `submitRound(sessionId, {items})` envía un pedido, `requestCheck(sessionId)` solicita la cuenta, `registerManualPayment({check_id, amount_cents, manual_method})` registra pagos en efectivo u otros métodos, y `closeTable(tableId)` cierra la sesión.

El objeto `comandaAPI` proporciona `getMenuCompact(branchId)`, un endpoint optimizado que retorna el menú sin imágenes para la toma rápida de pedidos. Esta optimización reduce significativamente el tiempo de carga y el consumo de datos.

El objeto `billingAPI` agrupa `confirmCashPayment(checkId, amount)` y `clearTable(tableId)`.

Finalmente, `publicAPI.getBranches()` obtiene la lista de sucursales sin requerir autenticación, usado en el paso 1 del flujo de login.

El Servicio WebSocket y su Resiliencia

La clase `WebSocketService` en `websocket.ts` con más de 400 líneas encapsula toda la complejidad de mantener una conexión WebSocket confiable. La URL de conexión es `/ws/waiter?token={token}` en el servidor de WebSocket (por defecto `ws://localhost:8001`).

El mecanismo de heartbeat envía un mensaje `{"type":"ping"}` cada 30 segundos (configurable en `WS_CONFIG.HEARTBEAT_INTERVAL`). Si el pong no llega en 10 segundos (`HEARTBEAT_TIMEOUT`), el servicio asume que la conexión está muerta y la cierra explícitamente. Esta detección proactiva es necesaria porque los WebSockets no siempre detectan automáticamente cuando la conexión se ha perdido, especialmente en redes móviles donde el dispositivo puede mantener una conexión aparentemente abierta que en realidad ya no funciona.

La reconexión automática implementa backoff exponencial con jitter. El primer reintento espera 1 segundo (`RECONNECT_INTERVAL`), el segundo espera 2 segundos más un componente aleatorio, el tercero espera 4 segundos más jitter, hasta un máximo de 30 segundos (`MAX_DELAY`). El jitter es crucial en producción: si el servidor se reinicia y cien clientes intentan reconectarse exactamente al mismo tiempo, el servidor puede sobrecargarse. Con jitter, los reintentos se distribuyen en el tiempo.

El límite de intentos está configurado en 50 (`MAX_RECONNECT_ATTEMPTS`). El fix RES-MED-01 agregó un callback `setOnMaxReconnectReached()` que notifica a la aplicación cuando se alcanza este límite, permitiendo mostrar un mensaje al usuario sugiriendo recargar la página.

Ciertos códigos de cierre indican errores no recuperables que no ameritan reintento. El código 4001 indica fallo de autenticación (token expirado o revocado), 4003 indica permisos insuficientes, y 4029 indica que se excedió el rate limit. Para estos códigos, definidos en `WS_CONFIG.NON_RECOVERABLE_CLOSE_CODES`, el servicio no programa más reintentos y notifica a la aplicación.

El fix WS-31-MED-02 agregó detección de cambio de visibilidad. Cuando el usuario cambia a otra aplicación o la pantalla se apaga, el navegador puede suspender la ejecución de JavaScript. Cuando el usuario vuelve y la página se hace visible nuevamente, el servicio verifica si la conexión sigue activa y, si no, inicia reconexión. Esto garantiza que el mozo que revisó su teléfono brevemente para algo personal puede volver a la aplicación y encontrarla funcionando.

El fix PWAW-A001 agregó tracking de expiración del JWT y scheduling de refresh. El servicio detecta cuándo el token está por expirar y puede invocar el callback de refresh proactivamente antes de que la conexión sea rechazada por token inválido.

Capítulo 4: El Procesamiento de Eventos en Tiempo Real

La Taxonomía de Eventos WebSocket

Los eventos WebSocket llegan como objetos JSON con una estructura consistente: `type` indica el tipo de evento, `branch_id` la sucursal, `table_id` la mesa afectada, `session_id` la sesión si aplica, y `entity` contiene datos específicos del evento como `round_id`, `call_id`, `check_id`, etc.

Los eventos del ciclo de vida de mesas incluyen `TABLE_SESSION_STARTED` (disparado cuando un comensal escanea el QR), `TABLE_STATUS_CHANGED` (cambios de estado de la mesa), y `TABLE_CLEARED` (sesión finalizada, mesa liberada).

Los eventos del ciclo de vida de rondas siguen el flujo completo: `ROUND_PENDING` (pedido nuevo esperando confirmación del mozo), `ROUND_CONFIRMED` (mozo verificó el pedido en la mesa), `ROUND_SUBMITTED` (enviado a cocina por admin/manager), `ROUND_IN_KITCHEN` (cocina comenzó preparación), `ROUND_READY` (cocina terminó), `ROUND_SERVED` (mozo entregó a la mesa), y `ROUND_CANCELED` (cancelado). El evento `ROUND_ITEM_DELETED` indica que el mozo eliminó un item de una ronda pendiente o confirmada.

Los eventos de llamados de servicio incluyen `SERVICE_CALL_CREATED` (cliente presionó el botón de llamado), `SERVICE_CALL_ACKED` (mozo indicó que vio

el llamado), y `SERVICE_CALL_CLOSED` (mozo atendió y resolvió el llamado).

Los eventos de facturación incluyen `CHECK_REQUESTED` (cliente solicitó la cuenta), `CHECK_PAID` (pago confirmado), `PAYMENT_APPROVED` (pago específico aprobado), y `PAYMENT_REJECTED` (pago rechazado).

El Handler de Eventos en el tablesStore

Cuando un evento llega a través del WebSocket, el store debe decidir qué hacer con él. El handler implementa una lógica ramificada extensa que procesa cada tipo de evento de manera apropiada.

Para eventos de ronda, el store primero verifica que el evento corresponda a una mesa en su lista. Esta verificación es crucial porque el WebSocket recibe eventos de toda la sucursal, pero el mozo solo debe reaccionar a eventos de sus sectores asignados. Si el evento es relevante, el store actualiza el campo `roundStatuses` con el nuevo estado de esa ronda específica, recalcula el `orderStatus` agregado de la mesa, actualiza contadores como `open_rounds`, y activa las animaciones correspondientes.

Para el evento `SERVICE_CALL_CREATED`, el store implementa deduplicación mediante un Set `seenServiceCallIds` que rastrea los IDs de llamados ya procesados. Esto es necesario porque el mismo llamado puede llegar múltiples veces si hay problemas de red o reconexiones. Sin deduplicación, el mozo podría recibir notificaciones repetidas. Además, el ID del llamado se agrega al array `activeServiceCallIds` de la mesa correspondiente,

permitiendo que el UI muestre cuántos llamados activos hay y ofrezca botones individuales de resolución.

Para `SERVICE_CALL_ACKED` y `SERVICE_CALL_CLOSED`, el store remueve el ID del array `activeServiceCallIds` y decrementa el contador `pending_calls`.

Para `TABLE_CLEARED`, el store ejecuta un reset completo del estado de la mesa: establece status a FREE, limpia `session_id`, resetea todos los contadores a cero, limpia el array de llamados activos, y resetea el estado de órdenes. También limpia todos los timeouts de animación pendientes para esa mesa, previniendo que animaciones huérfanas intenten actualizar estado inexistente.

Para eventos que no modifican campos específicos conocidos, el store realiza un fetch de la mesa actualizada desde el servidor mediante `tablesAPI.getTable(tableId)`. Antes de aplicar los datos del servidor, preserva el estado de animación local (`statusChanged`, `hasNewOrder`, etc.) para que el feedback visual continúe hasta su timeout natural.

El Sistema de Animaciones Temporales

Las animaciones visuales que alertan al mozo de cambios recientes requieren gestión cuidadosa de timeouts. El problema surge porque múltiples eventos pueden llegar para la misma mesa en rápida sucesión. Si cada evento programa un timeout independiente, pueden acumularse timeouts que se cancelan prematuramente o, peor, timeouts huérfanos que

intentan actualizar estado después de que la mesa fue limpiada.

La solución implementada utiliza tres Maps separados: `statusBlinkTimeouts` para animaciones de cambio de estado (blink azul de 1.5 segundos), `newOrderTimeouts` para animaciones de nuevo pedido (pulse amarillo de 2 segundos), y `serviceCallTimeouts` para animaciones de llamado de servicio (blink rojo de 3 segundos). El estado `ready_with_kitchen` tiene su propia animación de blink naranja que dura 5 segundos.

Cuando llega un evento que requiere animación, el código primero verifica si ya existe un timeout activo para esa mesa en el Map correspondiente. Si existe, lo cancela mediante `clearTimeout()`. Luego programa un nuevo timeout con la duración configurada en `ANIMATION_DURATIONS`, almacena el ID del timeout en el Map con la mesa como clave, y activa el flag booleano correspondiente en el estado de la mesa. Cuando el timeout expira, el callback elimina la entrada del Map y desactiva el flag mediante una actualización de estado.

Este patrón garantiza que solo existe un timeout activo por mesa por tipo de animación, y que eventos rápidos sucesivos extienden la animación en lugar de crear múltiples instancias conflictivas.

Capítulo 5: El Sistema de Notificaciones

La Dualidad de Canales: Visual y Sonoro

El mozo no puede estar mirando su pantalla constantemente. Está en movimiento, llevando platos, recogiendo pedidos, interactuando con clientes. Las notificaciones deben ser capaces de captar su atención incluso cuando el teléfono está en su bolsillo o sobre una bandeja.

El servicio de notificaciones en `notifications.ts` con aproximadamente 250 líneas implementa dos mecanismos complementarios. Las notificaciones web utilizan la API de Notification del navegador para mostrar mensajes visuales incluso cuando la aplicación está en segundo plano. El sonido de alerta utiliza un elemento de audio HTML para reproducir `/sounds/alert.mp3`.

La estrategia combina ambos canales según la urgencia del evento. Los eventos definidos como urgentes en `URGENT_WS_EVENTS` (SERVICE_CALL_CREATED, CHECK_REQUESTED, PAYMENT_APPROVED, etc.) reproducen el sonido independientemente del estado de permisos de notificación. Este comportamiento, documentado como QA-FIX, garantiza que el mozo sea alertado audiblemente incluso si no ha otorgado permisos de notificación. Luego, si el permiso está disponible, también se muestra la notificación visual.

El sonido de alerta se carga de manera lazy: el elemento de audio se crea solo cuando se necesita por primera vez, con `preload='none'` para no consumir ancho de banda cargando un archivo que podría nunca usarse.

Deduplicación y Control de Memoria

El fix WAITER-HIGH-02 introdujo un sistema de deduplicación basado en un Set `recentNotifications` que almacena tags de notificaciones recientes. Cada notificación tiene un tag derivado del tipo de evento y el ID de la entidad involucrada (por ejemplo, `SERVICE_CALL_CREATED:123`). Antes de mostrar una notificación, el servicio verifica si ese tag ya está en el Set. Si lo está, la notificación se descarta silenciosamente. Si no lo está, se añade al Set con un timeout de 5 segundos para su eliminación automática.

El fix WAITER-SVC-CRIT-03 abordó un problema de memory leak: el Set podía crecer indefinidamente si llegaban muchos eventos únicos. La solución implementa un límite de `MAX_RECENT_NOTIFICATIONS = 100` entradas. Cuando se alcanza el límite, el Set se vacía completamente mediante `clear()`. Este approach es más simple que implementar una cola FIFO y funciona bien en práctica porque el escenario de 100 notificaciones únicas en 5 segundos es extremadamente improbable.

Contenido Contextual de las Notificaciones

Las notificaciones incluyen información específica que permite al mozo tomar decisiones sin necesidad de abrir la aplicación. El mapping de eventos a contenido es:

- `ROUND_SUBMITTED`: título "Nuevo Pedido", cuerpo con número de mesa

- `SERVICE_CALL_CREATED`: título "Llamado de Mesa", cuerpo incluye el tipo de llamado (BILL para cuenta, ASSISTANCE para asistencia, OTHER para otros)
- `CHECK_REQUESTED`: título "Cuenta Solicitada", cuerpo con número de mesa
- `ROUND_READY`: título "Pedido Listo", cuerpo indicando que hay items para recoger de cocina

Las notificaciones urgentes requieren interacción del usuario para cerrarse, mientras que las no urgentes se cierran automáticamente después de 5 segundos.

Capítulo 6: La Interfaz de Usuario y sus Componentes

El Grid de Mesas con Agrupación por Sector

El componente `TableGrid.tsx` ocupa la mayor parte de la pantalla principal, mostrando todas las mesas asignadas al mozo organizadas por sector. La agrupación visual presenta un header con el nombre del sector (por ejemplo, "Interior", "Terraza"), un badge con el conteo de mesas en ese sector, y un indicador pulsante rojo si alguna mesa del sector tiene urgencias (llamados activos o pedidos pendientes de confirmar).

Los filtros en la parte superior permiten enfocarse en subconjuntos de mesas: ALL muestra todas, URGENT muestra solo las que tienen llamados o pedidos pendientes, ACTIVE muestra solo las ocupadas, FREE muestra solo las disponibles, y OUT_OF_SERVICE muestra las inhabilitadas. El filtro seleccionado

persiste en sessionStorage mediante el hook `usePersistedFilter`, así que si el mozo actualiza la página mantiene su contexto.

Dentro de cada sector, las mesas se ordenan por código alfanumérico. El componente `TableCard.tsx` renderiza cada mesa individual.

Las Tarjetas de Mesa y sus Animaciones

Cada `TableCard` es un rectángulo compacto diseñado para uso táctil con una sola mano. El código de mesa (INT-01, TER-03) aparece prominentemente en el centro. El color de fondo indica el estado: verde para FREE, rojo para ACTIVE, morado para PAYING, gris para OUT_OF_SERVICE.

Los badges en la esquina superior derecha proporcionan información crítica de un vistazo: un badge verde muestra el conteo de `open_rounds` (rondas pendientes de servir), un badge rojo muestra `pending_calls` (llamados activos), y un badge morado con el texto "Cuenta" aparece si `check_status === 'REQUESTED'`.

Las clases de animación se determinan por prioridad. La función `getAnimationClass()` evalúa en orden:

1. Si `hasServiceCall` es true → clase `animate-service-call-blink` (blink rojo, 3 segundos)
2. Si `orderStatus === 'ready_with_kitchen'` → clase `animate-ready-kitchen-blink` (blink naranja, 5 segundos)
3. Si `statusChanged` es true → clase `animate-status-blink` (blink azul, 1.5 segundos)

4. Si `hasNewOrder` es true → clase `animate-new-order-pulse` (pulse amarillo, 2 segundos)
5. Si `check_status === 'REQUESTED'` → clase `animate-check-pulse` (pulse morado, continuo)

El badge de estado de pedido debajo del código de mesa muestra el `orderStatus` agregado con colores distintivos: amarillo para pending, azul para confirmed/submitted/in_kitchen, naranja para ready_with_kitchen, verde para ready, gris para served.

El fix WAITER-COMP-HIGH-01 agregó un aria-label descriptivo para accesibilidad, describiendo el estado completo de la mesa para lectores de pantalla.

El Modal de Detalle de Mesa

Cuando el mozo toca una tarjeta de mesa, se abre `TableDetailModal.tsx` con más de 500 líneas de código. Este modal ocupa la mayor parte de la pantalla y presenta información detallada sobre la sesión activa.

Las pestañas de filtro en la parte superior permiten ver diferentes subconjuntos de rondas: "Todos" muestra todas las rondas, "Pendientes" filtra las que están en estado PENDING, CONFIRMED, SUBMITTED, o IN_KITCHEN, "Listos" muestra solo las READY, y "Servidas" muestra las SERVED.

La información de sesión incluye el listado de diners (comensales), las rondas agrupadas por categoría (Bebidas → Entradas → Principales → Postres), y el estado de cuenta con totales.

Las acciones disponibles dependen del estado de cada ronda:

- Para rondas en estado `PENDING`: aparece un botón prominente "Confirmar Pedido" que el mozo pulsa después de verificar físicamente el pedido en la mesa. Esta verificación es un paso crítico del flujo que previene que pedidos erróneos lleguen a cocina.

- Para rondas en estado `PENDING` o `CONFIRMED` (no enviadas a cocina): cada item individual tiene un ícono de papelera que permite eliminarlo. Un diálogo de confirmación previene eliminaciones accidentales. Si la eliminación deja la ronda vacía, la ronda completa se elimina automáticamente.

- Para rondas en estado `READY`: aparece un botón "Marcar como servido" que el mozo pulsa después de entregar los platos a la mesa.

La sección de llamados de servicio muestra los IDs de llamados activos almacenados en `activeServiceCallIds`, con un botón "Resolver" para cada uno que invoca `serviceCallsAPI.resolve()`.

El modal también incluye la pestaña `ComandaTab` para toma de pedidos rápida y un botón para abrir `FiscalInvoiceModal` si hay items consumidos.

El modal se actualiza en tiempo real mediante listeners de eventos WebSocket que recargan los datos cuando llegan eventos relevantes (ROUND_*, SERVICE_CALL_*, CHECK_*, PAYMENT_*). La gestión de

foco restaura el elemento anteriormente enfocado al cerrar el modal, y la tecla Escape lo cierra.

El Modal de Autogestión

El componente `AutogestionModal.tsx` con aproximadamente 300 líneas implementa el flujo de toma de pedidos gestionado directamente por el mozo. Este flujo es esencial para clientes que prefieren no usar sus teléfonos.

El modal opera en dos pasos. El primer paso presenta una lista de mesas FREE y ACTIVE. Para mesas FREE, el mozo ingresa la cantidad de comensales y el sistema crea una sesión mediante `waiterTableAPI.activateTable()`. Para mesas ACTIVE, simplemente usa la sesión existente.

El segundo paso presenta una interfaz split-view optimizada para velocidad. El panel izquierdo muestra el menú compacto obtenido de `comandaAPI.getMenuCompact()` (sin imágenes), organizado por categorías con un campo de búsqueda rápida. El mozo navega las categorías, busca productos por nombre, y los agrega al carrito con un toque.

El panel derecho muestra el carrito actual con controles de cantidad (+/-), el total acumulado, y un botón de envío. Al enviar, el sistema invoca `waiterTableAPI.submitRound()` y el pedido entra al flujo normal con estado PENDING.

El Tab de Comanda Rápida

El componente `ComandaTab.tsx` con aproximadamente 250 líneas proporciona la misma funcionalidad de toma de pedidos pero integrado dentro del `TableDetailModal`. Esto permite que el mozo que ya está viendo el detalle de una mesa pueda agregar items sin necesidad de abrir otro modal.

La implementación incluye un mount guard (`isMounted` flag) que previene actualizaciones de estado después de que el componente se desmonta, evitando el warning de React sobre memory leaks en efectos asíncronos.

Los Componentes de Factura Fiscal

Los componentes `FiscalInvoice.tsx` y `FiscalInvoiceModal.tsx` implementan la generación de facturas fiscales en formato argentino AFIP. Esta funcionalidad es una simulación para demostración; la integración real con AFIP requeriría certificados y comunicación con los servidores de la autoridad fiscal.

El componente `FiscalInvoice` renderiza una factura A4 (210mm × 297mm) con todos los elementos requeridos: header del negocio con nombre, CUIT, condición tributaria, y datos de contacto; badge del tipo de factura (A, B, o C según el tipo de cliente); tabla de items con cantidad, descripción, precio unitario, y total por línea; subtotal, tasa de IVA, monto de IVA, y total; número de CAE (simulado) y código QR placeholder; y método de pago utilizado.

El modal `FiscalInvoiceModal` envuelve la factura con un preview y un botón "Descargar PDF". La exportación a PDF utiliza las bibliotecas `jspdf` y

`html2canvas`: primero convierte el elemento DOM de la factura a canvas, luego lo embebe en un documento PDF que se descarga automáticamente.

Los tipos están definidos en `types/fiscal.ts`, incluyendo enums para `InvoiceType` (A, B, C) y `TaxCondition` (RESPONSABLE_INSCRIPTO, MONOTRIBUTO, CONSUMIDOR_FINAL, etc.), junto con helpers como `formatCuit()` para formatear el número de identificación tributaria.

Capítulo 7: Los Hooks Personalizados

usePullToRefresh: Gestos Táctiles

El hook `usePullToRefresh.ts` encapsula toda la lógica del gesto de arrastrar hacia abajo para refrescar. El mozo arrastra desde el inicio de la lista, un indicador visual muestra el progreso hacia el umbral de activación, y al soltar después del umbral se dispara la función de refresh.

El hook maneja los eventos táctiles (`touchstart`, `touchmove`, `touchend`), calcula las distancias considerando el scroll actual del contenedor, gestiona la resistencia del arrastre (se hace más difícil cuanto más se arrastra), y proporciona estados para que el componente renderice feedback apropiado.

El indicador de pull-to-refresh anuncia su estado a lectores de pantalla mediante una región aria-live,

garantizando accesibilidad para usuarios con discapacidad visual.

usePWA: Instalación de la Aplicación

El hook `usePWA.ts` gestiona el ciclo de vida de la PWA. Detecta el evento `beforeinstallprompt` que el navegador emite cuando la aplicación cumple los criterios para ser instalable, almacena el evento para invocarlo cuando el usuario esté listo, y expone funciones para mostrar el prompt de instalación y verificar si la aplicación ya está instalada.

También expone el estado `needRefresh` que indica cuando hay una nueva versión disponible del Service Worker, y la función `updateServiceWorker()` que fuerza la activación inmediata y recarga la página.

usePersistedFilter: Estado Persistente de Filtros

El hook `usePersistedFilter.ts` gestiona filtros que deben sobrevivir recargas de página. Utiliza sessionStorage para persistencia dentro de la sesión del navegador, permitiendo que el mozo que actualiza la página mantenga su filtro de mesas seleccionado.

useOnlineStatus: Detección de Conectividad

El hook `useOnlineStatus.ts` expone un booleano que indica si el dispositivo tiene conexión de red. Escucha los eventos `online` y `offline` del navegador y actualiza el estado reactivamente, permitiendo que la UI muestre banners de advertencia cuando se pierde conectividad.

Capítulo 8: La Configuración PWA y el Service Worker

El Manifiesto de Aplicación

El archivo `manifest.json` generado por vite-plugin-pwa describe la aplicación para el sistema operativo. Define el nombre completo "Sabor - Panel de Mozo" y el nombre corto "Mozo", el modo de visualización standalone (sin barra de navegación del navegador), la orientación portrait preferida, el color de tema naranja #f97316, y los íconos en resoluciones 192x192 y 512x512 para diferentes contextos de uso.

Los shortcuts permiten acciones rápidas desde el ícono en la pantalla de inicio: "Ver Mesas" navega a la raíz, y "Mesas Urgentes" navega con el filtro de urgentes preseleccionado.

Las screenshots proporcionan previews para el diálogo de instalación en dispositivos compatibles, con versiones wide y narrow para diferentes orientaciones.

Las Estrategias de Caché de Workbox

El Service Worker generado implementa múltiples estrategias de caché según el tipo de recurso.

Los assets estáticos (HTML, CSS, JavaScript, imágenes del bundle) se pre-cachean durante la instalación del Service Worker. Esto significa que se descargan todos de una vez y se sirven desde caché en visitas

subsiguientes, resultando en tiempos de carga casi instantáneos.

Las fuentes de Google utilizan estrategia CacheFirst con expiración de 365 días. Una vez descargada una fuente, se sirve desde caché indefinidamente.

Las llamadas a `/api/waiter/tables*` utilizan estrategia NetworkFirst con timeout de 5 segundos y caché de respaldo de 1 hora. Esto prioriza datos frescos pero permite funcionalidad degradada si la red falla.

Las imágenes de productos utilizan estrategia CacheFirst con expiración de 7 días. Las imágenes se descargan una vez y se sirven rápidamente desde caché, refrescándose semanalmente.

El fallback de navegación está configurado en `/index.html` con una denylist para rutas que comienzan con `/api`, asegurando que las rutas de API no sean interceptadas por el Service Worker.

Detección y Aplicación de Actualizaciones

Cuando se despliega una nueva versión de la aplicación, el Service Worker nuevo se descarga pero permanece en estado "waiting" hasta que todas las pestañas de la aplicación se cierren. El hook `usePWA` expone `needRefresh: true` cuando esto ocurre, permitiendo que la UI muestre un banner invitando al usuario a actualizar.

La función `updateServiceWorker()` fuerza la activación inmediata del nuevo Service Worker

mediante `skipWaiting()` y recarga todas las pestañas. Esto garantiza que el usuario siempre termina con todas sus pestañas en la misma versión del código, evitando inconsistencias.

Capítulo 9: El Sistema de Logging y Observabilidad

Loggers Contextuales

El archivo `logger.ts` implementa un sistema de logging estructurado con contextos. La función `createLogger(context)` retorna un objeto con métodos `debug`, `info`, `warn`, y `error`, cada uno prefijando los mensajes con timestamp, nivel, y contexto.

Los contextos definidos incluyen:

- `apiLogger = createLogger('API')` para mensajes relacionados con llamadas REST
- `wsLogger = createLogger('WebSocket')` para mensajes de conexión y eventos
- `authLogger = createLogger('Auth')` para flujo de autenticación y tokens
- `storeLogger = createLogger('Store')` para cambios de estado en Zustand
- `notificationLogger = createLogger('Notification')` para el sistema de alertas

El nivel `debug` solo produce output en desarrollo (`import.meta.env.DEV`), permitiendo logging verboso durante desarrollo sin contaminar la consola de producción.

El formato de salida es `[timestamp] [LEVEL] [context] message data`, donde data es el objeto opcional serializado como JSON para facilitar inspección.

Capítulo 10: Testing y Calidad de Código

Configuración de Vitest

La configuración de testing utiliza Vitest integrado con Vite. El archivo `vite.config.ts` incluye la sección de test que especifica `globals: true` para acceso global a funciones de test, `environment: 'jsdom'` para simular un DOM de navegador, y `setupFiles: ['./src/test/setup.ts']` para configuración inicial.

El archivo `setup.ts` configura mocks globales para APIs del navegador no disponibles en jsdom, como `localStorage`, `sessionStorage`, `BroadcastChannel`, y la API de Notification.

Archivos de Test Existentes

El proyecto incluye tests para los tres stores principales:

- `authStore.test.ts` verifica el flujo de login, la verificación de asignación, el refresh de tokens, y el logout correcto.

- `tablesStore.test.ts` verifica el procesamiento de eventos WebSocket, el cálculo del estado agregado de pedidos, y la gestión de animaciones.

- `retryQueueStore.test.ts` verifica la deduplicación de acciones, el procesamiento de la cola, y la limpieza de listeners.

Los tests utilizan React Testing Library para renderizado de componentes y `vi.fn()` para mocking de funciones.

Capítulo 11: El Flujo Completo de Pedidos

La Verificación como Paso Crítico

El sistema de pedidos implementa un flujo deliberadamente conservador donde los pedidos de clientes no llegan directamente a cocina. En lugar de eso, pasan primero por verificación del mozo, luego por aprobación del manager o admin. Esta cascada de validaciones responde a problemas reales observados en restaurantes.

Cuando un cliente envía un pedido desde pwaMenu, este llega con estado PENDING. El evento `ROUND_PENDING` se propaga a través del WebSocket y aparece en el dispositivo del mozo como una alerta de nuevo pedido (pulse amarillo). El mozo ve qué mesa tiene el pedido, va a verificarlo físicamente, y puede ajustar cantidades o eliminar items si el cliente cambió de opinión. Una vez satisfecho, pulsa "Confirmar Pedido"

en el `TableDetailModal` y el estado cambia a CONFIRMED.

Solo después de esta confirmación, el administrador o manager puede ver el pedido en el Dashboard y enviarlo a cocina, cambiando el estado a SUBMITTED. Este segundo paso existe para dar control adicional al management sobre cuándo se liberan pedidos a cocina, permitiendo por ejemplo agrupar pedidos de varias mesas para optimizar la producción.

Cuando cocina comienza la preparación, el estado cambia a IN_KITCHEN. Cuando termina, cambia a READY. El mozo recibe notificación sonora de que hay items listos para recoger. Después de entregar los platos, marca el pedido como SERVED.

El Flujo Alternativo de Autogestión

Para clientes que prefieren servicio tradicional, el mozo utiliza el modal de Autogestión. Activa una mesa libre ingresando la cantidad de comensales, navega el menú compacto agregando items al carrito, y envía el pedido. Este pedido entra al mismo flujo con estado PENDING, pero todo el proceso ocurre desde el dispositivo del mozo sin intervención del cliente.

Esta dualidad garantiza que el sistema no excluye a ningún tipo de cliente. Un restaurante puede tener mesas con códigos QR para clientes tech-savvy, y mesas tradicionales atendidas completamente por mozos. El backend maneja ambos flujos de manera unificada, y las métricas de tiempo de servicio capturan ambas modalidades.

Capítulo 12: Capacidades Offline y Resiliencia

El Problema de la Conectividad Intermitente

Los restaurantes no son data centers. La señal WiFi puede ser débil en ciertas áreas del salón. Las paredes gruesas de edificios antiguos pueden bloquear señales. Los dispositivos móviles de gama baja pueden tener receptores WiFi deficientes. Y durante horas pico, la congestión de red puede causar timeouts y paquetes perdidos.

Una aplicación que simplemente fallara cuando pierde conectividad sería inutilizable en este ambiente. El mozo no puede quedarse paralizado esperando que la red funcione mientras los clientes esperan.

La Estrategia de Caché y Cola

pwaWaiter implementa dos estrategias complementarias. La primera es el caché de estado: el Service Worker con estrategia NetworkFirst permite que la lista de mesas se sirva desde caché cuando el servidor no responde, con un indicador visual de que los datos pueden estar desactualizados.

La segunda estrategia es la cola de reintentos del `retryQueueStore`. Cuando el mozo intenta realizar una acción como marcar un pedido como servido y la llamada API falla por problemas de red, la acción se encola automáticamente. La cola se persiste en localStorage, así que sobrevive incluso si el mozo cierra la aplicación. Cuando el dispositivo detecta

reconexión mediante el evento `online`, procesa la cola automáticamente, ejecutando las acciones en el orden en que fueron encoladas.

Limitaciones del Modo Offline

Es importante notar que el modo offline tiene limitaciones. La aplicación no proporciona una experiencia completa offline de solo lectura; el caché de Workbox ayuda con assets estáticos y algunas respuestas de API, pero no hay sincronización completa de datos para visualización offline. Las acciones se encolan para retry pero no se ejecutan localmente con sincronización posterior.

Tampoco hay un diálogo de solicitud de permisos de notificación; el sistema simplemente solicita permisos en el momento del login, lo cual puede fallar silenciosamente si el navegador rechaza la solicitud.

Capítulo 13: Seguridad y Validaciones

Protección SSRF en el Cliente

El cliente API implementa validación estricta de URLs para prevenir que código malicioso pueda redirigir requests a servidores no autorizados. La configuración `API_CONFIG` define explícitamente los hosts permitidos (localhost, 127.0.0.1 en desarrollo) y los puertos permitidos (80, 443, 8000, 8001, y los puertos de las aplicaciones frontend).

En producción, la validación es más estricta: bloquea acceso directo a direcciones IP para forzar el uso de hostnames verificados, rechaza cualquier intento de alcanzar rangos de IP privados (10.x, 172.16-31.x, 192.168.x), y previene acceso a endpoints de metadata de cloud como 169.254.169.254 que podrían exponer credenciales.

Manejo Seguro de Tokens

Los tokens JWT de acceso tienen vida corta de 15 minutos para limitar el daño potencial si son comprometidos. El refresh token, con vida de 7 días, se almacena en una cookie HttpOnly que el JavaScript no puede leer, protegiéndolo de ataques XSS.

El mecanismo de refresh proactivo renueva el token un minuto antes de su expiración, evitando interrupciones de servicio. El flag `isRefreshing` previene race conditions donde múltiples componentes podrían disparar refreshes simultáneos. El contador de intentos fallidos limita a 3 retries antes de forzar logout, previniendo loops infinitos.

Headers de Seguridad

Aunque los headers de seguridad son responsabilidad del backend, vale la pena mencionar que el servidor responde con X-Content-Type-Options: nosniff para prevenir sniffing de MIME types, X-Frame-Options: DENY para prevenir clickjacking, Content-Security-Policy restrictivo, y HSTS en producción para forzar HTTPS.

Conclusión: Una Herramienta que Extiende al Profesional

pwaWaiter no pretende reemplazar al mozo humano sino potenciarlo. El juicio del profesional sobre cómo atender a cada cliente, cuándo ofrecer recomendaciones, cómo manejar situaciones difíciles: eso sigue siendo insustituiblemente humano. Lo que la aplicación hace es liberar ancho de banda mental, permitiendo que el mozo dedique su atención a lo que realmente importa en lugar de gastarla en recordar qué mesas tienen pedidos pendientes.

La arquitectura técnica refleja esta filosofía. El estado derivado garantiza que la información mostrada siempre es coherente con la realidad. Los eventos en tiempo real mantienen al mozo informado sin requerir que constantemente refresque la pantalla. Las capacidades offline garantizan que un problema de red no paraliza el servicio. La verificación de pedidos pone al mozo en control del flujo hacia cocina. Las facturas fiscales permiten cumplimiento tributario sin hardware adicional.

El código está diseñado para ser mantenible y extensible. Los stores de Zustand con selectores estables encapsulan la lógica de dominio de manera clara. Los servicios de comunicación implementan patrones robustos de resiliencia con timeouts, retries, y deduplicación. Los hooks personalizados extraen comportamiento reutilizable. Los componentes de UI son pequeños y enfocados. Los tests de Vitest verifican el comportamiento crítico. El sistema de

logging proporciona observabilidad para debugging en producción.

React 19 con sus hooks de optimismo y transiciones permite interfaces que responden instantáneamente a las acciones del usuario mientras las operaciones de red se resuelven en segundo plano. Zustand 5 con persist middleware mantiene el estado entre sesiones sin la complejidad de Redux. Vite 7 con PWA plugin genera una aplicación instalable que carga en milisegundos.

Y sobre todo, la aplicación está diseñada para desaparecer. El mejor software de productividad es el que se vuelve invisible, que el usuario opera sin pensar porque sus controles son intuitivos y sus respuestas predecibles. pwaWaiter aspira a ser ese tipo de herramienta: algo que el mozo usa naturalmente como extensión de sí mismo, que amplifica sus capacidades sin interponerse en su trabajo.

Documento técnico narrativo del proyecto pwaWaiter. Última actualización: Febrero 2026.

pwaMenu: La Mesa Digital

Introducción: El Problema de las Cartas Tradicionales

Imagina un grupo de amigos sentados en un restaurante, pasándose una carta de mano en mano, esperando turnos para ver los platos mientras el mozo aguarda pacientemente. Uno quiere verificar ingredientes por alergias, otro busca opciones vegetarianas, y un tercero simplemente quiere ver las fotos de los postres. Este ritual, tan familiar como ineficiente, representa exactamente el problema que pwaMenu resuelve.

La aplicación de menú para clientes es una PWA (Progressive Web App) que transforma cada teléfono móvil en una extensión de la carta del restaurante. Pero a diferencia de un simple catálogo digital, pwaMenu introduce un concepto revolucionario: el carrito compartido. En lugar de que cada comensal haga pedidos individuales, todos los dispositivos en una mesa comparten el mismo espacio de trabajo virtual, creando una experiencia verdaderamente colaborativa que respeta la naturaleza social de compartir una comida.

Esta arquitectura de "sesión compartida" presenta desafíos técnicos únicos. Cuando cinco personas miran el mismo carrito desde cinco dispositivos diferentes, ¿cómo se garantiza que todos vean exactamente lo mismo? ¿Cómo se previene que dos personas modifiquen la misma cantidad simultáneamente? ¿Y cómo se maneja la confirmación grupal para enviar un pedido cuando no todos están listos al mismo tiempo? Las respuestas a estas preguntas definen la arquitectura de pwaMenu, una aplicación construida sobre React 19, Zustand 5 y Vite 7, aprovechando las capacidades más recientes del ecosistema JavaScript para crear experiencias fluidas y resilientes.

Capítulo 1: Fundamentos Tecnológicos

React 19 y el Paradigma de Actualizaciones Optimistas

pwaMenu se construye sobre React 19.2.0, la versión más reciente del framework que introduce cambios paradigmáticos en el manejo de estados asíncronos. El hook `useOptimistic` permite mostrar cambios instantáneamente en la interfaz mientras las operaciones de red se resuelven en segundo plano. Cuando un comensal añade una hamburguesa al carrito, no espera confirmación del servidor: el item aparece inmediatamente. Si el servidor eventualmente rechaza la operación, el sistema revierte automáticamente al estado anterior sin intervención manual.

El hook `useTransition` complementa esta arquitectura permitiendo marcar actualizaciones como "no urgentes", evitando que cálculos costosos bloqueen la interacción del usuario. Cuando el menú carga cientos de productos, las operaciones de filtrado y ordenamiento se ejecutan sin congelar el scroll ni los botones táctiles.

El hook `useActionState`, introducido en React 19, transforma el manejo de formularios. En lugar de múltiples estados para loading, error y data, el hook encapsula todo el ciclo de vida de una acción asíncrona. Los componentes `ProductDetailModal`, `CallWaiterModal` y `JoinTable`

utilizan este patrón para manejar envíos de formularios con feedback automático de estado pendiente.

El compilador de React, integrado mediante babel-plugin-react-compiler, automatiza la memoización de componentes y callbacks. Esta optimización elimina la necesidad de envolver manualmente cada componente en `React.memo` o cada callback en `useCallback`, reduciendo significativamente el código boilerplate y los errores humanos asociados a optimizaciones manuales incorrectas.

Zustand 5 y la Gestión de Estado Modular

Para la gestión del estado global, la aplicación emplea Zustand en su versión 5.0.9. Esta biblioteca representa una alternativa minimalista a Redux que elimina la verbosidad característica de los patrones flux tradicionales. Un store de Zustand se define mediante una única función que retorna el estado inicial y las acciones que lo modifican, sin necesidad de reducers, action creators o middleware externos.

La arquitectura de stores en pwaMenu sigue un patrón modular riguroso. El **tableStore** gestiona toda la lógica relacionada con la sesión de mesa, el carrito compartido y los pedidos, organizado en cuatro archivos especializados: `store.ts` para las acciones principales, `types.ts` para las interfaces TypeScript, `selectors.ts` para los selectores optimizados, y `helpers.ts` para funciones puras de utilidad. El **menuStore** mantiene el catálogo de productos con caché temporal de cinco minutos. El **sessionStore** maneja la conexión con el backend y la persistencia de tokens. El **serviceCallStore** rastrea las llamadas al mozo con selectores memoizados.

Cada store implementa persistencia automática mediante el middleware `persist` de Zustand, que serializa el estado a localStorage y lo rehidrata al cargar la aplicación. Esta característica permite que un comensal cierre accidentalmente el navegador y, al reabrirlo, encuentre su sesión exactamente donde la dejó, con su carrito intacto y su identidad preservada.

Vite 7 y la Arquitectura de Empaquetado

Vite 7.2.4 actúa como el corazón del sistema de construcción, proporcionando un servidor de desarrollo con recarga instantánea y un proceso de build optimizado para producción. Su arquitectura basada en módulos ES nativos durante el desarrollo elimina la necesidad de empaquetar el código completo en cada cambio, resultando en tiempos de recarga medidos en milisegundos.

La configuración de producción implementa división de código estratégica mediante chunks manuales. Las dependencias de terceros se agrupan en un chunk `vendor` que cambia infrecuentemente y puede cachearse agresivamente. Las traducciones se separan en un chunk `i18n` que solo se descarga cuando el usuario cambia de idioma. Los componentes modales pesados como el chat de IA o los filtros avanzados residen en sus propios chunks, descargándose bajo demanda únicamente cuando el usuario los requiere.

El plugin vite-plugin-pwa transforma la aplicación en una Progressive Web App completa, generando automáticamente el service worker, el manifiesto de aplicación y los iconos en múltiples resoluciones. La configuración define comportamientos de caché diferenciados según el tipo de recurso: las imágenes de productos emplean estrategia CacheFirst con expiración de 30 días, mientras que las llamadas a API utilizan NetworkFirst con timeout de 5 segundos y fallback a caché.

Capítulo 2: El Viaje del Comensal

La Llegada a la Mesa

El primer contacto del comensal con pwaMenu ocurre a través de un código QR pegado en la mesa. Este simple escaneo desencadena una orquestación compleja que el usuario nunca percibe. El código contiene un identificador alfanumérico de la mesa (como "INT-01" o "TER-02") que la aplicación utiliza para conectarse con el backend y obtener o crear una sesión activa.

El componente `JoinTable` maneja este flujo inicial mediante un proceso de dos pasos implementado como wizard modular. El subdirectorio `JoinTable/` contiene `TableNumberStep.tsx` para la confirmación del número de mesa y `NameStep.tsx` para el ingreso opcional del nombre del comensal. Esta arquitectura modular facilita extensiones futuras como verificación de edad para locales nocturnos o selección de idioma preferido.

Primero, el usuario ve su número de mesa pre-poblado desde el QR y lo confirma. Luego, opcionalmente, ingresa su nombre para identificarse ante los demás comensales. Esta información viaja al backend a través del endpoint `/api/tables/code/{code}/session?branch_slug={slug}`, que responde con un **table token**, un JWT especializado con tiempo de vida de tres horas que autoriza todas las operaciones posteriores de ese comensal en esa mesa específica.

Lo notable de este diseño es su tolerancia a la ambigüedad. Los códigos de mesa no son únicos globalmente: cada sucursal puede tener su propia "INT-01". Por eso, la aplicación siempre envía el slug de la sucursal junto con el código de mesa, permitiendo al backend resolver la mesa correcta sin confusiones. Este pequeño detalle arquitectónico previene errores sutiles que serían devastadores en producción, donde un pedido enviado a la mesa equivocada destruiría la experiencia del usuario.

El Sistema de Identificación de Dispositivo

Antes de que el árbol de componentes React se monte, el archivo `main.tsx` ejecuta una inicialización crítica: la generación del identificador único de dispositivo. Este proceso ocurre de manera asíncrona para no bloquear el renderizado inicial, pero establece las bases para el sistema de fidelización que operará durante toda la sesión.

El módulo `deviceId.ts` genera dos identificadores complementarios. El `deviceId` es un UUID v4 simple generado mediante `crypto.randomUUID()` y almacenado en localStorage. Persiste entre sesiones del navegador mientras el usuario no limpie sus datos de navegación. Su simplicidad lo hace robusto pero vulnerable a pérdida si el usuario cambia de dispositivo o limpia datos.

El `deviceFingerprint` complementa al deviceId con una huella digital más sofisticada. Se computa un hash SHA-256 combinando características del navegador: user agent completo, resolución de pantalla y profundidad de color, zona horaria del sistema y preferencia de idioma, plataforma del sistema operativo, cantidad de memoria RAM disponible y núcleos de CPU, número máximo de puntos táctiles soportados. Esta combinación genera un identificador relativamente único que puede ayudar a reconocer el mismo dispositivo incluso si el localStorage se limpia.

La función `getDeviceInfo()` retorna ambos identificadores empaquetados, listos para enviarse al backend cuando el comensal se registra en una mesa. La función `isReturningDevice()` verifica si el dispositivo actual ha visitado anteriormente, permitiendo personalización desde el primer momento de la sesión.

El Catálogo Vivo

Una vez dentro de la sesión, el comensal accede al menú completo del restaurante a través de la página `Home.tsx`. Pero este no es un simple catálogo estático. El `menuStore` mantiene una representación local del menú que se actualiza dinámicamente y se cachea inteligentemente para evitar peticiones redundantes.

El menú se obtiene del endpoint `/api/public/menu/{slug}` que retorna la estructura completa de categorías, subcategorías, productos y alérgenos de la sucursal. Cada producto llega con información estructurada: nombre y descripción en múltiples idiomas, imagen, alérgenos asociados con nivel de presencia, perfiles dietéticos (vegano, vegetariano, sin gluten), métodos de cocción, y precios específicos de la sucursal.

La conversión de datos merece atención especial. El backend almacena precios en centavos (12550 representa $125.50) para evitar errores de punto flotante inherentes a la representación binaria de decimales, pero el frontend los presenta en pesos con decimales para la comodidad del usuario. Las funciones de conversión en `Home.tsx` (`convertBackendProduct`, `convertBackendCategory`, `convertBackendSubcategory`) transforman los tipos del backend (snake_case, IDs numéricos, precios en centavos) a los tipos del frontend (camelCase, IDs string, precios decimales).

El sistema de caché implementa una política TTL (Time To Live) de cinco minutos que balancea frescura contra eficiencia. Un menú obsoleto por más tiempo podría mostrar productos descontinuados, pero refrescar constantemente consumiría datos innecesarios. El flag `lastFetch` registra el timestamp de la última obtención, y el método `fetchMenu` con parámetro `forceRefresh` permite invalidación manual del caché cuando es necesario.

Capítulo 3: Los Filtros como Herramientas de Inclusión

El Desafío de las Restricciones Alimentarias

En cualquier grupo de comensales, las necesidades dietéticas varían enormemente. Uno puede ser celíaco, otro intolerante a la lactosa, y un tercero simplemente prefiere evitar frituras. Los sistemas tradicionales de filtrado ofrecen checkboxes binarios que ocultan productos, pero pwaMenu va mucho más allá con un sistema de filtrado sofisticado que reconoce la complejidad real de las restricciones alimentarias.

El hook `useAllergenFilter` representa esta sofisticación. No solo permite excluir alérgenos específicos, sino que distingue entre diferentes niveles de presencia. Un producto puede "contener" un alérgeno (presente como ingrediente declarado), "poder contenerlo" (trazas por contaminación cruzada en la cocina), o estar "libre de" él (garantizado sin presencia, procesado en ambiente controlado).

El usuario puede elegir entre tres niveles de restricción: modo **permisivo** que solo marca pero no excluye productos con el alérgeno, modo **moderado** que oculta productos donde el alérgeno es ingrediente pero permite aquellos con posibles trazas mostrando advertencia, y modo **estricto** que excluye cualquier producto con presencia confirmada o posible del alérgeno. Esta última opción es vital para personas con alergias severas donde incluso mínimas exposiciones representan riesgos médicos de anafilaxis.

Las Reacciones Cruzadas: Una Dimensión Oculta

La verdadera innovación del sistema de filtrado reside en su comprensión de las reacciones cruzadas entre alérgenos. El síndrome látex-fruta ilustra perfectamente este fenómeno: una persona alérgica al látex frecuentemente también reacciona a plátanos, aguacates, kiwis y castañas debido a proteínas estructuralmente similares. El síndrome de alergia oral relaciona el polen de abedul con manzanas, peras y cerezas. Estas conexiones no son obvias para el comensal promedio, pero ignorarlas puede tener consecuencias médicas serias.

El sistema obtiene del backend un grafo de reacciones cruzadas almacenado en el modelo `AllergenCrossReaction`, una relación self-referencial many-to-many que conecta alérgenos relacionados con probabilidades asociadas (alta, media, baja) y descripción de la relación. El hook `useAllergenFilter` permite al usuario configurar su sensibilidad a estas conexiones. Alguien con alergias leves podría considerar solo reacciones de alta probabilidad, mientras que alguien con historial de anafilaxis querrá incluir todas las conexiones conocidas.

El resultado es un filtrado que va más allá de lo que el usuario conscientemente sabe sobre sus alergias. Cuando un comensal marca "alergia al maní", el sistema puede advertirle sobre productos con otros frutos secos que frecuentemente causan reacciones cruzadas, protegiéndolo proactivamente.

Preferencias Dietéticas y Métodos de Cocción

El hook `useDietaryFilter` complementa el sistema de alérgenos con preferencias de estilo de vida. Las opciones incluyen vegetariano (sin carne pero permite lácteos y huevos), vegano (sin ningún producto animal), sin gluten (excluye trigo, cebada, centeno), apto para celíacos (sin gluten más garantía de no contaminación cruzada), keto (bajo en carbohidratos), y bajo en sodio. Cada opción representa no solo una restricción sino una forma de relacionarse con la comida.

La implementación requiere que el producto satisfaga todas las opciones seleccionadas simultáneamente mediante operador lógico AND, reconociendo que una persona puede ser

tanto vegetariana como intolerante al gluten. Los productos se filtran contra los atributos booleanos correspondientes almacenados en el modelo de producto.

Similarmente, `useCookingMethodFilter` permite excluir métodos de preparación específicos. Un usuario que evita frituras por razones de salud puede configurar el filtro una vez y olvidarse, viendo solo opciones compatibles con su preferencia. Las opciones incluyen frito, a la parrilla, al horno, hervido, al vapor, crudo, salteado, entre otros métodos registrados en el catálogo de `CookingMethod` del tenant.

El hook `useAdvancedFilters` combina los tres sistemas de filtrado (alérgenos, dietético, método de cocción) aplicándolos secuencialmente y proporcionando una lista final de productos que cumplen todos los criterios seleccionados.

La Persistencia Inteligente de Preferencias

Todo este sistema de filtrado sería inútil si el usuario tuviera que reconfigurarlo cada vez que visita el restaurante. El hook `useImplicitPreferences` resuelve este problema sincronizando las preferencias con el backend a través del endpoint `PATCH /api/diner/preferences` y asociándolas con el identificador del dispositivo.

La sincronización utiliza debouncing de dos segundos para evitar peticiones excesivas mientras el usuario ajusta múltiples filtros en rápida sucesión. El hook `useDebounce` implementa este comportamiento con protección contra race conditions, separando cuidadosamente el efecto de montaje/desmontaje del efecto de actualización de valor.

Cuando un comensal regresa semanas después con el mismo dispositivo, el endpoint `GET /api/diner/device/{device_id}/preferences` recupera sus preferencias guardadas y el hook las aplica automáticamente al cargar la aplicación. El comensal encuentra el menú ya filtrado según sus restricciones habituales sin configuración manual, creando una experiencia personalizada sin requerir registro explícito.

Capítulo 4: El Carrito Compartido

La Anatomía de una Sesión de Mesa

El corazón de pwaMenu es el `tableStore`, un estado Zustand que representa la sesión de mesa en su totalidad. La interface `TableState` define más de veinte propiedades organizadas en categorías funcionales: información de sesión (`session`, `currentDiner`), estado del carrito (`cart` implícito en session), control de operaciones (`isLoading`, `isSubmitting`, `submitSuccess`, `_submitting`), historial de pedidos (`orders`, `currentRound`, `lastOrderId`), estado de confirmación grupal (`roundConfirmation`), y registro de pagos (`dinerPayments`).

Cuando un comensal agrega un producto al carrito mediante la acción `addToCart`, el item queda etiquetado con su `diner_id` y nombre. Esto permite que el carrito muestre quién pidió qué mediante códigos de color consistentes generados por `getColorForIndex`, facilitando la división de cuentas al final y creando una sensación de propiedad sobre las selecciones individuales dentro del contexto colaborativo. Cada comensal puede modificar solo sus propios items comparando `diner_id` con `currentDiner.id`, pero todos pueden ver el carrito completo.

La sesión incluye un campo `last_activity` que se actualiza con cada interacción del usuario: agregar item, modificar cantidad, eliminar producto. El sistema de expiración verifica este timestamp en lugar del tiempo de creación, permitiendo que sesiones con actividad continua persistan indefinidamente mientras que sesiones abandonadas expiran después de ocho horas de inactividad.

El Flujo de Sincronización en Tiempo Real

Cuando cinco personas miran el mismo carrito desde cinco dispositivos diferentes, la consistencia visual es crítica. El flujo de sincronización opera en múltiples capas coordinadas.

Cuando un comensal añade un producto al carrito, la operación sigue un flujo preciso orquestado entre el store local, el backend REST y el gateway WebSocket:

1. La acción `addToCart` del store valida los datos de entrada: el producto existe en el menú, la cantidad está en rango válido (1-10), la sesión permanece activa y no expirada.

2. Se genera un ID temporal optimista mediante `generateId()` y se añade al estado local inmediatamente, proporcionando feedback visual instantáneo.

3. Se envía petición POST a `/api/diner/cart/add` con los datos del item: `product_id`, `quantity`, `notes`, `diner_id`.

4. El backend persiste el item en la tabla `cart_item` y emite evento `CART_ITEM_ADDED` al canal Redis de la sesión.

5. El gateway WebSocket distribuye el evento a todos los comensales conectados a esa mesa.

6. El hook `useCartSync` en cada dispositivo procesa el evento mediante el callback registrado con `dinerWS.on('CART_ITEM_ADDED', ...)`.

7. En el dispositivo originador, la función `isFromCurrentDiner()` detecta que el evento corresponde a la operación local y omite actualizaciones duplicadas.

8. En otros dispositivos, el item se añade al estado local mediante `addRemoteCartItem`, apareciendo instantáneamente en sus interfaces.

Si el paso 3 falla por error de red o validación del servidor, el item optimista se elimina del estado local mediante rollback automático, revirtiendo el cambio visual. El usuario ve el item desaparecer y recibe notificación del error vía toast.

Optimizaciones del Hook useCartSync

El hook `useCartSync` representa una pieza crítica de la arquitectura, responsable de mantener sincronizado el carrito local con los cambios de otros comensales. Su implementación incorpora múltiples optimizaciones identificadas durante auditorías de rendimiento (PERF-01, PERF-02, PERF-03).

Un caché LRU (Least Recently Used) almacena conversiones de items del backend al formato frontend, evitando reconstruir objetos idénticos repetidamente cuando el mismo producto se añade múltiples veces. La deduplicación de eventos mediante un Set con límite de 100 entradas y TTL de 5 segundos previene procesamiento duplicado de eventos que podrían llegar por múltiples caminos en condiciones de red inestables.

El debounce en reconexión agrupa actualizaciones que llegan en ráfaga cuando la conexión WebSocket se restablece tras una desconexión. En lugar de aplicar cada cambio individualmente provocando múltiples re-renders, se acumulan durante un segundo y se aplican en batch, mejorando significativamente la percepción de rendimiento.

El Problema de la Concurrencia Optimista

El sistema utiliza actualizaciones optimistas mediante el hook `useOptimisticCart` de React 19. Este hook envuelve el estado real del carrito con una capa de actualizaciones pendientes que se aplican instantáneamente a la interfaz mientras la operación real se ejecuta en segundo plano.

La generación de IDs temporales incluye un contador incremental además del UUID para garantizar unicidad incluso en escenarios de double-click rápido donde `Date.now()` podría

retornar el mismo valor. Esta protección, implementada tras identificar colisiones en auditoría, asegura que cada item optimista tenga un identificador único.

La deduplicación de items durante reconciliación merece mención especial. Cuando el mismo producto aparece dos veces con IDs diferentes (uno temporal, otro real), el sistema en `SharedCart.tsx` fusiona estas entradas preferiendo el ID permanente del backend. Un Map con clave compuesta `${product_id}-${diner_id}` detecta duplicados y los elimina antes del renderizado, previniendo glitches visuales durante la reconciliación entre estado optimista y estado confirmado.

La Confirmación Grupal: Un Protocolo de Consenso

El momento más delicado del flujo es enviar el pedido a cocina. A diferencia de una aplicación individual donde el usuario simplemente presiona "enviar", en una mesa compartida surge la pregunta: ¿están todos listos? ¿Alguien quiere agregar algo más?

El sistema de confirmación grupal implementa un protocolo de consenso inspirado en sistemas distribuidos. La interface `RoundConfirmation` define la estructura del estado de propuesta:

```typescript
interface RoundConfirmation {
  proposer_id: string          // Quién inició la propuesta
  proposed_at: number          // Timestamp de inicio
  status: 'pending' | 'confirmed' | 'cancelled' | 'expired'
  diner_statuses: Map<string, DinerReadyStatus>  // Estado por comensal
}

interface DinerReadyStatus {
  diner_id: string
  is_ready: boolean
  confirmed_at?: number
}
```

Cuando un comensal ejecuta `proposeRound()`, todos los demás reciben actualización visual mostrando quién propuso y el estado de cada participante. El componente `RoundConfirmationPanel` renderiza esta información con indicadores de color: verde para confirmados, gris para esperando.

Cada comensal puede ejecutar `confirmReady()` para marcar su disposición. El selector `useRoundConfirmationData` calcula derivados: `confirmationCount` (cuántos confirmaron), `allReady` (si todos confirmaron), `hasCurrentDinerConfirmed` (si el usuario actual ya confirmó), `isProposer` (si el usuario actual inició la propuesta).

Cuando `allReady` se vuelve true, un temporizador de 1.5 segundos inicia la cuenta regresiva visible. Este delay permite cancelación de último momento si alguien cambió de opinión. Al expirar el temporizador, `submitOrder()` se ejecuta automáticamente enviando el pedido a cocina.

La propuesta tiene un timeout de cinco minutos implementado mediante verificación de `proposed_at` contra `Date.now()`. Si expira sin confirmación unánime, el estado cambia a 'expired' y se limpia la propuesta. El proponente puede ejecutar `cancelRoundProposal()` en cualquier momento, y cualquier comensal puede revocar su confirmación mediante `cancelReady()` antes del envío final.

El diseño reconoce que el envío de un pedido es un momento social, no solo técnico. Forzar el envío cuando alguien aún está decidiendo sería tan molesto como que un comensal gritara al mozo sin consultar a los demás. El protocolo de consenso digitaliza la cortesía natural de preguntar "¿pedimos ya?".

Capítulo 5: La Conexión en Tiempo Real

El WebSocket del Comensal

Mientras el mozo y la cocina tienen sus propios canales WebSocket con autenticación JWT tradicional, el comensal utiliza un canal diferente autenticado mediante el token de mesa. La clase `DinerWebSocket` en `services/websocket.ts` encapsula esta conexión, manejando la complejidad de mantener un vínculo persistente en el entorno hostil de los navegadores móviles.

La conexión se establece hacia `${WS_URL}/ws/diner?table_token=${token}` donde el gateway WebSocket valida el token y extrae la información de sesión. Una vez autenticado, el comensal queda suscrito al canal de su mesa específica, recibiendo todos los eventos relevantes para esa sesión.

El patrón pub/sub interno de la clase permite registrar múltiples listeners para diferentes tipos de eventos. El método `on(eventType, callback)` registra un callback y retorna una función de cleanup para desregistro. Los hooks de React utilizan este mecanismo en sus efectos:

```typescript
useEffect(() => {
  const unsubscribe = dinerWS.on('CART_ITEM_ADDED', handleCartAdd)
  return unsubscribe  // Cleanup al desmontar
}, [])
```

Reconexión con Backoff Exponencial y Jitter

La reconexión automática utiliza backoff exponencial con jitter aleatorio, patrón crítico para aplicaciones móviles donde las conexiones se pierden frecuentemente. Cuando la conexión se pierde, el sistema espera un segundo antes del primer intento, luego dos, luego cuatro, hasta un máximo de treinta segundos entre intentos.

El jitter del 50% evita la "estampida de reconexiones" (thundering herd) donde muchos dispositivos que perdieron conexión simultáneamente (por caída momentánea del servidor) intentarían reconectarse exactamente al mismo tiempo, potencialmente sobrecargando el servidor nuevamente. El delay efectivo se calcula como:

```
delay = baseDelay * 2^attempt * random(0.5, 1.5)
```

El límite de 50 intentos máximos (CLIENT-MED-01 FIX) previene loops infinitos de reconexión cuando el problema es permanente (servidor caído indefinidamente, token expirado). Tras agotar los intentos, la conexión se marca como permanentemente fallida y se notifica al usuario sugiriendo recargar la página.

Los códigos de cierre WebSocket determinan el comportamiento de reconexión. Los códigos estándar 1000 (cierre normal) y 1001 (navegando fuera) no disparan reconexión. Los códigos de error recuperables (1006 conexión perdida, 1013 servidor sobrecargado) disparan el proceso de backoff. Los códigos especiales 4001 (autenticación fallida), 4003 (acceso prohibido) y 4029 (rate limited) indican problemas que no se resolverán con reintentos y provocan abandono inmediato.

El Heartbeat y la Detección de Conexiones Zombi

El protocolo de heartbeat envía un mensaje `{"type":"ping"}` cada treinta segundos y espera un `{"type":"pong"}` dentro de diez segundos. Si el pong no llega, el sistema cierra la conexión proactivamente y comienza el proceso de reconexión.

Este mecanismo detecta conexiones "zombi": aquellas que parecen abiertas según el estado del WebSocket pero han perdido comunicación real. Este escenario es común cuando un dispositivo móvil entra en modo suspensión sin cerrar apropiadamente las conexiones TCP subyacentes, o cuando un proxy intermediario cierra silenciosamente conexiones inactivas.

El listener de visibilidad complementa el sistema de heartbeat. El evento `visibilitychange` del documento detecta cuando el usuario cambia de pestaña o desbloquea el teléfono después de un período de suspensión. Al retornar a la aplicación, el sistema verifica inmediatamente el estado de la conexión: si el heartbeat está vencido o la conexión parece muerta, se restablece proactivamente en lugar de esperar al próximo ciclo de heartbeat. Este comportamiento asegura que el comensal siempre tenga información actualizada cuando activamente mira la aplicación.

Eventos del Ciclo de Vida del Pedido

A través del WebSocket, el comensal recibe actualizaciones sobre el progreso de sus pedidos. El ciclo de vida completo de un round atraviesa seis estados:

Estado	Evento WebSocket	Significado
PENDING	ROUND_PENDING	Pedido creado, aguarda verificación del mozo
CONFIRMED	ROUND_CONFIRMED	Mozo verificó presencialmente en la mesa
SUBMITTED	ROUND_SUBMITTED	Admin/Manager envió a cocina

| IN_KITCHEN | ROUND_IN_KITCHEN | Cocina comenzó preparación |

| READY | ROUND_READY | Cocina terminó, listo para servir |

| SERVED | ROUND_SERVED | Mozo entregó a la mesa |

El hook `useOrderUpdates` escucha estos eventos y actualiza el estado local mediante `updateOrderStatus`. Cada transición permite que el comensal sepa exactamente dónde está su comida sin necesidad de preguntar al mozo. El componente `OrderHistory` muestra esta información con indicadores visuales de progreso.

El evento `ROUND_ITEM_DELETED` merece atención especial. Cuando un mozo elimina un item de un pedido pendiente o confirmado (porque el producto no está disponible, por ejemplo), el evento se emite a todos los comensales de la mesa. El handler actualiza el carrito local removiendo el item correspondiente, manteniendo consistencia entre la realidad operativa del restaurante y la visión del cliente.

Los eventos de carrito (`CART_ITEM_ADDED`, `CART_ITEM_UPDATED`, `CART_ITEM_REMOVED`, `CART_CLEARED`, `CART_SYNC`) mantienen sincronizado el carrito compartido según se describió anteriormente.

Capítulo 6: El Reconocimiento del Cliente Recurrente

La Identificación Sin Registro

La mayoría de las aplicaciones de fidelización requieren creación de cuenta, ingreso de email, verificación por código, y todo un ritual que interrumpe la experiencia. pwaMenu invierte esta lógica: primero reconoce, después ofrece.

El sistema de identificación opera en cuatro fases evolutivas, cada una construyendo sobre la anterior:

Fase 1 - Device Tracking: El identificador único de dispositivo (`deviceId`) y la huella digital (`deviceFingerprint`) se generan en la primera visita y persisten en localStorage. Al registrar un comensal en una mesa, ambos identificadores se envían al backend en el payload de `POST /api/diner/register`. El endpoint `GET /api/diner/device/{device_id}/history` permite consultar

el historial de visitas de un dispositivo, retornando fechas, mesas visitadas y productos ordenados previamente.

Fase 2 - Preferencias Implícitas: Como se describió en el capítulo de filtros, el hook `useImplicitPreferences` sincroniza automáticamente las configuraciones de filtrado con el backend. En visitas posteriores, estas preferencias se cargan y aplican sin intervención del usuario.

Fase 3 - Recomendaciones Contextuales: Cuando un dispositivo conocido se conecta, el backend puede ofrecer información contextual: "Bienvenido de nuevo, la última vez pediste la Milanesa Napolitana" o "Basado en tus preferencias, te recomendamos evitar los platos con maní". El endpoint de menú puede incluir un campo `returning_device_suggestions` con esta información.

Fase 4 - Opt-in de Fidelización: Para usuarios que desean funcionalidades adicionales (acumulación de puntos, ofertas personalizadas, historial detallado), el sistema ofrece un registro voluntario que vincula el dispositivo a un perfil de cliente.

El Sistema de Opt-in

El hook `useCustomerRecognition` detecta cuando el dispositivo actual corresponde a un visitante frecuente sin cuenta registrada. Mediante el endpoint `GET /api/customer/recognize`, el sistema verifica si el `deviceId` está asociado a un cliente existente. Si no lo está pero el dispositivo tiene historial significativo (múltiples visitas), se presenta el `OptInModal` invitando al usuario a registrarse.

El formulario de registro implementa consentimiento granular cumpliendo requisitos GDPR. El usuario puede aceptar o rechazar independientemente:

- Almacenamiento de preferencias dietéticas
- Análisis de historial de compras para recomendaciones
- Comunicaciones promocionales por email/SMS
- Personalización mediante inteligencia artificial

El endpoint `POST /api/customer/register` crea el perfil de cliente con los consentimientos especificados. El campo `customer_id` en la tabla `Diner` vincula visitas posteriores al perfil registrado.

Una vez registrado, `GET /api/customer/suggestions` proporciona recomendaciones personalizadas: productos favoritos basados en historial, items populares entre clientes con preferencias similares, ofertas exclusivas para clientes fidelizados.

Este diseño respeta la privacidad por defecto mientras ofrece beneficios tangibles a quienes eligen participar. La diferencia con sistemas tradicionales es sutil pero significativa: el restaurante reconoce al cliente antes de pedirle que se registre, demostrando valor antes de solicitar compromiso.

Capítulo 7: El Cierre de Mesa y el Pago

El Flujo de Solicitud de Cuenta

Cuando el grupo termina de comer, cualquier comensal puede solicitar la cuenta tocando el botón correspondiente en `BottomNav`. Esta acción ejecuta `closeTable()` en el store, que internamente llama al endpoint `POST /api/billing/check/request` para crear un registro de cuenta (Check) en el backend.

El sistema valida que no queden items pendientes en el carrito (productos agregados pero no enviados), mostrando advertencia si existen. Esta validación previene el escenario donde un comensal olvida que había seleccionado un postre que nunca se pidió, evitando sorpresas desagradables al ver la cuenta.

El backend calcula el total sumando los precios de todos los items de todos los rounds de la sesión, aplicando promociones vigentes si corresponde. El modelo `Check` (con tabla `app_check` para evitar palabra reservada SQL) incluye desglose por item con `Charge`, subtotal, impuestos si aplican, y total final. El estado de la sesión cambia a `PAYING` y se emite evento `CHECK_REQUESTED` al canal del mozo asignado.

La solicitud de cuenta no cierra la sesión inmediatamente. El diseño reconoce que los comensales frecuentemente agregan "una última cosa" mientras esperan la cuenta, o que alguien puede pedir un café adicional. La sesión permanece activa para nuevos pedidos mientras está en estado `PAYING`, que simplemente se agregarán al total final en ciclos de pago subsecuentes.

La División de Cuentas

La página `CloseTable.tsx` presenta el resumen de consumo y opciones de división organizadas en componentes modulares: `CloseTableHeader`, `TotalCard`, `SummaryTab`, `OrdersList`, cada uno en el subdirectorio `close-table/`.

El helper `calculatePaymentShares()` en `tableStore/helpers.ts` soporta tres estrategias de división definidas por el tipo `SplitMethod`:

División igualitaria (`equal`): El total se divide equitativamente entre los comensales presentes. El cálculo `total / dinerCount` se redondea apropiadamente manejando centavos residuales asignándolos al último comensal.

División por consumo (`byConsumption`): Cada comensal paga exactamente lo que ordenó, calculado a partir del campo `diner_id` de cada item. La función itera sobre todos los items agrupando por comensal y sumando subtotales.

División personalizada (`custom`): Los comensales acuerdan montos arbitrarios, útil cuando alguien quiere invitar o cuando el cálculo exacto no coincide con el deseo social. La interfaz permite ingresar montos manuales por comensal.

El registro de pagos individuales mediante `dinerPayments` permite escenarios mixtos donde algunos pagan en efectivo, otros con tarjeta, y quizás uno transfiere su parte. El sistema trackea cada contribución mediante `recordDinerPayment()` hasta que el total queda cubierto.

Integración con Mercado Pago

Para pagos electrónicos, pwaMenu integra con Mercado Pago mediante el modelo de Checkout Pro. El servicio `mercadoPago.ts` orquesta el flujo completo.

Cuando un comensal selecciona "Pagar con Mercado Pago", la función `initiatePayment()` solicita al backend la creación de una preferencia de pago vía `POST /api/billing/mercadopago/preference`. El backend, utilizando el SDK de Mercado Pago, genera una preferencia que incluye: items con nombre y precio, datos del pagador, URLs de retorno para éxito (`/payment/success`), fallo (`/payment/failure`) y estado pendiente (`/payment/pending`).

El comensal es redirigido a `sandbox_init_point` (ambiente de prueba) o `init_point` (producción) donde Mercado Pago presenta su checkout. El usuario puede pagar con tarjeta de crédito/débito, saldo de cuenta Mercado Pago, transferencia bancaria, o métodos alternativos según su país y configuración del comercio.

Una vez completado el pago, Mercado Pago redirige al usuario de vuelta a pwaMenu a la página `PaymentResult.tsx`. Los query parameters de la URL (`status`, `payment_id`, `external_reference`) permiten determinar el resultado: aprobado, rechazado, o pendiente de confirmación.

La página `PaymentResult` interpreta estos parámetros y muestra feedback apropiado: celebración visual para pagos aprobados, mensaje de error con opción de reintento para rechazados, explicación de estado pendiente para pagos que requieren confirmación adicional (transferencias, pagos en efectivo en puntos de pago).

El backend recibe notificación independiente del resultado mediante webhook de Mercado Pago en `POST /api/billing/mercadopago/webhook`. Este webhook garantiza que el registro de pago se actualice incluso si el usuario cierra el navegador antes de retornar a la aplicación. El `MERCADOPAGO_WEBHOOK_SECRET` valida la autenticidad de la notificación.

Pagos en Efectivo y Tarjeta Física

Para pagos que no atraviesan Mercado Pago (efectivo entregado al mozo o tarjeta física procesada en terminal del restaurante), la aplicación registra la intención de pago pero la confirmación ocurre desde pwaWaiter.

El comensal selecciona el método de pago (efectivo o tarjeta) y el monto correspondiente a su porción. Esta información se registra localmente y se muestra al mozo en su aplicación. El mozo procesa el pago físicamente, verifica el monto, y confirma la recepción en pwaWaiter mediante `POST /api/waiter/payments/{id}/confirm`.

La confirmación del mozo emite evento `PAYMENT_APPROVED` que actualiza el estado del Check en todos los dispositivos de la mesa. Cuando todos los pagos parciales suman el total, el Check se marca como pagado completamente y la sesión puede cerrarse.

Capítulo 8: La Arquitectura de Estado

El Patrón de Selectores Estables

React 19 con Zustand 5 introduce requisitos estrictos para evitar re-renders infinitos. El problema surge cuando un selector retorna una nueva referencia de objeto o array en cada invocación: React detecta "cambio" y re-renderiza, lo que invoca el selector nuevamente, creando un loop infinito.

La solución implementada en `tableStore/selectors.ts` emplea constantes de referencia estable para valores vacíos. En lugar de retornar `[]` directamente, los selectores retornan constantes definidas a nivel de módulo:

```typescript
const EMPTY_CART_ITEMS: CartItem[] = []
const EMPTY_DINERS: Diner[] = []
const EMPTY_ORDERS: OrderRecord[] = []

export const selectCartItems = (state: TableState) =>
  state.session?.cart_items ?? EMPTY_CART_ITEMS
```

Dado que la misma referencia de objeto se retorna en cada invocación cuando no hay datos, React detecta correctamente que no hay cambios y omite el re-render.

Para selectores que filtran o transforman datos, se implementa un patrón de caché manual. El selector mantiene una referencia al input previo y su resultado correspondiente en un objeto de caché a nivel de módulo:

```typescript
const myItemsCache = {
  items: null as CartItem[] | null,
  dinerId: null as string | null,
  result: EMPTY_CART_ITEMS
}
```

```
export const selectMyItems = (state: TableState) => {
  const items = state.session?.cart_items ?? EMPTY_CART_ITEMS
  const dinerId = state.currentDiner?.id ?? null

  if (items === myItemsCache.items && dinerId === myItemsCache.dinerId) {
    return myItemsCache.result
  }

  const filtered = items.filter(i => i.diner_id === dinerId)
  myItemsCache.items = items
  myItemsCache.dinerId = dinerId
  myItemsCache.result = filtered.length > 0 ? filtered : EMPTY_CART_ITEMS
  return myItemsCache.result
}
```

Este patrón, aunque verbose, garantiza estabilidad referencial sin depender de bibliotecas externas de memoización y sin introducir hooks adicionales que complicarían el uso de los selectores.

La Persistencia de Sesión con Validación

El middleware `persist` de Zustand serializa automáticamente partes del estado a localStorage después de cada modificación. La configuración `partialize` define exactamente qué se persiste, excluyendo estado transitorio:

```typescript
persist(
  (set, get) => ({ /* estado y acciones */ }),
  {
    name: 'table-session',
```

```
    partialize: (state) => ({
      session: state.session,
      currentDiner: state.currentDiner,
      orders: state.orders,
      currentRound: state.currentRound,
      // Excluidos: isLoading, isSubmitting, submitSuccess, errors
    }),
    onRehydrateStorage: () => (state, error) => {
      if (error) {
        console.error('Rehydration failed:', error)
        // MED-02 FIX: Limpiar estado corrupto
        localStorage.removeItem('table-session')
        return
      }
    }
  }
)
```

La rehidratación incluye validaciones de expiración. El helper `isSessionExpired()` verifica `session.last_activity` contra `Date.now()` considerando el umbral de 8 horas definido en `SESSION.EXPIRY_HOURS`. Una sesión almacenada de hace ocho horas sin actividad probablemente corresponde a una comida que ya terminó; el sistema la descarta automáticamente en lugar de intentar reconectar a una mesa que probablemente tiene otros ocupantes.

La Coordinación Multi-Tab

Un mismo comensal puede tener la aplicación abierta en múltiples pestañas del navegador. El sistema detecta cambios en localStorage desde otras pestañas mediante el evento `storage` del objeto `window`, configurado en `App.tsx`:

```typescript
useEffect(() => {
```

```
  const handleStorageChange = (e: StorageEvent) => {
    if (e.key === 'table-session' && e.newValue) {
      const newState = JSON.parse(e.newValue)
      // Sincronizar estado desde otra pestaña
      syncFromStorage(newState)
    } else if (e.key === 'table-session' && !e.newValue) {
      // Otra pestaña cerró sesión
      clearSession()
    }
  }

  window.addEventListener('storage', handleStorageChange)
  return () => window.removeEventListener('storage', handleStorageChange)
}, [])
```

La estrategia de merge para items del carrito utiliza un Map con clave compuesta para deduplicación. La pestaña que detecta el cambio es receptora, considerando a la otra pestaña como fuente de verdad. Cada pestaña mantiene su propia identidad de comensal (`currentDiner`), pero comparte el carrito y los pedidos. Si una pestaña detecta que otra abandonó la sesión (el valor es null), limpia su propio estado.

Capítulo 9: El Cliente API Defensivo

La Protección SSRF

El cliente API en `services/api.ts` valida rigurosamente que las URLs de destino correspondan a hosts permitidos, previniendo ataques SSRF (Server-Side Request Forgery) donde un input malicioso podría redirigir peticiones a servicios internos de la infraestructura.

La validación de seguridad opera en múltiples capas. Primero, la URL se parsea y normaliza:

```typescript
const url = new URL(endpoint, API_BASE_URL)

// Bloquear credenciales embebidas en URL
if (url.username || url.password) {
  throw new ApiError('URL credentials not allowed', 400)
}
```

Las direcciones IP directas están bloqueadas para prevenir acceso a servicios internos:

```typescript
const BLOCKED_IP_PATTERNS = [
  /^127\./,          // Localhost IPv4
  /^10\./,           // Private class A
  /^172\.(1[6-9]|2\d|3[01])\./,  // Private class B
  /^192\.168\./,     // Private class C
  /^169\.254\./,     // Link-local
  /^0\./,            // Current network
  /^\[::1\]/,        // Localhost IPv6
  /^\[fc/i,          // IPv6 unique local
  /^\[fd/i,          // IPv6 unique local
  /^\[fe80:/i,       // IPv6 link-local
]

if (BLOCKED_IP_PATTERNS.some(p => p.test(url.hostname))) {
  throw new ApiError('IP addresses not allowed', 400)
}
```

Los puertos están restringidos a un conjunto conocido de puertos HTTP válidos:

```typescript
const ALLOWED_PORTS = ['', '80', '443', '8000', '8080', '8443', '3000', '5000']
const normalizedPort = url.port || (url.protocol === 'https:' ? '443' : '80')

if (!ALLOWED_PORTS.includes(normalizedPort)) {
  throw new ApiError(`Port ${url.port} not allowed`, 400)
}
```

El hostname debe coincidir exactamente con los hosts permitidos configurados en `API_CONFIG.ALLOWED_HOSTS`, sin permitir subdominios ni variantes:

```typescript
if (!ALLOWED_HOSTS.includes(url.hostname)) {
  throw new ApiError(`Host ${url.hostname} not allowed`, 400)
}
```

La Deduplicación de Peticiones

El sistema de deduplicación previene race conditions donde clicks rápidos podrían enviar múltiples veces la misma petición. Un Map `pendingRequests` registra cada petición en vuelo con una clave compuesta de método y endpoint:

```typescript
const pendingRequests = new Map<string, Promise<unknown>>()
const MAX_PENDING = 100
const CLEANUP_INTERVAL = 60000  // 1 minuto

function getRequestKey(method: string, url: string, body?: unknown): string {
```

```
  return `${method}:${url}:${JSON.stringify(body ?? '')}`
}

async function fetchWithDedup<T>(url: string, options: RequestInit): Promise<T> {
  const key = getRequestKey(options.method ?? 'GET', url, options.body)

  // Si existe petición idéntica en vuelo, reutilizar su promesa
  if (pendingRequests.has(key)) {
    return pendingRequests.get(key) as Promise<T>
  }

  // Límite de peticiones pendientes (HIGH-02 FIX)
  if (pendingRequests.size >= MAX_PENDING) {
    cleanupOldestPending()
  }

  const promise = fetch(url, options).then(/* ... */).finally(() => {
    pendingRequests.delete(key)
  })

  pendingRequests.set(key, promise)
  return promise
}
```

La comparación de body utiliza serialización JSON directa en lugar de hashing simple, evitando colisiones donde bodies diferentes produjeran el mismo hash (HIGH-05 FIX). El límite de 100 peticiones concurrentes y la limpieza periódica previenen crecimiento descontrolado del Map en escenarios patológicos.

El Manejo de Expiración de Sesión

Cuando el backend responde con 401 (no autorizado), el cliente API detecta si la petición usaba autenticación de mesa y dispara el flujo de manejo de sesión expirada:

```typescript
if (response.status === 401 && options.tableAuth) {
  // Token de mesa expirado o inválido
  onTokenExpired?.()
  throw new AuthError('Session expired', 'errors.sessionExpired')
}
```

El callback `onTokenExpired` está conectado al store, que presenta un `SessionExpiredModal` informativo. El modal explica que la sesión expiró (los tokens de mesa tienen vida de 3 horas) y sugiere al usuario escanear nuevamente el QR para obtener una nueva sesión.

Este flujo reconoce que el usuario puede simplemente haber dejado la aplicación abierta demasiado tiempo. La experiencia guía al usuario hacia la resolución en lugar de mostrar errores técnicos confusos.

Capítulo 10: La Internacionalización

Tres Idiomas, Una Experiencia

pwaMenu soporta español, inglés y portugués, reflejando la realidad multilingüe de restaurantes que reciben turistas internacionales. El sistema i18next se configura en `i18n/index.ts` con recursos de traducción en archivos JSON separados: `es.json`, `en.json`, `pt.json`.

El detector de idioma personalizado extiende el detector estándar de i18next con validación adicional:

```typescript
```

```
const validatedLanguageDetector = new LanguageDetector()


validatedLanguageDetector.addDetector({
  name: 'validatedLocalStorage',
  lookup() {
    const cached = localStorage.getItem('i18nextLng')
    // Solo retornar si es idioma válido
    return SUPPORTED_LANGUAGES.includes(cached) ? cached : null
  },
  cacheUserLanguage(lng) {
    // Solo cachear si es idioma válido
    if (SUPPORTED_LANGUAGES.includes(lng)) {
      localStorage.setItem('i18nextLng', lng)
    }
  }
})
```

Esta defensa previene corrupción de localStorage que podría dejar la aplicación en un estado de idioma inválido (por ejemplo, si un script malicioso o un bug escribiera un valor incorrecto).

La configuración de fallback establece español como idioma de respaldo universal:

```typescript
i18n.init({
  fallbackLng: {
    en: ['es'],      // Inglés fallback a español
    pt: ['es'],      // Portugués fallback a español
    default: ['es']  // Cualquier otro fallback a español
  },
  interpolation: {
```

```
    escapeValue: false  // React ya escapa
  }
})
```

Namespaces de Traducción

Los archivos de traducción organizan las claves en namespaces semánticos que facilitan la búsqueda y mantenimiento:

Namespace	Contenido
`general`	Textos comunes: botones, títulos, estados
`menu`	Catálogo: categorías, productos, descripciones
`cart`	Carrito: items, cantidades, totales
`payment`	Pago: métodos, división, confirmación
`errors`	Mensajes de error: validación, red, sesión
`filters`	Filtros: alérgenos, dietas, preferencias
`loyalty`	Fidelización: reconocimiento, registro, puntos
`roundConfirmation`	Confirmación grupal: propuesta, estados, acciones
`accessibility`	Accesibilidad: labels, announcements, hints
`bottomNav`	Navegación inferior: mozo, pedidos, cuenta

La Traducción de Productos

Más allá de la interfaz, el sistema soporta productos con nombres y descripciones traducidos. El modelo `Product` en el backend puede incluir campos localizados, y el hook `useProductTranslation` selecciona la versión apropiada según el idioma activo:

```typescript
const { t, i18n } = useTranslation()
```

```
const getLocalizedProduct = (product: Product) => ({
  ...product,
  name: product[`name_${i18n.language}`] ?? product.name,
  description: product[`description_${i18n.language}`] ?? product.description
})
```

El selector de idioma, ubicado discretamente en el Header, permite cambiar la preferencia manualmente mediante componentes `LanguageSelector` (dropdown) y `LanguageFlagSelector` (banderas). Esta configuración persiste en localStorage, asegurando que un turista anglófono no tenga que reconfigurar el idioma cada vez que escanea un nuevo QR en otra visita.

Capítulo 11: La Progressive Web App

La Instalabilidad

Como Progressive Web App, pwaMenu puede instalarse en la pantalla de inicio del dispositivo, proporcionando acceso rápido y experiencia de aplicación nativa sin barras de navegador. El hook `useInstallPrompt` detecta la disponibilidad del prompt de instalación:

```typescript
const [installPrompt, setInstallPrompt] = useState<BeforeInstallPromptEvent | null>(null)

useEffect(() => {
  const handler = (e: BeforeInstallPromptEvent) => {
    e.preventDefault()  // Prevenir prompt automático
    setInstallPrompt(e) // Guardarlo para uso controlado
  }

  window.addEventListener('beforeinstallprompt', handler)
```

```
  return () => window.removeEventListener('beforeinstallprompt', handler)
}, [])

const promptInstall = () => {
  installPrompt?.prompt()
}
```

El componente `InstallBanner` presenta un banner discreto invitando a instalar la aplicación cuando el prompt está disponible. Al tocar "Instalar", se invoca el prompt nativo del navegador.

El manifiesto de aplicación en `manifest.webmanifest` (generado por vite-plugin-pwa) define los metadatos de instalación:

```json
{
  "name": "Sabor - Menú Digital",
  "short_name": "Sabor",
  "description": "Menú digital colaborativo",
  "theme_color": "#f97316",
  "background_color": "#0a0a0a",
  "display": "standalone",
  "orientation": "portrait",
  "icons": [
    { "src": "pwa-192x192.png", "sizes": "192x192", "type": "image/png" },
    { "src": "pwa-512x512.png", "sizes": "512x512", "type": "image/png", "purpose": "maskable"
}
  ],
  "shortcuts": [
    { "name": "Ver Menú", "url": "/", "icons": [...] },
    { "name": "Productos Destacados", "url": "/?section=featured", "icons": [...] },
```

```
    { "name": "Bebidas", "url": "/?category=drinks", "icons": [...] },
    { "name": "Mi Carrito", "url": "/?cart=open", "icons": [...] }
  ]
}
```

El Service Worker y las Estrategias de Caché

El service worker generado por Workbox implementa estrategias de caché diferenciadas según el tipo de recurso y su volatilidad:

Precache (recursos de la aplicación): JavaScript, CSS, HTML de la aplicación se descargan durante la instalación del service worker. En visitas posteriores, se sirven instantáneamente desde caché mientras el service worker verifica actualizaciones en segundo plano.

CacheFirst (imágenes de productos): Las imágenes de productos provenientes de CDN externos utilizan estrategia que prioriza velocidad sobre frescura. Expiración de 30 días, límite de 60 entradas:

```typescript
{
  urlPattern: /^https:\/\/images\.unsplash\.com\//,
  handler: 'CacheFirst',
  options: {
    cacheName: 'product-images',
    expiration: { maxEntries: 60, maxAgeSeconds: 30 * 24 * 60 * 60 }
  }
}
```

CacheFirst (Google Fonts): Fuentes tipográficas con expiración de un año. Las URLs de fuentes incluyen hashes de versión que cambian cuando la fuente se actualiza:

```typescript
{
  urlPattern: /^https:\/\/fonts\.googleapis\.com\//,
  handler: 'CacheFirst',
  options: {
    cacheName: 'google-fonts',
    expiration: { maxAgeSeconds: 365 * 24 * 60 * 60 }
  }
}
```

NetworkFirst (APIs): Llamadas a la API del backend intentan obtener datos frescos del servidor con timeout de 5 segundos. Si la red falla o tarda demasiado, se sirve la última versión cacheada:

```typescript
{
  urlPattern: /^https:\/\/api\..*\/api\//,
  handler: 'NetworkFirst',
  options: {
    cacheName: 'api-cache',
    networkTimeoutSeconds: 5,
    expiration: { maxAgeSeconds: 60 * 60 }  // 1 hora
  }
}
```

Las Actualizaciones Transparentes

Cuando el administrador despliega una nueva versión de la aplicación, el service worker la detecta durante su ciclo de actualización. La configuración `registerType: 'prompt'` presenta una notificación al usuario en lugar de actualizar silenciosamente:

```typescript
// App.tsx
const { needRefresh, updateServiceWorker } = useRegisterSW({
  onNeedRefresh() {
    // Mostrar banner de actualización disponible
    setShowUpdateBanner(true)
  },
  onOfflineReady() {
    toast.info(t('general.offlineReady'))
  }
})

const handleUpdate = () => {
  updateServiceWorker(true)  // true = reload after update
}
```

Este patrón respeta la experiencia del usuario. Si está en medio de configurar un pedido, puede continuar con la versión actual y actualizar cuando termine. Si prefiere tener la última versión inmediatamente, puede aceptar la actualización que recargará la página con el nuevo código.

El Funcionamiento Offline

El componente `NetworkStatus` monitorea la conectividad mediante el evento `online`/`offline` del navegador:

```typescript
const [isOnline, setIsOnline] = useState(navigator.onLine)

useEffect(() => {
```

```
  const handleOnline = () => setIsOnline(true)
  const handleOffline = () => setIsOnline(false)

  window.addEventListener('online', handleOnline)
  window.addEventListener('offline', handleOffline)

  return () => {
    window.removeEventListener('online', handleOnline)
    window.removeEventListener('offline', handleOffline)
  }
}, [])
```

Cuando el dispositivo pierde conexión, un indicador visual informa al usuario. El hook `useOnlineStatus` expone este estado a cualquier componente que necesite adaptar su comportamiento.

Las operaciones de carrito que fallan por falta de conectividad se encolan en el `OfflineQueue`. Cuando la conectividad se restaura, la cola reproduce las operaciones en orden FIFO, reconciliando el estado local con el servidor. Este patrón permite que usuarios en zonas con conectividad intermitente continúen interactuando con la aplicación.

La página `offline.html` servida cuando la navegación falla completamente presenta un mensaje amigable indicando la falta de conexión y sugiriendo verificar la red WiFi o datos móviles. Esta página se precachea durante la instalación del service worker para garantizar su disponibilidad incluso sin ninguna conectividad.

Capítulo 12: Accesibilidad y Experiencia de Usuario

Estándares de Accesibilidad

La aplicación implementa estándares WCAG 2.1 nivel AA. Todos los elementos interactivos poseen labels accesibles mediante texto visible, atributo `aria-label`, o `aria-labelledby` apuntando a un elemento descriptivo.

Los modales implementan focus trap mediante el hook `useFocusTrap`: al abrirse, el foco se mueve al primer elemento focusable del modal; la navegación con Tab cicla dentro del modal sin escapar a elementos detrás del backdrop; al cerrarse, el foco retorna al elemento que abrió el modal.

```typescript
const useFocusTrap = (isOpen: boolean, containerRef: RefObject<HTMLElement>) => {
  useEffect(() => {
    if (!isOpen || !containerRef.current) return

    const focusableElements = containerRef.current.querySelectorAll(
      'button, [href], input, select, textarea, [tabindex]:not([tabindex="-1"])'
    )
    const firstElement = focusableElements[0] as HTMLElement
    const lastElement = focusableElements[focusableElements.length - 1] as HTMLElement

    const handleKeyDown = (e: KeyboardEvent) => {
      if (e.key !== 'Tab') return

      if (e.shiftKey && document.activeElement === firstElement) {
        e.preventDefault()
        lastElement.focus()
      } else if (!e.shiftKey && document.activeElement === lastElement) {
        e.preventDefault()
        firstElement.focus()
      }
    }

    firstElement?.focus()
```

```
    document.addEventListener('keydown', handleKeyDown)
    return () => document.removeEventListener('keydown', handleKeyDown)
  }, [isOpen, containerRef])
}
```

El hook `useEscapeKey` permite cerrar modales con la tecla Escape, soportando un estado `disabled` para prevenir cierre durante operaciones asíncronas pendientes.

Anuncios para Lectores de Pantalla

El hook `useAriaAnnounce` permite anunciar cambios de estado a lectores de pantalla mediante una región ARIA live que se crea dinámicamente:

```typescript
const useAriaAnnounce = () => {
  const regionRef = useRef<HTMLDivElement | null>(null)

  useEffect(() => {
    // Crear región live al montar
    const region = document.createElement('div')
    region.setAttribute('role', 'status')
    region.setAttribute('aria-live', 'polite')
    region.setAttribute('aria-atomic', 'true')
    region.className = 'sr-only'  // Visualmente oculto
    document.body.appendChild(region)
    regionRef.current = region

    return () => region.remove()
  }, [])

  const announce = useCallback((message: string) => {
```

```
    if (regionRef.current) {
      regionRef.current.textContent = ''  // Reset
      requestAnimationFrame(() => {
        if (regionRef.current) {
          regionRef.current.textContent = message
        }
      })
    }
  }, [])

  return announce
}
```

Cuando un producto se añade al carrito, se anuncia "Hamburguesa añadida al carrito". Cuando un pedido cambia de estado, se anuncia "Tu pedido está siendo preparado". Estos anuncios proporcionan feedback a usuarios que no pueden percibir cambios visuales.

Los iconos decorativos incluyen `aria-hidden="true"` para que lectores de pantalla los ignoren. Los iconos significativos (como el badge de cantidad en el carrito) incluyen texto alternativo mediante spans con clase `sr-only`.

Diseño Táctil y Viewport Móvil

Los componentes táctiles respetan tamaños mínimos de 44x44 píxeles según guías de accesibilidad de Apple y Google, garantizando objetivos de toque cómodos en dispositivos móviles.

Las clases de safe area de Tailwind (`safe-area-top`, `safe-area-bottom`) garantizan que el contenido no quede oculto tras notches de iPhone o barras de navegación de Android:

```css
.safe-area-bottom {
  padding-bottom: env(safe-area-inset-bottom, 0);
```

}
```

Todos los contenedores de página incluyen `overflow-x-hidden w-full max-w-full` para prevenir scroll horizontal accidental en móviles, un problema común cuando elementos con ancho fijo exceden el viewport.

---

## Capítulo 13: El Asistente con Inteligencia Artificial

### La Arquitectura del Chat

El directorio `AIChat/` contiene el chatbot de asistencia que permite a los comensales hacer preguntas sobre el menú, pedir recomendaciones o solicitar información nutricional. El componente principal `index.tsx` presenta una interfaz de chat con historial de mensajes, campo de entrada y sugerencias de preguntas frecuentes.

El sistema utiliza el endpoint de RAG (Retrieval-Augmented Generation) del backend que combina búsqueda semántica sobre la base de conocimiento del restaurante con generación de respuestas mediante modelo de lenguaje. Las preguntas del usuario se envían a `/api/public/rag/chat` junto con el contexto de la sesión actual.

### Handlers de Respuesta Especializados

El módulo `responseHandlers.ts` implementa un patrón de estrategia para procesar diferentes tipos de respuesta del modelo de IA:

```typescript
interface ResponseHandler {
  canHandle(response: AIResponse): boolean
  render(response: AIResponse): ReactNode
}
```
```

```
const productRecommendationHandler: ResponseHandler = {
  canHandle: (r) => r.type === 'product_recommendation',
  render: (r) => <ProductCards products={r.products} />
}

const nutritionalInfoHandler: ResponseHandler = {
  canHandle: (r) => r.type === 'nutritional_info',
  render: (r) => <NutritionTable data={r.nutrition} />
}

const textResponseHandler: ResponseHandler = {
  canHandle: () => true,  // Fallback
  render: (r) => <TextMessage text={r.text} />
}

const handlers = [productRecommendationHandler, nutritionalInfoHandler,
textResponseHandler]

const renderResponse = (response: AIResponse) => {
  const handler = handlers.find(h => h.canHandle(response))
  return handler?.render(response)
}
```

Las recomendaciones de productos se renderizan como cards interactivos que permiten agregar directamente al carrito. La información nutricional se presenta en tablas formateadas. Las respuestas de texto plano se muestran como párrafos estilizados con soporte para markdown básico.

Gestión de Estado del Chat

El contador de IDs de mensaje implementa un reset periódico cada 60 segundos para prevenir crecimiento indefinido en conversaciones largas (LOW-01 FIX):

```typescript
const messageIdCounter = useRef(0)
const lastResetTime = useRef(Date.now())

const getNextMessageId = () => {
  const now = Date.now()
  if (now - lastResetTime.current > 60000) {
    messageIdCounter.current = 0
    lastResetTime.current = now
  }
  return `msg-${++messageIdCounter.current}`
}
```

El historial de mensajes se mantiene en estado local del componente. Al cerrar y reabrir el chat, el historial se preserva durante la sesión pero no persiste en localStorage, reconociendo que las conversaciones con IA son típicamente efímeras.

Capítulo 14: Testing y Calidad de Código

Infraestructura de Testing

El framework de testing combina Vitest como test runner y Testing Library para renderizado de componentes React. La configuración en `vite.config.ts` habilita entorno jsdom para simular APIs del navegador en Node.js.

Los tests se ubican junto al código fuente con extensión `.test.ts`:

```
```

```
src/
├── hooks/
│   ├── useCartSync.ts
│   └── useCartSync.test.ts
├── stores/
│   └── tableStore/
│       ├── store.ts
│       ├── store.test.ts
│       ├── helpers.ts
│       └── helpers.test.ts
└── services/
    ├── api.ts
    └── api.test.ts
```

Patrones de Testing

Los tests unitarios verifican funciones puras como los helpers del tableStore:

```typescript
describe('calculateCartTotal', () => {
  it('should sum prices correctly', () => {
    const items = [
      { price: 100, quantity: 2 },
      { price: 50, quantity: 1 }
    ]
    expect(calculateCartTotal(items)).toBe(250)
  })

  it('should return 0 for empty cart', () => {
    expect(calculateCartTotal([])).toBe(0)
```

```
  })
})
```

Los tests de hooks utilizan `renderHook` de Testing Library:

```typescript
describe('useCartSync', () => {
  it('should add item when receiving CART_ITEM_ADDED', async () => {
    const mockWS = createMockWebSocket()
    const { result } = renderHook(() => useCartSync())

    // Simular evento WebSocket
    mockWS.emit('CART_ITEM_ADDED', { item: mockItem })

    await waitFor(() => {
      expect(result.current.items).toContainEqual(mockItem)
    })
  })
})
```

Los tests de servicios mockean fetch para simular respuestas de API:

```typescript
describe('api.fetchMenu', () => {
  beforeEach(() => {
    global.fetch = jest.fn()
  })

  it('should handle network errors gracefully', async () => {
```

```
    global.fetch.mockRejectedValue(new Error('Network error'))

    await expect(menuAPI.fetchMenu('test-slug'))
      .rejects.toThrow('errors.network')
  })
})
```

Scripts de Ejecución

```bash
npm test              # Watch mode - re-ejecuta tests afectados al modificar archivos
npm run test:run      # Ejecución única para CI/CD pipelines
npm run test:coverage # Genera reporte de cobertura de código
```

La cobertura actual se enfoca en las áreas más críticas: lógica de carrito, sincronización de estado, manejo de errores, validación de seguridad. Los componentes visuales tienen menor cobertura dado que su corrección se verifica más efectivamente mediante testing manual y visual.

Capítulo 15: Métricas de Rendimiento

Web Vitals

El módulo `utils/webVitals.ts` recolecta métricas de rendimiento real de usuarios mediante la biblioteca web-vitals:

```typescript
import { onLCP, onFID, onCLS, onFCP, onTTFB } from 'web-vitals'
```

```
const metrics: Record<string, number> = {}

export const initWebVitals = () => {
  onLCP((metric) => { metrics.lcp = metric.value })
  onFID((metric) => { metrics.fid = metric.value })
  onCLS((metric) => { metrics.cls = metric.value })
  onFCP((metric) => { metrics.fcp = metric.value })
  onTTFB((metric) => { metrics.ttfb = metric.value })
}

export const getMetrics = () => ({ ...metrics })
```

Las métricas se almacenan en sessionStorage durante la visita. Los targets de rendimiento establecidos son:

Métrica	Target	Descripción
LCP	< 2.5s	Largest Contentful Paint - tiempo hasta elemento visual principal
FID	< 100ms	First Input Delay - latencia de primera interacción
CLS	< 0.1	Cumulative Layout Shift - estabilidad visual
FCP	< 1.8s	First Contentful Paint - primer contenido visible
TTFB	< 600ms	Time to First Byte - respuesta inicial del servidor

Optimizaciones Implementadas

Lazy Loading de Componentes: Los modales, el chat de IA, los filtros avanzados y otros componentes secundarios se cargan mediante `React.lazy()` solo cuando el usuario los requiere:

```typescript
```

```
const ProductDetailModal = lazy(() => import('./ProductDetailModal'))
const AdvancedFiltersModal = lazy(() => import('./AdvancedFiltersModal'))
const AIChat = lazy(() => import('./AIChat'))
```

Lazy Loading de Imágenes: Las imágenes de productos implementan lazy loading nativo:

```typescript
<img
  src={product.image}
  loading="lazy"
  decoding="async"
  alt={product.name}
/>
```

Selectores Memoizados: Como se describió anteriormente, los selectores de Zustand con caches manuales previenen re-cálculos innecesarios.

Throttling de Operaciones: Las operaciones de carrito utilizan throttling de 100-200ms (definido en `constants/timing.ts`) para prevenir ráfagas de peticiones por clicks rápidos.

Reflexión Final: La Digitalización de lo Social

pwaMenu no es simplemente un catálogo digital ni una aplicación de pedidos. Es una herramienta que respeta y amplifica la naturaleza inherentemente social de compartir una comida. El carrito compartido no es una limitación técnica sino una decisión de diseño que refleja cómo las personas realmente comen en grupo: viendo lo que otros piden, sugiriendo opciones, decidiendo juntos cuándo enviar el pedido.

Las protecciones de alérgenos van más allá de los checkboxes típicos porque una alergia severa no es una "preferencia": es una cuestión de seguridad que merece consideración seria. El

reconocimiento de dispositivos ofrece personalización sin exigir registro porque la hospitalidad genuina no comienza con formularios.

La arquitectura técnica —React 19 con actualizaciones optimistas, Zustand con selectores estables, WebSocket con reconexión resiliente, Service Worker con estrategias de caché inteligentes— no existe por amor a la tecnología. Cada decisión responde a una pregunta fundamental: ¿cómo hace esto que la experiencia del comensal sea más agradable, más segura, o más conveniente?

Cuando la tecnología desaparece y solo queda la experiencia fluida de elegir y compartir comida con personas queridas, el software ha cumplido su propósito. pwaMenu aspira a ese ideal: ser tan invisible como una buena carta de restaurante, pero infinitamente más capaz.

Documento técnico narrativo del proyecto pwaMenu. Última actualización: Febrero 2026.

La Capa Compartida: Fundamento Arquitectónico del Backend

Versión 3.0 - Febrero 2026

Introducción

La carpeta backend/shared/ constituye el núcleo fundacional del sistema Integrador, representando aproximadamente nueve mil líneas de código distribuidas en cuarenta y un archivos especializados organizados en cuatro módulos principales. Esta capa trasciende la noción convencional de utilidades compartidas para convertirse en el verdadero sistema nervioso de la aplicación, donde convergen todas las decisiones fundamentales de seguridad, configuración, comunicación inter-servicios y gestión de estado distribuido.

El diseño de esta capa adhiere rigurosamente a los principios de Clean Architecture, ocupando el círculo más interno del sistema. Las capas externas, tanto la REST API como el WebSocket Gateway, dependen exclusivamente de este núcleo compartido, pero nunca ocurre lo inverso. Esta dirección unidireccional de dependencias garantiza que modificaciones en la lógica de presentación o en los endpoints no perturben los servicios fundamentales, permitiendo evolución independiente de cada componente sin efectos secundarios inesperados.

La arquitectura interna de la capa refleja una organización meticulosa por responsabilidad funcional que ha evolucionado a través de múltiples iteraciones de refactorización. El módulo de configuración, con aproximadamente mil ciento trece líneas, centraliza todas las variables de entorno, constantes de dominio y sistemas de logging estructurado. El módulo de seguridad, con mil setecientas veinticinco líneas, encapsula la totalidad de la lógica de autenticación desde verificación de tokens JWT hasta revocación en tiempo real mediante Redis y rate limiting. El módulo de infraestructura, el más extenso con tres mil trescientas cuarenta y ocho líneas, gestiona las conexiones a PostgreSQL y Redis incluyendo un sofisticado sistema de eventos basado en publicación/suscripción con circuit breaker para resiliencia. Finalmente, el módulo de utilidades con dos mil setecientas cincuenta y nueve líneas provee excepciones estandarizadas, validadores de seguridad, health checks con timeout, y los schemas Pydantic que definen el contrato de la API.

Capítulo 1: El Sistema de Configuración

El módulo de configuración reside en el directorio config/ y comprende tres archivos fundamentales que establecen la fuente única de verdad para todo el sistema. Esta centralización elimina completamente la dispersión de valores hardcodeados que plagaba versiones anteriores del código y que generaba inconsistencias difíciles de rastrear entre diferentes partes de la aplicación.

El archivo settings.py implementa el patrón de configuración centralizada mediante Pydantic BaseSettings, una aproximación que combina la flexibilidad de variables de entorno con la seguridad de tipos estáticos de Python. La clase Settings, que abarca aproximadamente ciento setenta y nueve líneas de código cuidadosamente documentado, actúa como el único punto de verdad para todas las configuraciones del sistema.

La conectividad a bases de datos queda definida mediante el atributo database_url, cuyo valor por defecto construye una cadena de conexión PostgreSQL apuntando a localhost en el puerto 5432 con la base de datos integrador. Esta configuración puede sobreescribirse mediante la variable de entorno DATABASE_URL, permitiendo que el mismo código opere en entornos de desarrollo, staging y producción sin modificación alguna. De manera análoga, redis_url establece la conexión al broker de mensajería utilizando el puerto 6380 como valor por defecto en lugar del convencional 6379, una decisión deliberada que coincide con la configuración de Docker Compose

del proyecto y evita conflictos con instalaciones Redis locales preexistentes.

El subsistema de autenticación JWT recibe tratamiento particularmente detallado en la configuración. Los tokens de acceso mantienen una vida útil deliberadamente corta de quince minutos, expresada en el atributo jwt_access_token_expire_minutes. Esta decisión reduce drásticamente la ventana de exposición en caso de que un token sea comprometido, ya que un atacante tendría apenas un cuarto de hora para explotar credenciales robadas antes de que expiren naturalmente. Los tokens de refresco extienden esta duración a siete días mediante jwt_refresh_token_expire_days, permitiendo sesiones prolongadas sin requerir intervención explícita del usuario para re-autenticarse mientras mantienen la seguridad mediante tokens de acceso de corta duración.

Los tokens de mesa diseñados para comensales que escanean códigos QR recibieron una reducción significativa de tiempo de vida en el artefacto CRIT-04 FIX. Originalmente configurados para ocho horas, estos tokens ahora expiran tras tres horas de uso, una medida de hardening que limita la exposición en contextos públicos donde múltiples personas podrían observar o fotografiar un código QR antes de que sea utilizado legítimamente por el comensal autorizado.

La configuración de cookies HttpOnly introducida en SEC-09 representa una medida crítica de mitigación contra ataques de Cross-Site Scripting. Los parámetros cookie_secure, cookie_samesite y cookie_domain controlan el comportamiento detallado

de estas cookies que almacenan los tokens de refresco. En ambiente de desarrollo cookie_secure permanece en False para permitir transmisión sobre HTTP local sin cifrar, pero en producción este valor debe activarse para requerir HTTPS exclusivamente. El valor lax de cookie_samesite ofrece un balance calculado entre protección CSRF y usabilidad, permitiendo que las cookies se envíen durante navegación top-level mientras bloquean requests cross-site automáticos.

La configuración de WebSocket incorpora límites operacionales críticos para mantener la estabilidad del sistema bajo carga sostenida. El parámetro ws_max_connections_per_user establece un límite de tres conexiones simultáneas por usuario, previniendo el agotamiento de recursos por clientes problemáticos que podrían abrir múltiples pestañas del navegador. El límite global ws_max_total_connections de quinientas conexiones establece un techo absoluto de capacidad que protege al servidor de colapso por sobrecarga. El rate limiting por conexión expresado en ws_message_rate_limit con treinta mensajes por segundo previene ataques de denegación de servicio a nivel de protocolo WebSocket.

Los pools de Redis reciben configuración diferenciada para operaciones síncronas y asíncronas, una arquitectura documentada como LOAD-LEVEL2. El pool asíncrono con redis_pool_max_connections configurado en cincuenta conexiones soporta las operaciones de publicación/suscripción y eventos en tiempo real donde la naturaleza no bloqueante es esencial. El pool síncrono con redis_sync_pool_max_connections en veinte conexiones atiende operaciones inherentemente

bloqueantes como verificación de tokens contra blacklist y rate limiting. Esta separación evita que operaciones síncronas que deben esperar respuesta antes de continuar degraden la latencia del sistema de eventos en tiempo real.

El método validate_production_secrets implementa una validación de arranque que previene despliegues inseguros en producción. Cuando el ambiente está configurado como production, el sistema rechaza iniciar si detecta que JWT_SECRET o TABLE_TOKEN_SECRET contienen valores por defecto conocidos o strings con menos de treinta y dos caracteres. Esta validación actúa como última línea de defensa contra configuraciones inseguras que podrían pasar desapercibidas durante el proceso de despliegue.

Capítulo 2: Logging Estructurado y Observabilidad

El módulo logging.py con aproximadamente cuatrocientas dieciocho líneas de código implementa un sistema de logging dual que adapta su formato según el ambiente de ejecución detectado, proporcionando trazabilidad completa de operaciones con correlación de requests distribuidos.

La clase StructuredFormatter produce logs en formato JSON para ambientes de producción, donde cada entrada incluye timestamp, nivel, mensaje, nombre del logger, identificador de request correlacionado, y cualquier metadata adicional como campos JSON independientes. Este formato facilita enormemente la ingestión por

herramientas de observabilidad como ELK Stack, CloudWatch, Datadog o cualquier sistema capaz de parsear JSON estructurado. Los campos adicionales incluyen información de excepción cuando está disponible y ubicación del código fuente en modo debug para facilitar el diagnóstico.

La clase DevelopmentFormatter genera logs coloreados y legibles para desarrollo local, donde la prioridad es la comprensión humana rápida durante sesiones de debugging. Los colores varían según el nivel del log: cian para debug, verde para información, amarillo para warnings, rojo para errores, y magenta para críticos. El identificador de request aparece atenuado como prefijo entre corchetes para no distraer del contenido principal mientras permanece disponible para correlación manual.

La clase StructuredLogger extiende el logger estándar de Python para soportar contexto adicional mediante keyword arguments. Esta capacidad transforma logs simples en registros ricos en metadata que facilitan debugging post-mortem y análisis de comportamiento del sistema. Una llamada con mensaje User logged in y argumentos user_id, branch_id y role produce una entrada que incluye toda esa información como campos JSON independientes, permitiendo filtrado y agregación en sistemas de log management.

El sistema incluye funciones especializadas para enmascaramiento de información personalmente identificable identificadas como SHARED-HIGH-02 FIX. La función mask_email transforma direcciones como user@example.com en us***@example.com, preservando suficiente información para identificar

aproximadamente al usuario durante debugging mientras protege su identidad completa en logs que podrían ser accesibles a personal de operaciones. La función mask_jti trunca identificadores JWT a sus primeros ocho caracteres, suficientes para correlacionar logs sin exponer el identificador completo que podría ser utilizado para impersonación si los logs se filtraran. La función mask_user_id oculta parcialmente identificadores numéricos de usuario mostrando solo los primeros dígitos.

Las funciones de auditoría de seguridad implementadas bajo SEC-LOW-03 FIX proporcionan un trail estructurado para eventos críticos que requieren retención extendida y potencial análisis forense. La función audit_ws_connection registra conexiones y desconexiones WebSocket con detalles como user_id, session_id, origen de la conexión y razón de cierre. La función audit_auth_event captura intentos de login exitosos o fallidos con dirección IP y user agent. La función audit_rate_limit_event documenta cuando un cliente excede límites de rate limiting. La función audit_token_event registra emisión, verificación y revocación de tokens. Todas estas funciones utilizan un logger dedicado security.audit que puede configurarse independientemente para persistencia especializada.

El módulo proporciona loggers preconfigurados para cada dominio funcional del sistema. Los loggers rest_api_logger y ws_gateway_logger sirven como loggers de nivel superior para cada servicio. Los loggers billing_logger, kitchen_logger y diner_logger proporcionan contexto específico de dominio. El security_audit_logger especializado mantiene el trail

de eventos de seguridad separado del logging operacional normal.

Capítulo 3: Constantes de Dominio y Validación de Estados

El archivo constants.py con aproximadamente cuatrocientas ochenta y seis líneas de código centraliza absolutamente todas las constantes de dominio del sistema, eliminando los denominados magic strings y magic numbers dispersos que dificultaban el mantenimiento y generaban inconsistencias sutiles entre diferentes partes del código.

La clase Roles define los cuatro roles fundamentales del sistema como strings inmutables: ADMIN con acceso completo al sistema, MANAGER para gestión de sucursales, KITCHEN para operaciones de cocina, y WAITER para gestión de mesas y pedidos. Más allá de definir valores individuales, la clase incluye agrupaciones semánticas que simplifican dramáticamente las verificaciones de permisos. El conjunto MANAGEMENT_ROLES agrupa ADMIN y MANAGER permitiendo verificaciones simplificadas en código que requiere cualquier rol de gestión. El conjunto STAFF_ROLES incluye todos los roles que representan personal del restaurante. El conjunto KITCHEN_ACCESS_ROLES agrupa los roles que pueden acceder a funcionalidad de cocina.

Las clases de estado definen no solo los valores posibles para cada entidad sino también agrupaciones semánticas que encapsulan reglas de negocio

complejas. La clase RoundStatus incluye los estados PENDING cuando el comensal crea el pedido y el mozo debe verificarlo, CONFIRMED cuando el mozo verificó en la mesa, SUBMITTED cuando administración envió a cocina, IN_KITCHEN cuando cocina está preparando, READY cuando cocina terminó, SERVED cuando fue entregado al comensal, y CANCELED para pedidos cancelados. El conjunto ACTIVE agrupa todos los estados donde un pedido está en proceso activo excluyendo los estados terminales SERVED y CANCELED. El conjunto KITCHEN_VISIBLE indica qué estados deben aparecer en la pantalla de cocina, específicamente SUBMITTED e IN_KITCHEN ya que cocina no debe ver pedidos hasta que administración los envíe. El conjunto DINER_VISIBLE lista los estados que deben comunicarse a los comensales vía WebSocket.

La clase TableStatus define LIBRE para mesas disponibles, ACTIVE para sesiones en progreso, PAYING cuando se solicitó la cuenta, y OUT_OF_SERVICE para mesas cerradas o reservadas. La clase SessionStatus define OPEN, PAYING y CLOSED para el ciclo de vida de sesiones de mesa. La clase CheckStatus define OPEN, REQUESTED, IN_PAYMENT y PAID para el ciclo de facturación. Las clases TicketStatus, TicketItemStatus y ServiceCallStatus implementadas bajo HIGH-07 FIX definen estados para tickets de cocina, items individuales de ticket, y llamadas de servicio respectivamente.

El diccionario ROUND_TRANSITIONS codifica la máquina de estados completa de pedidos, definiendo transiciones válidas como pares de estado origen a conjunto de estados destino. Desde PENDING las transiciones válidas son a CONFIRMED o CANCELED ya

que el mozo debe verificar o cancelar. Desde CONFIRMED se puede transicionar a SUBMITTED o CANCELED ya que solo administración envía a cocina. Desde SUBMITTED a IN_KITCHEN o CANCELED. Esta codificación explícita previene transiciones inválidas que podrían corromper el estado del sistema.

El diccionario ROUND_TRANSITION_ROLES agrega restricciones de rol a cada transición implementando control de acceso basado en el flujo de trabajo del restaurante. La transición de PENDING a CONFIRMED solo puede ser ejecutada por WAITER, ADMIN o MANAGER reflejando que un mesero debe verificar físicamente el pedido en la mesa. La transición de CONFIRMED a SUBMITTED requiere ADMIN o MANAGER ya que solo ellos pueden enviar pedidos a cocina. La transición de IN_KITCHEN a READY solo puede ser ejecutada por KITCHEN ya que solo el personal de cocina sabe cuándo un plato está terminado.

Las funciones de validación proporcionan una API declarativa para verificar estados y transiciones implementada bajo HIGH-07 FIX. La función validate_round_status verifica si un string representa un estado válido de RoundStatus. La función validate_ticket_status hace lo mismo para tickets. La función validate_round_transition verifica si la transición de un estado a otro es válida según ROUND_TRANSITIONS. La función get_allowed_round_transitions retorna el conjunto de estados destino permitidos desde un estado origen para un conjunto de roles dado, combinando la máquina de estados con las restricciones de rol.

La clase ErrorMessages centraliza mensajes de error en español garantizando consistencia lingüística en toda la API. Mensajes como Categoría no encontrada, Permisos insuficientes, Transición de estado inválida, y Token expirado se definen una sola vez y se referencian consistentemente en todo el código, facilitando localización futura y mantenimiento del tono de comunicación con usuarios.

La clase Limits centraliza los límites de validación numéricos. Los límites de cantidad van de uno a noventa y nueve para items de carrito. Los precios van de cero a cien mil en centavos. Las longitudes de string tienen doscientos caracteres para nombres, dos mil para descripciones, y dos mil cuarenta y ocho para URLs. La paginación tiene límite por defecto de cincuenta y máximo de doscientos. Los comensales por mesa van de uno a veinte.

Capítulo 4: Autenticación JWT y Tokens de Mesa

El archivo auth.py con aproximadamente quinientas sesenta y una líneas de código implementa un sistema de autenticación dual que soporta tanto personal del restaurante mediante tokens JWT estándar como comensales en mesa mediante tokens específicos de sesión. Esta dualidad refleja los dos flujos de autenticación fundamentales del sistema: empleados que inician sesión con credenciales permanentes, y clientes efímeros que escanean códigos QR para una experiencia sin fricción.

La función sign_jwt genera tokens firmados utilizando el algoritmo HS256, incluyendo tanto claims estándar definidos por la especificación JWT como claims personalizados del dominio de negocio. Los claims estándar incluyen iss como issuer configurado como integrador, aud como audience integrador:staff, iat como timestamp de creación, y exp como expiración calculada sumando el tiempo de vida configurado al momento de emisión. Los claims de dominio incluyen sub con el identificador de usuario como string, tenant_id con el identificador del tenant para multi-tenancy, branch_ids con la lista de sucursales accesibles, roles con la lista de roles del usuario, email para auditoría, y type indicando si es token access o refresh.

El artefacto CRIT-AUTH-04 FIX documenta la adición del claim jti como JWT ID a cada token, un identificador único generado mediante UUID versión cuatro que habilita la revocación individual de tokens. Antes de esta implementación revocar un token significaba invalidar toda la sesión del usuario; ahora es posible revocar tokens específicos mientras otros tokens del mismo usuario permanecen válidos, habilitando escenarios como cierre de sesión selectivo desde dispositivos específicos.

La función verify_jwt implementa múltiples capas de validación que deben superarse secuencialmente. Primero la librería PyJWT valida la firma criptográfica del token verificando que no ha sido alterado desde su emisión. Luego valida la expiración rechazando tokens cuyo timestamp exp es anterior al momento actual. Después verifica issuer y audience rechazando tokens emitidos por otros sistemas o

dirigidos a otras audiencias. Las validaciones adicionales documentadas como SHARED-HIGH-03 FIX verifican presencia y formato de claims requeridos: sub debe ser un string que represente un entero válido, tenant_id debe ser un entero positivo, type debe ser exactamente access o refresh, roles debe ser una lista no vacía, y branch_ids debe ser una lista de enteros. Finalmente si todas las validaciones estructurales pasan la función consulta la blacklist de Redis para verificar que el token no ha sido explícitamente revocado.

El patrón fail-closed documentado como SHARED-HIGH-01 FIX gobierna el manejo de errores durante la verificación. Si Redis no está disponible para verificar la blacklist ya sea por fallo de red, timeout, o cualquier otra razón, el sistema deniega acceso en lugar de asumirlo. Esta decisión prioriza explícitamente seguridad sobre disponibilidad, aceptando que algunos usuarios legítimos podrían ser temporalmente rechazados durante una falla de Redis a cambio de garantizar que tokens potencialmente revocados nunca obtengan acceso.

Los tokens de mesa evolucionaron significativamente durante el desarrollo del proyecto. Originalmente utilizaban un formato HMAC propietario, pero en la Fase cinco migraron a JWT estándar por consistencia y para aprovechar toda la infraestructura de verificación existente. La función sign_table_token genera tokens JWT con issuer integrador:table y audience integrador:diner que los distinguen claramente de tokens de staff. Los claims incluyen table_id, session_id, branch_id, y tenant_id necesarios para autorizar operaciones en contexto de

mesa. La función verify_table_token mantiene retrocompatibilidad detectando el formato del token: si contiene tres partes separadas por puntos es JWT y se procesa como tal, de lo contrario se intenta verificación HMAC legacy permitiendo migración sin downtime.

El dependency current_user_context proporciona integración directa con FastAPI para extraer y validar el usuario actual desde el header Authorization. Este dependency extrae el token JWT, lo verifica completamente, y retorna el payload decodificado como diccionario que contiene sub con el user_id como string, tenant_id como entero, branch_ids como lista de enteros, roles como lista de strings, y email. Las funciones auxiliares require_roles y require_branch validan que el usuario tenga los roles necesarios o acceso a la sucursal requerida, lanzando HTTPException 403 en caso contrario.

Capítulo 5: Hashing Seguro de Contraseñas

El módulo password.py con aproximadamente ochenta y dos líneas de código concentrado en funcionalidad crítica implementa hashing seguro mediante bcrypt directo. La decisión de usar bcrypt directamente en lugar de la librería passlib que era la opción original responde a problemas de compatibilidad descubiertos durante la migración a Python 3.14 donde passlib generaba warnings de deprecación y en algunos casos errores de runtime.

La función hash_password genera hashes utilizando doce rondas de bcrypt, un balance cuidadosamente calculado entre seguridad y rendimiento. Con doce rondas el hashing de una contraseña requiere aproximadamente doscientos cincuenta milisegundos en hardware típico, suficiente para hacer inviables ataques de fuerza bruta pero no tanto como para degradar perceptiblemente la experiencia de login del usuario. El incremento de una ronda duplica aproximadamente el tiempo de cómputo, por lo que la diferencia entre diez y doce rondas es sustancial sin ser excesiva. El resultado es un string de sesenta caracteres comenzando con el prefijo $2b$ que identifica el algoritmo y la versión.

El artefacto CRIT-AUTH-03 FIX documenta la eliminación del soporte para contraseñas en texto plano que existía para compatibilidad con datos legacy de versiones anteriores del sistema. La función verify_password ahora rechaza explícitamente cualquier hash que no comience con los prefijos bcrypt estándar $2a$, $2b$ o $2y$, generando un log de seguridad nivel WARNING si detecta intentos de autenticación con formatos no soportados. Este log permite identificar cuentas que aún tendrían contraseñas en formato legacy, situación que debería ser imposible en producción pero que podría ocurrir en ambientes de desarrollo o testing con datos antiguos. La verificación utiliza comparación en tiempo constante para prevenir ataques de timing que podrían revelar información sobre la contraseña.

Capítulo 6: Revocación de Tokens en Tiempo Real

El servicio token_blacklist.py con aproximadamente trescientas treinta y cuatro líneas de código proporciona dos mecanismos complementarios de revocación que cubren diferentes escenarios de uso, utilizando Redis como almacén de estado distribuido que permite que cualquier instancia del servidor reconozca tokens revocados instantáneamente.

La revocación individual mediante la función blacklist_token permite invalidar un token específico identificado por su jti, útil para cierre de sesión desde un dispositivo particular o revocación de emergencia de un token comprometido. La función recibe el payload decodificado del token a revocar y calcula el TTL apropiado antes de almacenar en Redis. Si el token expira en treinta minutos no tiene sentido almacenar su jti por siete días; en cambio el TTL de la entrada Redis se configura para coincidir con el tiempo restante de validez del token más un pequeño margen de seguridad. Una vez que el token habría expirado naturalmente la entrada Redis se elimina automáticamente, previniendo crecimiento indefinido de la blacklist. La key utiliza el prefijo auth:blacklist: seguido del jti.

La revocación por usuario mediante revoke_all_user_tokens se utiliza durante logout y cambio de contraseña para invalidar cualquier token existente del usuario. En lugar de enumerar y blacklistear cada token individualmente lo cual sería costoso y propenso a race conditions, esta función almacena el timestamp actual asociado al user_id bajo la key auth:user:revoke: seguida del user_id. Cualquier token con iat anterior a este timestamp se

considera inválido sin importar su jti individual. El TTL de esta entrada es igual a la vida máxima de tokens de refresco de siete días, garantizando que cualquier token existente haya expirado naturalmente antes de que la marca de revocación desaparezca.

La función is_token_blacklisted implementa el patrón fail-closed con particular rigor utilizando el cliente Redis síncrono para verificación ya que esta función se invoca desde el middleware de autenticación que opera síncronamente. Si la conexión Redis falla, si ocurre un timeout, o si cualquier excepción se genera durante la verificación, la función retorna True indicando que el token está blacklisteado. Esta decisión significa que una falla de Redis causa denial of service temporal en lugar de bypass de seguridad.

La función optimizada check_token_validity documentada como PERF-CRIT-01 FIX utiliza Redis PIPELINE para verificar tanto blacklist individual como revocación por usuario en un solo round-trip de red. Sin esta optimización cada verificación de token requería dos llamadas Redis secuenciales; con el pipeline ambas verificaciones se envían juntas y las respuestas se reciben juntas, reduciendo la latencia de verificación aproximadamente a la mitad lo cual es crítico dado que esta verificación ocurre en cada request autenticado.

Las variantes síncronas de todas estas funciones identificadas por el sufijo _sync utilizan el pool de conexiones síncronas de Redis separado. Esta separación es necesaria porque el middleware de autenticación de FastAPI donde se invoca la

verificación de tokens opera en contexto síncrono mientras que el resto de la aplicación es predominantemente asíncrono. Mezclar operaciones síncronas y asíncronas en el mismo pool Redis generaba deadlocks y errores de event loop documentados en CRIT-LOCK-02.

Capítulo 7: Protección contra Abuso mediante Rate Limiting

El módulo rate_limit.py con aproximadamente trescientas sesenta y seis líneas de código implementa protección multicapa contra abuso combinando la librería slowapi para endpoints REST estándar con lógica personalizada basada en Redis para casos especiales que requieren mayor control, particularmente el endpoint de login que es objetivo frecuente de ataques de fuerza bruta.

Slowapi proporciona rate limiting por dirección IP para la mayoría de endpoints utilizando sliding window algorithm para distribución uniforme de requests permitidos. La configuración permite especificar límites como diez requests por minuto o cien requests por hora, con headers estándar que informan al cliente cuántos requests le quedan y cuándo se resetea su ventana.

El rate limiting por email utiliza lógica Redis personalizada para el endpoint de login donde limitar por IP no es suficiente ya que atacantes pueden rotar IPs pero usualmente atacan un email específico. El límite configurado mediante LOGIN_RATE_LIMIT permite

cinco intentos por defecto, con una ventana de tiempo de sesenta segundos configurada en LOGIN_RATE_WINDOW. Estos valores son configurables mediante settings permitiendo ajuste según las necesidades operativas.

El script Lua que implementa esta lógica documentado como REDIS-HIGH-06 FIX garantiza atomicidad de las operaciones INCR y EXPIRE que de otra manera sufrirían race conditions. El script primero ejecuta INCR sobre la key de rate limit incrementando el contador atómicamente y retornando el nuevo valor. Si el valor retornado es uno significa que la key no existía previamente y acaba de ser creada, por lo que una llamada EXPIRE establece el tiempo de vida de la ventana. Si el valor es mayor que uno la key ya existía con su TTL configurado y no requiere modificación. Este patrón garantiza que nunca exista una key sin TTL situación que causaría bloqueos permanentes del email afectado.

El patrón fail-closed documentado como REDIS-CRIT-01 FIX gobierna el comportamiento cuando Redis no está disponible. Si la verificación de rate limit falla por cualquier razón ya sea error de conexión, timeout, o excepción inesperada, el sistema rechaza la request con HTTP 503 Service Unavailable en lugar de permitirla. Esta decisión aunque impacta disponibilidad durante fallas de Redis previene ataques de fuerza bruta que podrían aprovechar momentos de inestabilidad del sistema de rate limiting.

La función check_email_rate_limit_sync implementada bajo CRIT-LOCK-02 FIX utiliza el cliente Redis síncrono directamente. La implementación anterior

utilizaba asyncio.run para ejecutar código asíncrono desde contexto síncrono pero esto generaba conflictos cuando el event loop ya estaba corriendo, situación común durante request handling en FastAPI. La nueva implementación usa ThreadPoolExecutor para ejecutar la verificación en un thread separado evitando completamente conflictos de event loop. El executor utiliza un máximo de dos worker threads y tiene un timeout de cinco segundos para prevenir bloqueos indefinidos.

Capítulo 8: Gestión de Base de Datos

El módulo db.py con aproximadamente noventa y seis líneas de código altamente optimizado configura SQLAlchemy para operación eficiente en ambientes de producción con alta concurrencia. El engine se crea con parámetros cuidadosamente seleccionados que balancean rendimiento, resiliencia y uso de recursos.

El parámetro pool_pre_ping activo instruye a SQLAlchemy a verificar cada conexión antes de usarla ejecutando un simple SELECT 1 para confirmar que la conexión sigue viva. Esta verificación detecta conexiones muertas ya sea por timeout del servidor PostgreSQL, por reinicio de la base de datos, o por problemas de red, antes de que causen errores durante queries reales. El overhead es mínimo comparado con el costo de manejar errores de conexión muerta a mitad de transacción.

El tamaño del pool se calcula dinámicamente mediante la función _calculate_pool_size implementada bajo

DEFECTO-05 FIX. La fórmula utiliza dos veces el número de cores de CPU más uno, con un máximo de veinte conexiones. Esta adaptación automática permite que el sistema funcione correctamente tanto en máquinas de desarrollo pequeñas como en servidores de producción potentes. El parámetro max_overflow de quince permite crear hasta treinta y cinco conexiones totales bajo carga, las veinte permanentes más quince adicionales que se cierran después de un período de inactividad.

El parámetro pool_timeout de treinta segundos define cuánto tiempo una request esperará por una conexión disponible antes de fallar con timeout. El parámetro pool_recycle de mil ochocientos segundos equivalente a treinta minutos instruye al pool a cerrar y recrear conexiones que han estado abiertas por más de ese tiempo, previniendo problemas con servidores PostgreSQL configurados para cerrar conexiones idle.

La función get_db proporciona un generador compatible con FastAPI Depends garantizando que cada request HTTP reciba una sesión de base de datos limpia y que ésta se cierre correctamente incluso si la request termina con excepción. El patrón try/finally asegura que session.close siempre se ejecute. La función get_db_context ofrece la misma funcionalidad como context manager para código fuera de endpoints FastAPI permitiendo uso con la sintaxis with.

La función safe_commit documentada como HIGH-01 FIX encapsula el patrón de commit con rollback automático ante fallos. Sin esta abstracción cada lugar que hace commit debería manejar excepciones y ejecutar rollback manualmente, un patrón repetitivo propenso a

olvidos que podrían dejar transacciones en estados inconsistentes. Con safe_commit el código simplemente llama a la función y confía en que cualquier fallo dejará la sesión en estado limpio con rollback automático.

Capítulo 9: El Sistema de Eventos y Mensajería

El paquete events/ representa la evolución arquitectónica más significativa de la capa compartida, habiendo sido refactorizado desde un archivo monolítico de más de mil líneas a una estructura modular de once archivos especializados que suman aproximadamente mil ochocientas veinte líneas. Esta refactorización mejora dramáticamente la mantenibilidad, testabilidad y comprensibilidad del sistema de eventos en tiempo real.

El módulo redis_pool.py gestiona dos pools de conexiones Redis separados que sirven propósitos distintos y nunca deben mezclarse. El pool asíncrono obtenido mediante get_redis_pool soporta las operaciones de publicación/suscripción y eventos en tiempo real donde la naturaleza no bloqueante es absolutamente esencial. El pool síncrono obtenido mediante get_redis_sync_client atiende operaciones inherentemente bloqueantes como verificación de blacklist y rate limiting que deben completarse antes de continuar la ejecución.

El pool asíncrono utiliza el patrón singleton con double-check locking para garantizar thread safety durante inicialización. Una variable global almacena

la instancia del pool inicialmente como None. La función get_redis_pool primero verifica si el pool ya existe sin adquirir lock retornándolo inmediatamente si es así. Si no existe adquiere un lock asíncrono y vuelve a verificar ya que otro coroutine podría haber creado el pool mientras se esperaba el lock. Solo si después de adquirir el lock el pool sigue sin existir se procede a crearlo con cincuenta conexiones máximas, timeout de socket de cinco segundos, y health check interval de treinta segundos.

El dataclass Event en event_schema.py define la estructura unificada para absolutamente todos los eventos del sistema garantizando consistencia y facilitando el procesamiento downstream. Cada evento contiene un type que lo identifica como ROUND_SUBMITTED o TABLE_CLEARED. El campo tenant_id soporta multi-tenancy asegurando que eventos de un restaurante nunca se mezclen con eventos de otro. El campo branch_id identifica la sucursal específica donde ocurrió el evento. Los campos opcionales table_id, session_id y sector_id proporcionan contexto adicional cuando es aplicable. El campo entity contiene los datos específicos del evento como diccionario JSON. El método **__post_init__** documentado como SHARED-HIGH-06 FIX valida todos los campos inmediatamente después de la creación de la instancia rechazando eventos malformados antes de que lleguen a Redis.

El archivo event_types.py centraliza las constantes de tipo de evento utilizadas en todo el sistema. Los eventos de ciclo de vida de pedidos incluyen ROUND_PENDING, ROUND_CONFIRMED, ROUND_SUBMITTED, ROUND_IN_KITCHEN, ROUND_READY, ROUND_SERVED,

ROUND_CANCELED y ROUND_ITEM_DELETED. Los eventos de carrito compartido introducidos para sincronización en tiempo real entre dispositivos incluyen CART_ITEM_ADDED, CART_ITEM_UPDATED, CART_ITEM_REMOVED, CART_CLEARED y CART_SYNC. Los eventos de facturación incluyen CHECK_REQUESTED, CHECK_PAID, PAYMENT_APPROVED, PAYMENT_REJECTED y PAYMENT_FAILED. Los eventos de administración para sincronización del Dashboard incluyen ENTITY_CREATED, ENTITY_UPDATED, ENTITY_DELETED y CASCADE_DELETE.

El módulo channels.py centraliza la construcción de nombres de canales Redis eliminando la posibilidad de typos o inconsistencias que podrían causar que eventos no lleguen a sus destinatarios. La función channel_branch_waiters recibe un branch_id y retorna el string branch:{id}:waiters para notificaciones a meseros de esa sucursal. La función channel_branch_kitchen construye branch:{id}:kitchen para personal de cocina. La función channel_sector_waiters recibe un sector_id y construye sector:{id}:waiters para notificaciones filtradas por sector. La función channel_session construye session:{id} para notificaciones a todos los comensales de una sesión de mesa particular. Cada función valida que el ID recibido sea un entero positivo antes de construir el nombre del canal bajo REDIS-MED-07 FIX.

Capítulo 10: Publicación de Eventos con Resiliencia

El módulo publisher.py implementa la publicación de eventos con múltiples capas de resiliencia que garantizan entrega confiable incluso en condiciones adversas. Antes de intentar publicar la función publish_event verifica el tamaño del evento serializado contra un límite máximo de sesenta y cuatro kilobytes bajo REDIS-HIGH-07 FIX, rechazando eventos anormalmente grandes que podrían indicar bugs o intentos de abuso.

Luego consulta el circuit breaker para verificar si Redis está disponible. Si el breaker está abierto la función retorna inmediatamente sin intentar la publicación evitando esperas inútiles que degradarían el rendimiento. Si el breaker permite la operación se procede con el intento de publicación utilizando retry configurable con tres intentos por defecto bajo REDIS-HIGH-03 FIX. Si el primer intento falla se espera un tiempo calculado mediante backoff exponencial con jitter bajo REDIS-MED-01 FIX antes de reintentar. Cada intento subsecuente espera aproximadamente el doble del anterior más o menos variabilidad del jitter que previene thundering herd cuando múltiples instancias reintentan simultáneamente. Después de cada intento fallido se registra el fallo en el circuit breaker; después de un intento exitoso se registra el éxito.

La función publish_to_stream utiliza Redis Streams mediante el comando XADD en lugar de pub/sub tradicional para eventos clasificados como críticos. La diferencia fundamental es que Streams persisten los eventos y permiten que consumidores que estaban offline hagan catch-up cuando reconectan. En contraste pub/sub es fire-and-forget: si nadie está

escuchando cuando se publica un evento ese evento se pierde permanentemente. Para eventos críticos como cambios de estado de pedidos y pagos la persistencia de Streams garantiza que ningún evento se pierda aunque el Gateway se reinicie.

El módulo circuit_breaker.py implementa el patrón Circuit Breaker para proteger al sistema de cascading failures cuando Redis experimenta problemas de disponibilidad. El breaker opera en tres estados claramente definidos. En estado CLOSED que es el estado normal de operación todas las llamadas a Redis proceden normalmente. Cada fallo incrementa un contador interno y cuando este contador alcanza el umbral configurado de cinco fallos consecutivos el breaker transiciona a estado OPEN. En estado OPEN el breaker rechaza inmediatamente todas las llamadas a Redis sin siquiera intentar la conexión retornando un error específico. El breaker permanece en OPEN por treinta segundos durante el cual el sistema opera en modo degradado pero estable. Transcurrido el tiempo el breaker transiciona a estado HALF_OPEN donde permite un número limitado de llamadas de prueba. Si estas pruebas tienen éxito el breaker transiciona de vuelta a CLOSED; si alguna falla vuelve a OPEN por otro período de espera.

El módulo domain_publishers.py proporciona funciones de alto nivel que encapsulan la lógica de routing de eventos determinando automáticamente a qué canales enviar según el tipo de evento y el contexto. La función publish_round_event determina los canales destino según el tipo de evento: ROUND_PENDING y ROUND_CONFIRMED se envían a waiters y admin ya que cocina no debe ver pedidos hasta que management los

envíe, ROUND_SUBMITTED se envía adicionalmente a kitchen, y ROUND_IN_KITCHEN, ROUND_READY y ROUND_SERVED se envían también al canal de session para que los comensales vean el progreso de su pedido.

Capítulo 11: Excepciones HTTP Centralizadas

El módulo exceptions.py con aproximadamente trescientas diecisiete líneas de código define una jerarquía de excepciones HTTP que combina logging automático con respuestas estandarizadas. Esta centralización garantiza que errores similares produzcan respuestas idénticas en toda la API facilitando el debugging y mejorando la experiencia del desarrollador que consume la API.

La clase base AppException extiende HTTPException de FastAPI agregando logging estructurado al momento de instanciación. Cada excepción derivada registra automáticamente su creación con nivel apropiado: warning para errores de cliente como 404 o 403, y error para problemas de servidor como 500. El log incluye contexto relevante como entity_type, entity_id, user_id, y cualquier otro keyword argument pasado al constructor facilitando correlación y diagnóstico.

La clase NotFoundError proporciona un constructor semántico para errores 404. Una llamada como NotFoundError con argumentos Producto y product_id produce una respuesta HTTP 404 con body conteniendo detail igual a Producto con ID 123 no encontrado y un

log estructurado que incluye entity type, ID, y tenant para facilitar debugging. Las clases derivadas SessionNotFoundError, CheckNotFoundError y RoundNotFoundError proporcionan mensajes específicos para cada tipo de entidad.

La clase ForbiddenError maneja errores 403 de autorización con mensaje que indica qué acción estaba prohibida. Las clases derivadas BranchAccessError indica No autorizado para acceder a esta sucursal, e InsufficientRoleError lista los roles requeridos para la operación.

La clase ValidationError y sus derivadas manejan errores 400 de validación de negocio. InvalidStateError indica que una entidad está en un estado que no permite la operación solicitada incluyendo el estado actual y los estados esperados. InvalidTransitionError indica que la transición de estado solicitada no es válida incluyendo estados origen y destino. DuplicateEntityError indica que ya existe una entidad con el identificador proporcionado. PaymentAmountError valida montos de pago.

La clase ExternalServiceError distingue entre servicios externos no disponibles que retornan 503 Service Unavailable, y errores de comunicación con gateways externos que retornan 502 Bad Gateway. Cuando es aplicable la respuesta incluye el header Retry-After indicando cuándo el cliente debería reintentar. La clase RateLimitError para errores 429 incluye automáticamente el header Retry-After con el tiempo de espera en segundos.

Capítulo 12: Validadores de Seguridad

El módulo validators.py con aproximadamente doscientas catorce líneas de código implementa validaciones críticas de seguridad particularmente para prevención de ataques XSS y SSRF que podrían ocurrir a través de URLs proporcionadas por usuarios.

La función validate_image_url implementa múltiples capas de defensa contra URLs maliciosas identificada como CRIT-02 FIX. La primera capa valida el scheme rechazando javascript: que podría ejecutar código, data: que podría contener payloads maliciosos, file: que podría acceder al filesystem del servidor, y cualquier otro scheme que no sea http o https. La segunda capa previene SSRF verificando que el host destino no sea interno. La lista de hosts bloqueados incluye localhost y 127.0.0.1 para la máquina local, los rangos de IP privadas 10.0.0.0/8, 172.16.0.0/12 y 192.168.0.0/16, la IP de link-local 169.254.169.254 que es el endpoint de metadata en clouds como AWS y GCP, y strings como metadata.google para acceso a metadata de instancia en Google Cloud. Esta validación previene que un atacante use el servidor como proxy para acceder a servicios internos. La tercera capa valida extensiones de imagen permitiendo solo jpg, png, gif, webp y svg. La cuarta capa limita la longitud de la URL a dos mil cuarenta y ocho caracteres previniendo ataques de buffer overflow.

La función escape_like_pattern documentada como HIGH-01 FIX escapa los caracteres especiales porcentaje y guión bajo que tienen significado especial en

cláusulas SQL LIKE. Sin este escape un atacante podría proporcionar un término de búsqueda como porcentaje que matchearía todos los registros potencialmente causando un full table scan costoso y exfiltrando datos. El porcentaje se escapa como backslash porcentaje y el guión bajo como backslash guión bajo.

La función validate_quantity proporciona validación de rango para cantidades de items de carrito verificando que el valor esté dentro del rango configurado de uno a noventa y nueve por defecto. La función sanitize_search_term normaliza términos de búsqueda removiendo caracteres de control como null bytes, espacios excesivos, y limitando la longitud máxima a cien caracteres para prevenir queries sobredimensionados.

Capítulo 13: Health Checks con Timeout

El módulo health.py documentado como ARCH-OPP-03 FIX con aproximadamente doscientas sesenta y ocho líneas de código proporciona decoradores y utilidades para implementar health checks robustos con comportamiento consistente y protección contra timeouts.

El decorador health_check_with_timeout envuelve una función de health check asíncrona con protección de timeout y manejo estructurado de resultados. El decorador recibe el timeout en segundos y el nombre del componente como parámetros. La función decorada se ejecuta con asyncio.wait_for que cancela la operación si excede el timeout. La latencia se mide

mediante time.perf_counter para precisión de nanosegundos. Si la función completa exitosamente se retorna un HealthCheckResult con status HEALTHY incluyendo la latencia medida y cualquier detail retornado por la función como tamaño de pool o conteo de conexiones. Si ocurre timeout se retorna status UNHEALTHY con error indicando timeout after N seconds. Si ocurre cualquier otra excepción se retorna status UNHEALTHY con el mensaje de error capturado.

La clase HealthCheckResult es un dataclass frozen que encapsula el resultado de un health check. Incluye status como enum HealthStatus con valores HEALTHY, DEGRADED o UNHEALTHY. Incluye component como string identificando qué componente se verificó. Incluye latency_ms como float indicando cuánto tardó la verificación en milisegundos. Incluye error opcional como string si hubo fallo. Incluye details opcional como diccionario con información adicional específica del componente.

La función aggregate_health_checks recibe una lista de coroutines de health check y las ejecuta en paralelo mediante asyncio.gather. Los resultados se agregan en un diccionario donde la key es el nombre del componente y el value es su HealthCheckResult. El status global se calcula como HEALTHY si todos los componentes están healthy, DEGRADED si alguno está unhealthy pero la mayoría están healthy, y UNHEALTHY si la mayoría están unhealthy o si componentes críticos específicos como base de datos o Redis fallaron.

Capítulo 14: Schemas Pydantic Compartidos

El archivo schemas.py con aproximadamente novecientas cuarenta y ocho líneas de código contiene los schemas Pydantic utilizados por endpoints públicos y operacionales de la API. Estos schemas definen el contrato de comunicación entre frontend y backend garantizando que ambos lados acuerden en la estructura de datos intercambiados con validación automática.

Los schemas de autenticación incluyen LoginRequest con campos email y password ambos requeridos como strings, LoginResponse con access_token, refresh_token, token_type siempre como Bearer, expires_in indicando segundos hasta expiración, y UserInfo anidado con datos del usuario. RefreshTokenRequest contiene solo el token de refresco a renovar. TokenPayload representa el contenido decodificado de un JWT con todos los claims tipados.

Los schemas de mesa y sesión incluyen TableCard con información resumida de una mesa para listados incluyendo id, código, capacidad, estado y datos de sesión activa si existe. TableSessionResponse contiene los datos retornados al crear o obtener una sesión incluyendo session_id, table_token para autenticación de comensal, y datos de la mesa. TableSessionDetail proporciona toda la información de una sesión activa incluyendo lista de diners, rounds con sus items, y service calls pendientes.

Los schemas de pedidos incluyen SubmitRoundRequest con la estructura de un pedido a enviar conteniendo lista de CartItem donde cada uno tiene product_id, quantity entre uno y noventa y nueve, y notes opcional de hasta quinientos caracteres. RoundOutput contiene los datos de un round creado incluyendo id, status, items con sus detalles, y timestamps de creación y actualización.

Los schemas de facturación incluyen RequestCheckResponse retornado cuando se solicita la cuenta conteniendo check_id y total en centavos. CashPaymentRequest contiene amount_cents y método de pago. PaymentResponse incluye payment_id, status, y detalles del cambio si corresponde. CheckDetailOutput proporciona el detalle completo de una cuenta incluyendo charges por diner, payments recibidos, y allocations mostrando cómo se asignó cada pago.

El archivo admin_schemas.py con aproximadamente ochocientas diez líneas adicionales contiene schemas específicos para el Dashboard administrativo. Siguiendo Clean Architecture estos schemas residen en la capa compartida para que los Domain Services puedan usarlos sin depender de la capa de routers. Los schemas siguen un patrón consistente donde EntityOutput contiene todos los campos para lectura con Config from_attributes=True para conversión automática desde modelos SQLAlchemy, EntityCreate contiene campos requeridos para creación con validadores para campos especiales como URLs de imagen, y EntityUpdate contiene campos opcionales para actualización parcial.

Capítulo 15: Correlación de Requests y Telemetría

El módulo correlation.py con aproximadamente cien líneas de código proporciona tracking de requests distribuidos mediante context variables permitiendo que todos los logs generados durante el procesamiento de una request compartan el mismo identificador único facilitando correlación en sistemas de log management.

El sistema utiliza ContextVar de Python para almacenar el request_id de manera thread-safe y compatible con código asíncrono. El CorrelationIdMiddleware instalado en la aplicación FastAPI genera o extrae el identificador: si el cliente envía un header X-Request-ID se utiliza ese valor permitiendo trazabilidad end-to-end; si no se genera un UUID nuevo. El identificador se almacena en la context variable y se incluye en el header de respuesta permitiendo al cliente correlacionar sus logs con los del servidor.

El CorrelationIdFilter se instala en todos los handlers de logging agregando automáticamente el campo request_id a cada log record. Cuando se utiliza StructuredFormatter este campo aparece como campo JSON independiente; con DevelopmentFormatter aparece como prefijo entre corchetes. La función setup_correlation_logging instala el filtro en todos los handlers existentes durante el startup de la aplicación.

El módulo telemetry.py con aproximadamente ciento treinta y nueve líneas proporciona hooks de

observabilidad adicionales para integración con sistemas de distributed tracing cuando es necesario instrumentación más detallada que simple correlación por request ID.

Capítulo 16: Thread Safety y Patrones de Concurrencia

El sistema implementa múltiples instancias del patrón double-check locking para inicialización lazy de recursos compartidos identificado en múltiples artefactos CRIT-LOCK. Este patrón evita tanto la creación múltiple de recursos como la contención innecesaria de locks en el camino común donde el recurso ya existe.

Para el pool Redis asíncrono una variable global _redis_pool comienza como None. La función get_redis_pool primero verifica si el pool existe sin adquirir lock retornando inmediatamente si ya está inicializado evitando cualquier overhead de sincronización en el caso común. Si no existe adquiere un asyncio.Lock y vuelve a verificar dentro del lock ya que otro coroutine podría haber inicializado el pool mientras se esperaba. Solo si después de adquirir el lock el pool sigue siendo None se procede a crearlo. Este segundo check dentro del lock es lo que hace double-check al patrón y previene creación duplicada que desperdiciaría recursos y podría causar inconsistencias.

Para recursos síncronos como el pool de conexiones Redis síncrono y el ThreadPoolExecutor de rate

limiting el patrón utiliza threading.Lock en lugar de asyncio.Lock ya que opera en contexto síncrono. El artefacto CRIT-LOCK-02 FIX documenta específicamente que usar asyncio.Lock desde código síncrono causa errores de event loop porque asyncio.Lock requiere un event loop activo que no existe cuando se ejecuta código síncrono puro.

El sistema utiliza un patrón de cleanup en dos fases para estructuras de datos compartidas evitando el error dictionary changed size during iteration. La primera fase toma un snapshot de los items bajo el lock. La segunda fase itera sobre el snapshot identificando entries a eliminar sin modificar la estructura original. La tercera fase adquiere nuevamente el lock y aplica las eliminaciones verificando que cada key todavía existe antes de eliminar ya que otra operación podría haberla eliminado mientras no se sostenía el lock.

Conclusión

La capa shared/ representa mucho más que una colección de utilidades compartidas entre componentes. Es el verdadero corazón arquitectónico del sistema Integrador donde convergen las decisiones fundamentales de seguridad, configuración, comunicación y resiliencia que definen el comportamiento del sistema completo.

La evolución de esta capa documentada meticulosamente mediante más de novecientos artefactos de auditoría con prefijos CRIT, HIGH, MED y LOW refleja un proceso

continuo de hardening y refinamiento. Desde la migración de tokens de mesa a formato JWT estándar pasando por la implementación de circuit breakers para resiliencia Redis hasta la refactorización del sistema de eventos en módulos especializados, cada cambio respondió a necesidades reales descubiertas durante operación y testing.

La separación clara de responsabilidades en cuatro módulos principales permite que equipos trabajen independientemente en diferentes aspectos del sistema. Un ingeniero de seguridad puede modificar la lógica de verificación de tokens sin afectar el sistema de eventos. Un especialista en infraestructura puede optimizar pools de conexiones sin tocar la lógica de negocio. Un desarrollador de producto puede agregar nuevos schemas sin preocuparse por autenticación.

Los patrones establecidos garantizan comportamiento predecible incluso en las condiciones más adversas. El patrón fail-closed para seguridad asegura que fallas de infraestructura resulten en denial of service temporal en lugar de bypass de controles. El circuit breaker para resiliencia previene cascadas de fallos cuando Redis experimenta problemas. El double-check locking para concurrencia garantiza inicialización correcta de recursos compartidos sin contención innecesaria. El structured logging para observabilidad facilita diagnóstico y correlación de eventos distribuidos.

Para cualquier desarrollador que se incorpore al proyecto comprender profundamente esta capa es requisito fundamental. Aquí se definen los contratos

que todas las demás capas deben respetar. Aquí se establecen los invariantes de seguridad que nunca deben violarse. Aquí reside en el sentido más literal el fundamento sobre el cual se construye absolutamente todo lo demás del ecosistema Integrador.

REST API: Arquitectura y Funcionamiento

Version 3.0 - Febrero 2026

Introducción

La REST API constituye el núcleo transaccional del ecosistema Integrador, implementando toda la lógica de negocio que sustenta las operaciones de un restaurante moderno. Este servicio, ejecutándose en el puerto 8000, actúa como guardián de los datos y orquestador de las reglas de negocio que gobiernan desde la gestión de catálogos de productos hasta el procesamiento de pagos y la coordinación de pedidos en tiempo real.

La arquitectura de la API fue diseñada siguiendo los principios de Clean Architecture, estableciendo una separación rigurosa entre las distintas capas de responsabilidad. Los routers HTTP actúan como controladores delgados que únicamente manejan preocupaciones de transporte, delegando inmediatamente a servicios de dominio que encapsulan toda la lógica de negocio. Estos servicios, a su vez, utilizan repositorios especializados para el acceso a datos, manteniendo una independencia total respecto a los detalles de persistencia.

El sistema implementa multi-tenancy completo, donde cada restaurante opera en aislamiento total de los demás, compartiendo la misma infraestructura pero con datos completamente segregados. Este aislamiento se garantiza automáticamente a través de repositorios que filtran todas las consultas por tenant_id, eliminando la posibilidad de fugas de información entre organizaciones.

La seguridad permea cada capa de la arquitectura. Desde middlewares que validan orígenes y tipos de contenido, pasando por autenticación JWT con tokens de corta duración, hasta un sistema de control de acceso basado en roles que implementa el patrón Strategy para proporcionar permisos granulares según el rol del usuario. Cada operación se audita completamente, preservando un rastro detallado de quién hizo qué y cuándo.

Capítulo 1: La Estructura Modular del Proyecto

La carpeta rest_api reside dentro del directorio backend y organiza su código en subdirectorios claramente diferenciados por responsabilidad. Esta organización refleja los principios de separación de preocupaciones y facilita tanto la navegación del código como su evolución independiente.

El archivo main.py constituye el punto de entrada de la aplicación FastAPI. En este archivo se configura la secuencia de middlewares de seguridad, donde el orden es crítico ya que CORS debe registrarse último para funcionar correctamente. Se registran dieciocho routers que cubren todas las áreas funcionales del sistema. Se integra OpenTelemetry para observabilidad distribuida y se configura el rate limiting mediante slowapi. El archivo permanece deliberadamente simple, delegando la lógica compleja a módulos especializados.

El directorio core contiene la configuración fundamental de la aplicación. El archivo lifespan.py gestiona el ciclo de vida completo del servicio, orquestando la inicialización de la base de datos, la activación de la extensión pgvector para embeddings vectoriales, la ejecución de seeds iniciales, el arranque de procesadores en background como el procesador Outbox y el reintentador de webhooks, y el cierre ordenado de conexiones durante el shutdown. El archivo middlewares.py implementa los middlewares de seguridad que añaden headers protectores a todas las respuestas y validan los tipos de contenido de las peticiones entrantes, incluyendo Content-Security-Policy, Strict-Transport-Security para producción, y Permissions-Policy para deshabilitar APIs del navegador no utilizadas. El archivo cors.py centraliza la configuración de orígenes permitidos para peticiones cross-origin, distinguiendo entre desarrollo donde se permiten localhost en diversos puertos y producción donde los orígenes provienen de variables de entorno.

El directorio models organiza los modelos de SQLAlchemy en más de veinte archivos modulares, cada uno agrupando entidades relacionadas por dominio. Esta modularización permite que el modelo de datos crezca sin que ningún archivo individual se vuelva inmanejable. El archivo base.py define la clase Base de SQLAlchemy y el AuditMixin que proporciona soft delete y campos de auditoría a todas las entidades. El archivo tenant.py define Tenant representando un restaurante completo y Branch representando una sucursal física. El archivo user.py implementa User y UserBranchRole para la relación muchos a muchos entre usuarios y sucursales con roles específicos. El archivo catalog.py contiene Category, Subcategory, Product y BranchProduct para precios específicos por sucursal. El archivo allergen.py gestiona Allergen, ProductAllergen con tipos de presencia y niveles de riesgo, y AllergenCrossReaction para reacciones cruzadas entre alérgenos. El archivo ingredient.py estructura IngredientGroup, Ingredient, SubIngredient y ProductIngredient en una jerarquía de tres niveles. El archivo product_profile.py implementa doce tablas de relación para perfiles

dietéticos, métodos de cocción, sabores y texturas. El archivo sector.py define BranchSector y WaiterSectorAssignment para asignaciones diarias de mozos. El archivo table.py representa Table y TableSession para el ciclo de vida de una mesa. El archivo customer.py implementa Customer con consentimiento GDPR y Diner para comensales individuales con tracking de dispositivo. El archivo cart.py define CartItem para el carrito compartido en tiempo real. El archivo order.py contiene Round y RoundItem para el ciclo de vida de pedidos. El archivo kitchen.py gestiona KitchenTicket, KitchenTicketItem y ServiceCall. El archivo billing.py implementa Check usando el nombre de tabla app_check para evitar conflictos con palabras reservadas de SQL, junto con Payment, Charge y Allocation para el modelo FIFO de pagos. El archivo promotion.py define Promotion, PromotionBranch y PromotionItem. El archivo recipe.py almacena Recipe y RecipeAllergen para fichas técnicas de cocina. El archivo knowledge.py soporta KnowledgeDocument con embeddings vectoriales y ChatLog para el sistema RAG de chatbot. El archivo audit.py proporciona AuditLog para logging estructurado de operaciones. El archivo outbox.py implementa OutboxEvent para el patrón de eventos transaccionales garantizados.

El directorio services representa el corazón de la lógica de negocio, organizado en subdirectorios especializados. El subdirectorio domain contiene los servicios de dominio principales que implementan las operaciones de negocio para cada entidad, siendo la ubicación preferida para nueva lógica de negocio bajo los principios de Clean Architecture. El subdirectorio crud proporciona utilidades para operaciones de datos como soft delete, auditoría, repositorios tipados y el CRUDFactory que está deprecado en favor de servicios de dominio. El subdirectorio permissions implementa el sistema de control de acceso basado en roles mediante el patrón Strategy. El subdirectorio events gestiona la publicación de eventos de dominio incluyendo el patrón Outbox para eventos críticos. El subdirectorio payments maneja el procesamiento de pagos con patrones de resiliencia como circuit breaker y retry con backoff exponencial.

El directorio routers organiza los endpoints HTTP en grupos funcionales. El subdirectorio auth maneja autenticación sin requerir token previo. El subdirectorio public expone endpoints accesibles sin autenticación. El subdirectorio admin contiene quince routers especializados para operaciones CRUD del dashboard de administración. El subdirectorio diner implementa operaciones para comensales autenticados mediante tokens de mesa. El subdirectorio kitchen proporciona endpoints para personal de cocina. El subdirectorio waiter implementa operaciones para mozos. El subdirectorio billing maneja pagos y facturación. El subdirectorio content gestiona contenido administrable como recetas, ingredientes, promociones y el chatbot RAG. El subdirectorio tables gestiona sesiones de mesa.

Capítulo 2: El Flujo de Datos en Clean Architecture

La arquitectura sigue un flujo unidireccional donde cada capa solo conoce y depende de la capa inmediatamente inferior. Los routers HTTP reciben las peticiones y las transforman en llamadas a servicios de dominio. Los servicios de dominio orquestan la lógica de negocio y utilizan repositorios para acceder a los datos. Los repositorios encapsulan las consultas SQL y retornan entidades del modelo. Las entidades del modelo representan el estado persistido en la base de datos.

Este diseño proporciona múltiples beneficios tangibles. La testabilidad mejora dramáticamente ya que cada capa puede probarse en aislamiento mediante mocks de las capas inferiores. La mantenibilidad se incrementa porque los cambios en una capa no propagan efectos inesperados a las demás. La evolución del sistema se facilita ya que nuevas funcionalidades se añaden en la capa apropiada sin modificar las existentes.

Los routers permanecen deliberadamente delgados, conteniendo únicamente lógica de transformación HTTP. No realizan validaciones de negocio, no acceden directamente a la base de datos, y no contienen condicionales complejos. Su única responsabilidad es recibir una petición, extraer los parámetros relevantes mediante dependencias de FastAPI, invocar al servicio apropiado pasando el contexto de permisos, y formatear la respuesta.

Los servicios de dominio concentran toda la inteligencia del negocio. Validan reglas de negocio complejas como restricciones de unicidad personalizadas. Orquestan operaciones que involucran múltiples entidades manteniendo consistencia transaccional. Publican eventos para notificar cambios a otros sistemas mediante Redis o el patrón Outbox. Manejan los efectos secundarios de las operaciones como actualizar caches o sincronizar datos derivados. Cada servicio se especializa en un dominio específico como productos, categorías, pedidos o facturación.

Los repositorios abstraen completamente los detalles de persistencia. Encapsulan las consultas SQLAlchemy proporcionando métodos semánticos como find_by_branch o find_all_active. Configuran el eager loading de relaciones para prevenir problemas de N+1 queries que degradarían el rendimiento. Aplican filtros de tenant y branch automáticamente garantizando aislamiento multi-tenant. Manejan la paginación mediante offset y limit a nivel de base de datos. El código de los servicios nunca escribe sentencias SQL ni manipula sesiones de base de datos directamente.

Capítulo 3: El Ciclo de Vida de la Aplicación

El archivo lifespan.py implementa el patrón de gestión de ciclo de vida que FastAPI proporciona mediante context managers asíncronos. Esta función se ejecuta una vez al inicio del servidor y otra al cierre, permitiendo inicialización y limpieza coordinadas de recursos.

Durante la fase de startup, el sistema ejecuta una secuencia cuidadosamente ordenada de inicializaciones. Primero valida que los secretos de producción estén correctamente configurados, rechazando arrancar si se detectan valores por defecto inseguros para JWT_SECRET o TABLE_TOKEN_SECRET en entorno de producción. Luego configura el logging estructurado para facilitar el diagnóstico en producción. A continuación habilita la extensión pgvector en PostgreSQL para soportar embeddings de inteligencia artificial necesarios para el sistema RAG. Después invoca Base.metadata.create_all para crear todas las tablas del modelo si no existen. Posteriormente ejecuta los seeds de datos iniciales para poblar configuraciones básicas como roles, categorías predeterminadas y datos de demostración. Luego registra los handlers de webhooks para procesamiento de pagos de Mercado Pago. Arranca los procesadores en background que incluyen el procesador Outbox para publicar eventos garantizados, el reintentador de webhooks fallidos, y el scheduler de refresh-ahead para renovación proactiva de tokens. Finalmente inicializa las métricas Prometheus y opcionalmente precalienta caches de Redis.

Durante la fase de shutdown, el sistema detiene ordenadamente todos los componentes activos. Cancela el scheduler de refresh-ahead esperando que finalice graciosamente. Detiene el procesador Outbox permitiendo que complete los eventos en progreso. Cierra el cliente HTTP de Ollama si estaba activo para el chatbot RAG. Libera el executor del rate limiter. Cierra el pool de conexiones Redis tanto asíncrono como síncrono. Esta secuencia ordenada previene pérdida de datos y garantiza que las operaciones en progreso completen antes del cierre.

Capítulo 4: La Base Auditable de Modelos

Todos los modelos de la API heredan de una clase base AuditMixin que proporciona capacidades uniformes de auditoría y soft delete. Esta herencia garantiza que cada entidad en el sistema mantenga un rastro completo de su historia, cumpliendo con requisitos de trazabilidad y recuperación de datos.

El mixin proporciona un campo booleano is_active que implementa el patrón de soft delete. En lugar de eliminar registros físicamente de la base de datos, las operaciones de eliminación simplemente marcan este campo como falso. Todas las consultas normales filtran automáticamente por is_active verdadero mediante los repositorios, haciendo invisibles los registros eliminados sin perderlos permanentemente. Este campo está indexado para mantener el rendimiento de las consultas filtradas.

Los campos de timestamp registran el momento exacto de cada operación. El campo created_at almacena el instante de creación con zona horaria UTC, usando datetime.now con timezone.utc para evitar el problema de naive datetimes que fue corregido en el fix CRIT-01. El campo updated_at se actualiza automáticamente en cada modificación mediante onupdate de SQLAlchemy. El campo deleted_at registra el momento del soft delete, permitiendo análisis forense de cuándo se eliminaron registros.

Los campos de tracking de usuario denormalizan tanto el identificador como el email del usuario que realizó cada operación. Esta denormalización deliberada permite mostrar información de auditoría sin necesidad de joins adicionales, sacrificando normalización por rendimiento en consultas de auditoría frecuentes. Los campos incluyen created_by_id y created_by_email para creación, updated_by_id y updated_by_email para modificaciones, y deleted_by_id y deleted_by_email para eliminaciones.

El mixin también proporciona métodos de instancia para realizar las operaciones de forma consistente. El método soft_delete recibe el usuario que realiza la operación y actualiza tanto is_active como los campos de auditoría de eliminación. El método restore revierte un soft delete, reactivando la entidad. Los métodos set_created_by y set_updated_by establecen los campos de auditoría correspondientes en creación y actualización.

Capítulo 5: El Sistema de Repositorios

El sistema de repositorios implementa tres niveles de abstracción que proporcionan aislamiento progresivo según las necesidades de cada entidad. Esta jerarquía permite que el código de servicios trabaje con interfaces consistentes mientras el repositorio maneja automáticamente el filtrado apropiado.

El BaseRepository proporciona operaciones fundamentales sin ningún filtrado automático. Este nivel se utiliza para entidades que no requieren aislamiento, como configuraciones globales del sistema. Ofrece métodos para buscar por identificador, listar todos los registros con paginación, contar registros, y verificar existencia. Todos los métodos aceptan opciones de eager loading para prevenir N+1.

El TenantRepository extiende el base añadiendo filtrado automático por tenant_id. Todas las operaciones heredadas ahora requieren un parámetro de tenant y lo aplican automáticamente a las consultas mediante cláusulas where inyectadas. Este nivel se utiliza para entidades que

pertenecen a un tenant específico pero no están asociadas a una sucursal particular, como los ingredientes globales del restaurante o los alérgenos definidos a nivel de tenant.

El BranchRepository añade una capa adicional de filtrado por branch_id sobre el filtrado de tenant. Las operaciones a este nivel requieren tanto tenant como branch, aplicando ambos filtros a todas las consultas. Este nivel se utiliza para entidades como categorías, mesas o sectores que pertenecen a una sucursal específica. El método find_by_branch localiza entidades de una sucursal específica. El método find_by_branches acepta una lista de identificadores de sucursales para filtrar por múltiples ubicaciones, útil para managers que tienen acceso a varias sucursales.

Los repositorios encapsulan toda la lógica de consultas SQLAlchemy, manteniendo el código de servicios limpio de detalles de persistencia. Un servicio simplemente invoca métodos del repositorio con parámetros de negocio, sin construir queries ni manejar sesiones. El método find_by_id localiza una entidad por su identificador primario, retornando None si no existe o no pertenece al tenant especificado. El método find_all lista entidades aplicando filtros opcionales, ordenamiento, y paginación eficiente a nivel de base de datos. El método find_by_ids recupera múltiples entidades por una lista de identificadores en una sola consulta, evitando el antipatrón de N consultas para N identificadores. El método count retorna el número de registros que cumplen los criterios especificados. El método exists verifica eficientemente si existe al menos un registro que cumpla los criterios.

Los repositorios configuran estrategias de eager loading para prevenir el problema de N+1 queries. La estrategia selectinload ejecuta una segunda consulta para cargar las relaciones, resultando en exactamente dos queries independientemente del número de registros, siendo óptima para relaciones uno a muchos. La estrategia joinedload utiliza un JOIN SQL para cargar las relaciones en una sola consulta, siendo óptima para relaciones uno a uno o muchos a uno. La combinación de ambas estrategias permite cargar grafos complejos de entidades eficientemente, como productos con sus precios por sucursal mediante selectinload y su receta asociada mediante joinedload.

Un detalle técnico importante es el uso correcto de comparaciones booleanas en SQLAlchemy. El fix HIGH-DEEP-04 establece que se debe usar is_(True) en lugar de doble igual True, ya que este último genera SQL incorrecto. Lo mismo aplica para comparaciones con None, donde se usa is_(None) o is_not(None).

Capítulo 6: Los Servicios de Dominio

Los servicios de dominio residen en el subdirectorio services/domain y representan el corazón de la lógica de negocio. Cada servicio se especializa en un dominio específico del negocio, encapsulando todas las reglas, validaciones, y operaciones relacionadas con ese dominio. Esta es la ubicación preferida para nueva lógica de negocio, reemplazando el patrón CRUDFactory que está deprecado.

La clase base BaseCRUDService proporciona operaciones estándar de creación, lectura, actualización, y eliminación que los servicios especializados heredan y pueden sobrescribir. Esta herencia reduce la duplicación de código mientras permite personalización donde sea necesaria. El constructor recibe la sesión de base de datos siguiendo el patrón de inyección de dependencias, junto con el modelo SQLAlchemy, el schema Pydantic de salida, el nombre de la entidad en español para mensajes de error, y configuraciones opcionales como si la entidad tiene branch_id o si soporta soft delete.

La clase BranchScopedService extiende la base para entidades que pertenecen a una sucursal específica, añadiendo automáticamente el uso de BranchRepository y métodos de listado por branch. Los servicios para categorías, mesas, sectores y tablas heredan de esta clase.

Los servicios implementan hooks que se invocan en momentos específicos del ciclo de vida de las operaciones, permitiendo personalización sin sobrescribir completamente los métodos base. Este patrón de Template Method mantiene la estructura común mientras habilita comportamiento específico por entidad.

El hook _validate_create se invoca antes de crear una entidad, recibiendo los datos propuestos y el tenant_id. Aquí el servicio puede validar reglas de negocio complejas como verificar que la categoría padre existe, verificar unicidad personalizada más allá de constraints de base de datos, o rechazar la operación con una excepción descriptiva si los datos no cumplen invariantes del negocio.

El hook _validate_update se invoca antes de actualizar una entidad, recibiendo la entidad existente y los datos propuestos. Puede validar que los cambios son permitidos según el estado actual de la entidad, verificar invariantes del negocio que dependen de valores anteriores, o prevenir modificaciones de campos protegidos como identificadores o claves foráneas críticas.

El hook _validate_delete se invoca antes de eliminar una entidad, verificando que la eliminación es permitida. Puede verificar que no existan dependencias que impedirían la eliminación sin cascada, o que el usuario tenga permisos específicos para eliminar esta entidad particular más allá del permiso genérico de rol.

El hook _after_create se invoca después de crear exitosamente una entidad, útil para efectos secundarios como publicar eventos de dominio ENTITY_CREATED a través de Redis, actualizar caches invalidando entradas relacionadas, o notificar a sistemas externos mediante webhooks.

El hook _after_update se invoca después de actualizar exitosamente, permitiendo publicar eventos de cambio ENTITY_UPDATED o sincronizar datos derivados que dependen de los valores modificados.

El hook _after_delete se invoca después de eliminar exitosamente, permitiendo publicar eventos de eliminación ENTITY_DELETED, limpiar datos relacionados en sistemas externos, o registrar la eliminación en logs de auditoría adicionales.

Entre los servicios especializados más importantes, el CategoryService gestiona categorías de productos con ordenamiento automático, calculando el siguiente orden disponible cuando se crea una categoría sin especificar orden y permitiendo reorganizar múltiples categorías en una sola operación transaccional mediante reorder_categories.

El ProductService maneja la complejidad de productos con múltiples relaciones. La creación de un producto puede incluir precios por sucursal mediante BranchProduct, asociaciones con alérgenos incluyendo tipos de presencia y niveles de riesgo, vinculación con ingredientes en la jerarquía de tres niveles, perfiles dietéticos, información de cocción con métodos y tiempos, perfiles sensoriales con sabores y texturas, modificaciones permitidas, advertencias, configuración RAG, y opcionalmente sincronización con una receta vinculada. El servicio orquesta todas estas operaciones relacionadas manteniendo consistencia transaccional.

El AllergenService gestiona alérgenos incluyendo sus reacciones cruzadas. Cuando se asocia un alérgeno a un producto, el servicio puede advertir sobre otros alérgenos con reacciones cruzadas conocidas que deberían considerarse para la seguridad alimentaria del comensal.

El StaffService implementa reglas especiales para gestión de personal. Los managers solo pueden crear staff con roles iguales o menores a su propio rol. Las modificaciones de roles requieren verificar que el usuario tiene permiso para asignar ese rol específico. La eliminación de un usuario verifica que no sea el último administrador del tenant.

El RoundService gestiona el ciclo de vida completo de pedidos desde su creación hasta su entrega. Implementa las transiciones de estado PENDING a CONFIRMED cuando el mozo verifica, CONFIRMED a SUBMITTED cuando el admin envía a cocina, SUBMITTED a IN_KITCHEN cuando la cocina comienza preparación, IN_KITCHEN a READY cuando termina, y READY a SERVED cuando se entrega. Cada transición valida que el estado anterior sea correcto, actualiza los timestamps correspondientes, y publica el evento apropiado.

El BillingService maneja la complejidad de facturación y pagos. Implementa el modelo de asignación FIFO donde los pagos se distribuyen a cargos en orden cronológico. Integra con Mercado Pago para pagos electrónicos con manejo de webhooks. Utiliza circuit breaker para resiliencia ante fallos del proveedor de pagos.

Capítulo 7: El Sistema de Routers

Los routers HTTP se organizan en grupos funcionales que reflejan las diferentes audiencias y casos de uso de la API. Esta organización facilita la aplicación de políticas de seguridad diferentes por grupo y permite que los equipos trabajen independientemente en diferentes áreas.

El grupo auth maneja autenticación sin requerir token previo. Incluye el endpoint de login que valida credenciales mediante bcrypt, genera tokens JWT de acceso con duración de quince minutos y tokens de refresh con duración de siete días, y establece el token de refresh como cookie HttpOnly para prevenir acceso desde JavaScript siguiendo el fix SEC-09. El endpoint de refresh lee el token desde la cookie HttpOnly o el cuerpo como fallback, verifica contra la blacklist de Redis, genera nuevos tokens, y rota el refresh token para invalidar el anterior siguiendo SEC-06. El endpoint de logout invoca revoke_all_user_tokens para invalidar todos los tokens del usuario en todos los dispositivos agregándolos a la blacklist de Redis, y limpia la cookie HttpOnly. El endpoint me retorna información del usuario autenticado extraída del token JWT.

El grupo public expone endpoints accesibles sin autenticación. Incluye el catálogo público de productos para mostrar el menú en pwaMenu, la lista de sucursales para selección pre-login en pwaWaiter, y los endpoints de health check básico y detallado que incluye estado de Redis y métricas de conexiones.

El grupo tables gestiona sesiones de mesa mediante tokens de mesa en lugar de tokens de usuario. Incluye la creación de sesiones cuando un comensal escanea el código QR de la mesa, verificando que la mesa no tenga ya una sesión activa, creando el primer Diner con el device_id del header para tracking, y generando un table token firmado con HMAC que contiene session_id, branch_id y tenant_id. La obtención de estado de sesión permite sincronización entre dispositivos que comparten la misma mesa.

El grupo diner implementa operaciones para comensales autenticados mediante tokens de mesa validados en el header X-Table-Token. Incluye el registro de comensales adicionales en

una sesión existente. La gestión del carrito compartido mediante endpoints para agregar, actualizar, eliminar y listar items, donde cada operación publica eventos CART_ITEM_ADDED, CART_ITEM_UPDATED o CART_ITEM_REMOVED a través de Redis para sincronización en tiempo real entre dispositivos. La creación de pedidos que combina items de todos los comensales en un Round con estado PENDING. Las operaciones de fidelización de clientes incluyendo registro con consentimiento GDPR, reconocimiento de dispositivo vinculado a customer, y sugerencias personalizadas basadas en historial.

El grupo kitchen proporciona endpoints para personal de cocina autenticado con JWT y rol KITCHEN o superior. Incluye el listado de rondas con estado SUBMITTED para la vista de nuevos pedidos y estado IN_KITCHEN para pedidos en preparación. La actualización de estado de preparación permite transiciones de SUBMITTED a IN_KITCHEN cuando la cocina comienza y de IN_KITCHEN a READY cuando termina. La gestión de tickets de cocina agrupa items de pedido para organización interna.

El grupo waiter implementa operaciones para mozos autenticados con JWT y rol WAITER o superior. Incluye la verificación de asignación de sucursal para el día actual mediante verify-branch-assignment. El listado de mesas filtradas por sectores asignados mediante WaiterSectorAssignment, donde ADMIN y MANAGER ven todas las mesas. El menú compacto para comanda rápida que retorna productos sin imágenes para reducir payload. La creación de rondas en nombre de clientes para el flujo de comanda rápida donde el mozo toma el pedido verbalmente. La confirmación de rondas que transiciona de PENDING a CONFIRMED cuando el mozo verifica el pedido en la mesa.

El grupo billing maneja pagos y facturación. Incluye la solicitud de cuenta que crea un Check y publica CHECK_REQUESTED mediante el patrón Outbox para garantía de entrega. El registro de pagos en efectivo que crea Payment y ejecuta el algoritmo de asignación FIFO a Charges pendientes. La integración con Mercado Pago para pagos electrónicos incluyendo creación de preferencias de pago y recepción de webhooks de confirmación con verificación de firma HMAC. El procesamiento de webhooks actualiza el estado del pago y publica PAYMENT_APPROVED o PAYMENT_REJECTED mediante Outbox.

El grupo admin expone operaciones CRUD completas para el panel de administración. Contiene quince routers especializados para diferentes entidades organizados por dominio. El router products gestiona productos con todas sus relaciones. El router categories maneja categorías con ordenamiento. El router subcategories gestiona subcategorías anidadas. El router allergens administra alérgenos y reacciones cruzadas. El router ingredients maneja la jerarquía de ingredientes. El router staff gestiona usuarios y roles con restricciones por rol del operador. El router branches administra sucursales. El router tables gestiona mesas físicas. El router sectors define sectores del restaurante. El router assignments maneja asignaciones diarias de mozos a sectores. El router promotions gestiona promociones con validación de fechas. El router audit proporciona acceso a logs de auditoría. El router restore permite

recuperar entidades eliminadas. El router reports genera reportes operacionales. El router tenant administra configuración del restaurante.

El grupo content gestiona contenido administrable que no encaja en otras categorías. Incluye catálogos de métodos de cocción, perfiles de sabor, texturas y tipos de cocina. La gestión de recetas con fichas técnicas de cocina incluyendo ingredientes, instrucciones, tiempos y rendimientos. La administración de promociones con branches y productos asociados. El chatbot RAG con inteligencia artificial que utiliza embeddings vectoriales en pgvector para búsqueda semántica de documentos de conocimiento.

El grupo metrics expone un endpoint para scraping de Prometheus que retorna métricas del sistema en formato text/plain, incluyendo conexiones activas, eventos procesados, errores, y latencias.

Todos los routers siguen un patrón consistente que mantiene la lógica HTTP separada de la lógica de negocio. El primer paso extrae los parámetros de la petición HTTP mediante argumentos tipados y Query con valores por defecto. El segundo paso obtiene las dependencias inyectadas incluyendo la sesión de base de datos mediante get_db y el usuario autenticado mediante current_user que valida el token JWT y extrae los claims. El tercer paso crea un PermissionContext a partir del usuario autenticado que selecciona automáticamente la estrategia de permisos apropiada según los roles. El cuarto paso valida permisos invocando métodos del contexto como require_management o require_branch_access que lanzan excepciones HTTP si el usuario no tiene los permisos necesarios. El quinto paso instancia el servicio de dominio apropiado y delega la operación, retornando el resultado que FastAPI serializa automáticamente a JSON.

Capítulo 8: El Sistema de Autenticación

La autenticación utiliza JSON Web Tokens firmados que contienen toda la información necesaria para autorizar peticiones sin consultar la base de datos en cada request. Este diseño stateless permite escalabilidad horizontal del servicio.

El token de acceso tiene una duración corta de quince minutos, limitando la ventana de exposición si un token es comprometido. Contiene claims con el identificador del usuario en el campo sub, el email para logging, el tenant_id al que pertenece, los branch_ids de sucursales a las que tiene acceso, y los roles asignados como ADMIN, MANAGER, KITCHEN o WAITER. La firma HMAC-SHA256 con JWT_SECRET garantiza que el token no ha sido modificado.

El token de refresh tiene una duración más larga de siete días, permitiendo renovar el acceso sin requerir credenciales frecuentemente. Este token se almacena como cookie HttpOnly para prevenir acceso desde JavaScript y mitigar ataques XSS. La cookie se configura con secure verdadero en producción para requerir HTTPS, samesite lax para protección CSRF, y path restringido a /api/auth. Cuando se utiliza para obtener nuevos tokens, se verifica contra una lista de tokens revocados almacenada en Redis con TTL igual a la duración del token, soportando logout efectivo. El fix SEC-06 implementa rotación del refresh token donde cada uso invalida el token anterior, previniendo reuso si es interceptado.

El token de mesa tiene una duración de tres horas, reducida desde ocho horas en el fix CRIT-04 para limitar la ventana de exposición. Contiene el session_id de la sesión de mesa, el branch_id de la sucursal, y el tenant_id del restaurante. Este token permite a los comensales operar sin cuentas de usuario, autenticándose únicamente mediante el código QR de la mesa que se codifica con el código de mesa y el slug de sucursal. La firma utiliza TABLE_TOKEN_SECRET independiente del secreto JWT.

El flujo de autenticación de usuarios comienza cuando el cliente envía credenciales al endpoint de login. El servidor verifica que el usuario existe y está activo. Valida la contraseña mediante bcrypt que incluye salt automático y es resistente a timing attacks. Genera el token de acceso con los claims del usuario. Genera el token de refresh con un identificador único JTI. Establece la cookie HttpOnly con el refresh token. Retorna el access token en el cuerpo de la respuesta junto con información básica del usuario.

Para peticiones autenticadas, el cliente incluye el access token en el header Authorization con formato Bearer seguido del token. La dependencia current_user de FastAPI extrae el token, verifica la firma con JWT_SECRET, comprueba que no haya expirado, consulta la blacklist de Redis para tokens revocados, y retorna un diccionario con los claims del usuario que incluye sub, email, tenant_id, branch_ids, y roles.

Cuando el access token expira, el cliente llama al endpoint de refresh. El servidor lee el refresh token desde la cookie HttpOnly que el navegador envía automáticamente con credentials include. Verifica la firma y expiración del refresh token. Consulta la blacklist para verificar que no ha sido revocado. Genera nuevos tokens de acceso y refresh. Revoca el refresh token anterior agregándolo a la blacklist. Establece la nueva cookie con el refresh token rotado. Retorna el nuevo access token.

El logout revoca todos los tokens del usuario invocando revoke_all_user_tokens que lista todos los JTI activos del usuario y los agrega a la blacklist de Redis con TTL de siete días. Limpia la cookie HttpOnly. Esto invalida la sesión en todos los dispositivos inmediatamente.

La autenticación de comensales por mesa funciona diferente. Cuando un comensal escanea el código QR, el frontend decodifica el código de mesa y slug de sucursal. Envía una petición al endpoint de creación de sesión sin autenticación previa. El servidor busca la mesa por código dentro de la sucursal identificada por slug. Verifica que la mesa no tenga sesión activa. Crea una nueva TableSession con timestamp de inicio. Crea el primer Diner capturando el device_id del header X-Device-Id para tracking cross-session. Genera el table token firmado con HMAC-SHA256 usando TABLE_TOKEN_SECRET. Retorna el token que el frontend almacena y usa en peticiones subsecuentes mediante el header X-Table-Token.

Capítulo 9: El Sistema de Permisos

El PermissionContext actúa como fachada para todo el sistema de autorización, proporcionando una interfaz consistente independientemente del rol del usuario. Su constructor analiza los claims del token JWT y selecciona automáticamente la estrategia de permisos apropiada basándose en los roles, eligiendo la de mayor privilegio si el usuario tiene múltiples roles.

Las propiedades del contexto exponen información extraída del token de manera tipada. La propiedad user_id retorna el identificador numérico del usuario parseado del claim sub. La propiedad tenant_id retorna el identificador del restaurante. La propiedad branch_ids retorna la lista de sucursales accesibles. La propiedad roles retorna la lista de roles asignados. Las propiedades booleanas is_admin e is_management simplifican verificaciones comunes donde is_management es verdadero si el usuario es ADMIN o MANAGER.

Los métodos de verificación de capacidad consultan la estrategia para determinar si una acción está permitida. El método can recibe una acción del enum Action que puede ser CREATE, READ, UPDATE o DELETE, junto con la entidad o tipo de entidad y opcionalmente un branch_id. Retorna un booleano permitiendo lógica condicional en el código del servicio. Los métodos can_create, can_read, can_update y can_delete son atajos convenientes para acciones específicas.

Los métodos de requerimiento lanzan excepciones si la condición no se cumple, cortocircuitando la ejecución. El método require_admin lanza ForbiddenError si el usuario no tiene rol ADMIN. El método require_management acepta administradores o managers, lanzando excepción para otros roles. El método require_branch_access verifica acceso a una sucursal específica comprobando que branch_id esté en la lista branch_ids del usuario o que sea ADMIN que tiene acceso implícito a todo.

El sistema implementa el patrón Strategy con cinco estrategias especializadas para diferentes roles. Las estrategias siguen además el Principio de Segregación de Interfaces mediante mixins que permiten componer comportamientos sin duplicación de código.

La estrategia de administrador implementada en AdminStrategy otorga acceso total a todas las operaciones del tenant. Puede crear, leer, actualizar, y eliminar cualquier entidad. No tiene restricciones de sucursal. El método filter_query retorna la query sin modificaciones adicionales más allá del filtro de tenant que ya aplican los repositorios.

La estrategia de manager implementada en ManagerStrategy permite operaciones limitadas a sus sucursales asignadas. Puede crear personal con roles iguales o menores, mesas, alérgenos, y promociones dentro de sus branches. Puede leer y actualizar la mayoría de entidades de sus sucursales. No puede eliminar entidades, delegando esa responsabilidad a administradores. El método filter_query añade una cláusula where que filtra por branch_id en la lista de branch_ids del usuario.

La estrategia de cocina implementada en KitchenStrategy es más restrictiva, enfocada en operaciones de preparación de alimentos. Define conjuntos explícitos de entidades legibles como Round, KitchenTicket, Product, Category y Recipe, y actualizables como Round y KitchenTicket. Puede leer estas entidades de sus sucursales asignadas. Puede actualizar el estado de rondas y tickets. No puede crear ni eliminar entidades, heredando NoCreateMixin y NoDeleteMixin que retornan falso para esas operaciones. El filtrado de consultas excluye automáticamente entidades de sucursales no asignadas.

La estrategia de mozo implementada en WaiterStrategy soporta operaciones de servicio al cliente. Puede crear llamadas de servicio ServiceCall, rondas Round, y registrar comensales Diner. Puede leer mesas, sesiones, rondas, productos y cuentas de sus sucursales. Puede actualizar rondas para confirmar pedidos y llamadas de servicio para marcar atención. No puede eliminar entidades. El filtrado considera tanto sucursales como potencialmente sectores asignados para el día actual mediante WaiterSectorAssignment.

La estrategia de solo lectura implementada en ReadOnlyStrategy proporciona acceso mínimo para roles especiales o no reconocidos. Solo puede leer entidades básicas, sin capacidad de crear, actualizar, o eliminar. Se utiliza como fallback cuando el usuario no tiene ningún rol reconocido.

Los mixins permiten componer comportamientos comunes entre estrategias sin duplicación. El mixin NoCreateMixin implementa can_create retornando siempre falso. El mixin NoDeleteMixin implementa can_delete retornando siempre falso. El mixin NoUpdateMixin implementa can_update retornando siempre falso. El mixin BranchFilterMixin implementa el filtrado automático de consultas por branch_ids del usuario, verificando que la entidad tenga

atributo branch_id y añadiendo el filtro correspondiente. El mixin BranchAccessMixin proporciona helpers para verificar acceso a branches específicos incluyendo _user_has_branch_access y _get_entity_branch_id.

La función get_highest_privilege_strategy recibe la lista de roles del usuario y retorna la instancia de estrategia con mayor privilegio. El orden de precedencia es ADMIN primero, luego MANAGER, KITCHEN, WAITER, y finalmente ReadOnly como fallback. Si un usuario tiene roles WAITER y MANAGER, obtiene ManagerStrategy que tiene más permisos.

Capítulo 10: El Sistema de Eventos de Dominio

El sistema de eventos permite que diferentes partes de la aplicación reaccionen a cambios sin acoplamiento directo. Cuando un servicio modifica datos, publica un evento que describe el cambio. Otros sistemas suscritos a ese tipo de evento pueden reaccionar apropiadamente, desde actualizar interfaces en tiempo real hasta sincronizar sistemas externos.

Los tipos de eventos están definidos en un módulo centralizado. Los eventos de entidad incluyen ENTITY_CREATED cuando se crea una nueva entidad, ENTITY_UPDATED cuando se modifica, ENTITY_DELETED cuando se elimina, y CASCADE_DELETE cuando la eliminación afecta entidades relacionadas. Los eventos de pedido siguen el ciclo de vida con ROUND_PENDING para nuevos pedidos, ROUND_CONFIRMED cuando el mozo verifica, ROUND_SUBMITTED cuando se envía a cocina, ROUND_IN_KITCHEN cuando la cocina comienza preparación, ROUND_READY cuando está listo, ROUND_SERVED cuando se entrega, y ROUND_CANCELED para cancelaciones. Los eventos de carrito incluyen CART_ITEM_ADDED, CART_ITEM_UPDATED, CART_ITEM_REMOVED y CART_CLEARED. Los eventos de servicio incluyen SERVICE_CALL_CREATED cuando el comensal solicita atención y SERVICE_CALL_ACKED cuando el mozo responde. Los eventos de pago incluyen CHECK_REQUESTED cuando se solicita la cuenta, CHECK_PAID cuando se completa el pago, PAYMENT_APPROVED para confirmación de pago electrónico, PAYMENT_REJECTED cuando el pago es rechazado, y PAYMENT_FAILED para errores de procesamiento.

La publicación de eventos se realiza mediante funciones especializadas del módulo events. La función publish_event recibe el pool de Redis, el canal destino, y el evento serializado. El routing de eventos a canales Redis se determina automáticamente según el tipo. Los eventos de administración van al canal admin de la sucursal siguiendo el patrón admin:branch:id. Los eventos de cocina van al canal kitchen:branch:id. Los eventos de mozo van al canal waiters:branch:id. Los eventos de sesión van al canal session:session_id para los comensales específicos de esa mesa. Los eventos con sector_id específico se enrutan solo a mozos asignados a ese sector.

Para eventos no críticos donde cierta pérdida es aceptable, la publicación se realiza directamente a Redis de manera asíncrona. Esto incluye eventos de carrito CART_*, eventos de estado intermedio de pedido como ROUND_CONFIRMED o ROUND_IN_KITCHEN, eventos de sesión de mesa, y eventos de entidad ENTITY_*.

Para eventos críticos donde la pérdida es inaceptable, el sistema implementa el patrón Outbox que garantiza consistencia entre datos y eventos mediante transacciones de base de datos. Esto incluye eventos financieros como CHECK_REQUESTED, CHECK_PAID, PAYMENT_APPROVED y PAYMENT_REJECTED. Eventos de flujo crítico como ROUND_SUBMITTED y ROUND_READY. Eventos de servicio como SERVICE_CALL_CREATED donde hay SLA de respuesta.

Cuando un servicio necesita publicar un evento garantizado, en lugar de publicar directamente a Redis, invoca write_outbox_event o sus variantes especializadas write_billing_outbox_event, write_round_outbox_event, o write_service_call_outbox_event. Esta función inserta un registro en la tabla OutboxEvent dentro de la misma transacción que modifica los datos de negocio. El registro incluye tenant_id para aislamiento multi-tenant, event_type como ROUND_SUBMITTED, aggregate_type como round o check, aggregate_id con el identificador de la entidad, payload con los datos del evento en JSON, status como PENDING, y retry_count inicializado en cero. Si la transacción de negocio falla, tanto los datos como el evento se revierten. Si la transacción tiene éxito, el evento queda persistido junto con los datos, garantizando consistencia.

Un procesador en background implementado en outbox_processor.py consulta periódicamente la tabla de outbox buscando eventos con status PENDING ordenados por created_at. Para cada evento encontrado, actualiza el status a PROCESSING para evitar procesamiento duplicado por otras instancias. Intenta publicarlo a Redis mediante el canal apropiado según el tipo de evento. Si la publicación tiene éxito, marca el registro con status PUBLISHED y registra processed_at. Si falla, incrementa retry_count, registra el error en last_error, y vuelve a status PENDING para reintento. El reintento usa backoff exponencial comenzando en un segundo y duplicando hasta un máximo de treinta segundos. Después de cinco intentos fallidos, el evento se marca como FAILED y se registra para intervención manual.

El procesador se inicia durante el startup de la aplicación en lifespan.py como una tarea asíncrona. Se configura con un intervalo de polling de un segundo y un tamaño de lote de cincuenta eventos por ciclo. Durante el shutdown, se cancela la tarea esperando que complete el lote actual.

Este diseño garantiza que los eventos eventualmente se publican si los datos se persistieron, eliminando la posibilidad de inconsistencias donde los datos cambian pero el evento se pierde. El trade-off es mayor latencia en la entrega del evento, típicamente entre uno y dos segundos

adicionales, y complejidad del procesador. Para eventos donde la latencia sub-segundo es crítica y cierta pérdida es aceptable, se usa publicación directa.

Capítulo 11: Middlewares de Seguridad

El middleware de headers de seguridad implementado en SecurityHeadersMiddleware añade headers protectores a todas las respuestas HTTP, implementando defensas en profundidad contra diversos vectores de ataque web.

El header X-Content-Type-Options con valor nosniff previene que navegadores intenten adivinar el tipo de contenido, mitigando ataques de sniffing MIME que podrían ejecutar scripts disfrazados como otros tipos de archivo.

El header X-Frame-Options con valor DENY previene que la página sea embebida en frames de otros sitios, mitigando ataques de clickjacking donde un atacante superpone una interfaz maliciosa sobre la aplicación.

El header Content-Security-Policy define una política restrictiva de fuentes de contenido siguiendo el fix HIGH-MID-01. Establece default-src como self para permitir solo recursos del mismo origen. Configura script-src como self sin unsafe-inline para prevenir scripts inyectados. Restringe style-src, img-src, font-src y connect-src a fuentes conocidas. Esta política mitiga ataques XSS al impedir la ejecución de código no autorizado.

El header Strict-Transport-Security se añade solo en producción e indica a navegadores que siempre deben usar HTTPS para este dominio. Incluye includeSubDomains y especifica max-age de un año en segundos, previniendo downgrades a HTTP donde un atacante podría interceptar tráfico.

El header Referrer-Policy controla cuánta información se envía en el header Referer cuando el usuario navega desde la aplicación a otros sitios. El valor strict-origin-when-cross-origin envía solo el origen para navegación cross-origin, protegiendo paths sensibles de ser filtrados a terceros.

El header Permissions-Policy deshabilita APIs de navegador que no son necesarias para la aplicación, incluyendo geolocation, microphone, y camera. Esto reduce la superficie de ataque y previene que código malicioso abuse de estas APIs.

El middleware de validación de Content-Type implementado en ContentTypeValidationMiddleware verifica que las peticiones con cuerpo utilicen tipos de contenido válidos, previniendo ataques que explotan parsing inesperado de formatos siguiendo el fix HIGH-04.

Para peticiones POST, PUT, y PATCH que modifican datos, el middleware verifica que el header Content-Type sea application/json o application/x-www-form-urlencoded. Otros tipos de contenido resultan en una respuesta 415 Unsupported Media Type sin procesar el cuerpo, previniendo que payloads maliciosos en formatos inesperados lleguen a los handlers.

Ciertos paths están exentos de esta validación y se mantienen en una lista explícita. Los endpoints de webhook de Mercado Pago aceptan el formato específico del proveedor que puede diferir. Los endpoints de health check no tienen cuerpo que validar. Esta lista de excepciones se mantiene mínima y documentada.

La configuración de CORS se centraliza en cors.py y controla qué orígenes pueden realizar peticiones a la API desde navegadores web, previniendo que sitios maliciosos realicen peticiones en nombre de usuarios autenticados.

En desarrollo, la configuración permite orígenes localhost en los puertos utilizados por las aplicaciones frontend. Esto incluye el puerto 5176 para pwaMenu, 5177 para Dashboard, 5178 y 5179 para pwaWaiter, junto con sus equivalentes usando la dirección IP 127.0.0.1 en lugar del hostname.

En producción, los orígenes permitidos se configuran mediante la variable de entorno ALLOWED_ORIGINS como lista separada por comas. Solo los orígenes explícitamente listados pueden realizar peticiones, rechazando cualquier otro origen con un error CORS que el navegador interpreta bloqueando la respuesta.

Los métodos HTTP permitidos incluyen GET para lecturas, POST para creaciones, PUT y PATCH para actualizaciones, DELETE para eliminaciones, y OPTIONS para preflight requests. Los headers permitidos incluyen Authorization para tokens JWT, Content-Type para especificar el tipo de cuerpo, X-Table-Token para autenticación de mesa, X-Request-ID para trazabilidad de peticiones, y X-Device-Id para identificación de dispositivo en tracking cross-session.

El middleware se registra último en la cadena porque debe procesar la respuesta después de todos los demás middlewares pero su verificación de origen ocurre primero.

Capítulo 12: Rate Limiting

El sistema implementa rate limiting para proteger contra abuso y ataques de denegación de servicio. La implementación utiliza slowapi que integra con FastAPI y almacena contadores en Redis para compartir estado entre múltiples instancias del servicio.

Los endpoints de login tienen límites estrictos para prevenir ataques de fuerza bruta contra credenciales. Se aplica un límite de cinco intentos por minuto por dirección IP. También se aplica un límite por email para prevenir que un atacante distribuido apunte a una cuenta específica. Después de exceder el límite, las peticiones reciben respuesta 429 Too Many Requests con un header Retry-After indicando cuántos segundos esperar.

Los endpoints de pago tienen límites moderados para prevenir abuso mientras permiten operaciones legítimas. La solicitud de cuenta check/request permite diez peticiones por minuto por sesión de mesa. Los pagos en efectivo cash/pay permiten veinte por minuto. Las operaciones de Mercado Pago mercadopago/* permiten cinco por minuto por sesión, siendo más restrictivas debido al costo de integración externa.

El registro de comensales diner/register tiene un límite de veinte por minuto por dirección IP para prevenir creación masiva de diners falsos que podrían saturar las mesas.

Los endpoints administrativos tienen límites más generosos de cien peticiones por minuto ya que requieren autenticación JWT y los usuarios legítimos pueden necesitar realizar múltiples operaciones en secuencia durante la gestión del restaurante.

El rate limiting utiliza Redis para mantener contadores compartidos entre instancias del servicio mediante una clave que combina el identificador del límite con el identificador del cliente. Esto garantiza que los límites se apliquen correctamente incluso con múltiples réplicas de la API detrás de un balanceador de carga. Los contadores expiran automáticamente después de la ventana de tiempo, liberando memoria de Redis.

Capítulo 13: Flujo de Creación de Producto

Cuando un administrador crea un nuevo producto desde el panel de administración, el sistema orquesta una secuencia completa que valida permisos, persiste datos con todas sus relaciones, y notifica a sistemas conectados.

El navegador envía una petición POST al endpoint admin/products con el cuerpo JSON conteniendo los datos del producto incluyendo nombre, descripción, categoría, subcategoría opcional, imagen, precios por sucursal, alérgenos con tipos de presencia, ingredientes, perfiles dietéticos, información de cocción, y configuración RAG opcional.

El middleware de CORS verifica que el origen del navegador esté en la lista de permitidos. El middleware de Content-Type valida que sea application/json. El middleware de seguridad prepara los headers protectores para la eventual respuesta.

El router de productos extrae el cuerpo y lo valida contra el schema Pydantic de creación de producto. Los campos obligatorios como nombre y category_id deben estar presentes. Los tipos deben coincidir con las especificaciones. Las validaciones de formato como longitud máxima de nombre, formato de URL de imagen, y rangos de precios se verifican automáticamente lanzando 422 Unprocessable Entity si fallan.

El router obtiene el usuario autenticado del token JWT mediante la dependencia current_user que verifica firma, expiración, y blacklist. Crea un PermissionContext que analiza los roles del usuario y selecciona la estrategia correspondiente. Invoca require_management que verifica que el usuario tenga rol ADMIN o MANAGER, lanzando 403 Forbidden si no cumple. Si el usuario es MANAGER, verifica además mediante require_branch_access que tenga acceso a todas las sucursales especificadas en los precios del producto.

El router instancia ProductService pasando la sesión de base de datos y delega la creación invocando create_full. El servicio invoca su hook de validación _validate_create que verifica que la categoría existe en el tenant, que la subcategoría si se especifica pertenece a esa categoría, que el nombre no esté duplicado dentro de la categoría, y que las URLs de imagen pasen validación SSRF previniendo acceso a IPs internas.

El servicio crea la instancia del modelo Product con los datos proporcionados, estableciendo el tenant_id del contexto del usuario. Invoca helpers de auditoría para registrar created_by_id y created_by_email con los datos del usuario actual. Añade la entidad a la sesión de SQLAlchemy.

El servicio procesa las relaciones del producto. Crea registros BranchProduct para cada precio por sucursal especificado. Crea registros ProductAllergen para cada alérgeno con su tipo de presencia y nivel de riesgo. Crea registros ProductIngredient vinculando con ingredientes de la

jerarquía. Crea o actualiza el DietaryProfile asociado. Crea registros ProductCookingMethod con tiempos de preparación. Crea registros ProductFlavor y ProductTexture para el perfil sensorial. Crea registros de modificaciones permitidas y advertencias. Configura RAGConfig si se especifica.

Si el producto especifica inherits_from_recipe con una receta vinculada, el servicio invoca _sync_from_recipe que copia los alérgenos de la receta al producto, manteniendo sincronización automática.

El servicio realiza un solo commit transaccional que persiste el producto y todas sus relaciones atómicamente. Si cualquier operación falla, todo se revierte.

Después del commit exitoso, el hook _after_create publica un evento ENTITY_CREATED a Redis mediante el canal admin de la sucursal. El WebSocket Gateway recibe el evento mediante su suscripción Redis. Lo distribuye a las conexiones administrativas de esa sucursal. Los dashboards conectados reciben el evento via WebSocket y actualizan su interfaz mostrando el nuevo producto sin necesidad de refresh manual.

El servicio transforma la entidad persistida en el schema de salida Pydantic ProductOutput y retorna al router. El router retorna este schema que FastAPI serializa a JSON con código 200 OK. Los headers de seguridad añadidos por el middleware acompañan la respuesta al navegador.

Capítulo 14: Flujo de Pedido desde Escaneo QR hasta Entrega

El ciclo de vida completo de un pedido atraviesa múltiples estados, involucra diferentes actores, y genera eventos que sincronizan todas las interfaces en tiempo real.

Cuando un comensal escanea el código QR de la mesa con su dispositivo móvil, el navegador decodifica la información embebida que incluye el código de mesa como INT-01 y el slug de sucursal. El frontend de pwaMenu envía una petición POST al endpoint tables/code/INT-01/session con el query parameter branch_slug. No se requiere autenticación para este endpoint inicial.

El router busca la sucursal por su slug único. Busca la mesa por su código dentro de esa sucursal específica, ya que los códigos de mesa no son únicos globalmente sino solo dentro de cada sucursal. Verifica que la mesa no tenga ya una sesión activa.

El servicio de mesas crea una nueva instancia de TableSession con el timestamp de inicio. Crea un primer Diner asociado a la sesión, capturando el device_id del header X-Device-Id para tracking cross-session que permitirá reconocer al comensal en visitas futuras. Genera un table token firmado con HMAC-SHA256 usando TABLE_TOKEN_SECRET que contiene session_id, branch_id, y tenant_id con expiración de tres horas.

El servicio publica un evento TABLE_SESSION_STARTED a Redis. El WebSocket Gateway lo distribuye a todos los mozos de esa sucursal mediante el canal waiters:branch:id. En pwaWaiter, los mozos asignados al sector de esa mesa ven la mesa cambiar de estado libre a ocupada con animación de blink azul. En el Dashboard, la mesa muestra estado ocupada en rojo.

El router retorna el table token y la información de sesión al frontend de pwaMenu. El frontend almacena el token en memoria y lo incluye en todas las peticiones subsecuentes mediante el header X-Table-Token.

El comensal navega el menú, aplica filtros de alérgenos o preferencias dietéticas, y agrega items al carrito. Cada operación de agregar invoca POST diner/cart/add con el producto y cantidad. El servicio crea un CartItem vinculado a la sesión y al diner específico. Publica CART_ITEM_ADDED al canal de sesión. Otros dispositivos en la misma mesa reciben el evento y actualizan su vista del carrito compartido, mostrando quién agregó cada item con su nombre y color identificador.

Cuando los comensales están listos, uno de ellos inicia el proceso de confirmación grupal. El sistema muestra un panel de confirmación donde cada comensal indica que está listo. Cuando todos confirman, el frontend envía POST diner/orders con el carrito completo.

El servicio de pedidos verifica idempotencia mediante un idempotency_key único para prevenir duplicados si hay reintento. Crea un Round con todos los items del carrito combinando los de todos los comensales. Establece status PENDING. Captura el precio de cada producto al momento del pedido en RoundItem para preservar el precio histórico. Limpia los CartItems de la sesión. Publica CART_CLEARED para sincronizar que el carrito está vacío en todos los dispositivos.

Publica ROUND_PENDING al canal de waiters y admin. En pwaWaiter, los mozos asignados al sector ven una notificación de nuevo pedido con la mesa parpadeando en amarillo. En el

Dashboard, la mesa muestra badge Pendiente en amarillo. La cocina NO recibe este evento ya que primero debe verificarlo un mozo.

Un mozo se acerca a la mesa, verifica verbalmente que el pedido coincide con lo solicitado por los comensales, y confirma desde pwaWaiter. Invoca PATCH waiter/rounds/id/confirm. El servicio valida que el round está en PENDING. Actualiza a CONFIRMED registrando confirmed_by_user_id con el mozo que verificó. Publica ROUND_CONFIRMED al canal admin.

En el Dashboard, la mesa muestra badge Confirmado en azul. El administrador o manager ve el pedido verificado y puede enviarlo a cocina mediante el botón correspondiente. Invoca PATCH admin/rounds/id/submit. El servicio valida que el round está en CONFIRMED. Actualiza a SUBMITTED registrando submitted_at con el timestamp. Invoca write_round_outbox_event para garantizar entrega del evento crítico. Realiza commit que persiste tanto el cambio de estado como el OutboxEvent atómicamente.

El procesador Outbox detecta el nuevo evento pendiente, lo marca como procesando, y publica ROUND_SUBMITTED a los canales kitchen, admin, y waiters. Las terminales de cocina reciben el evento y muestran el pedido en la columna Nuevos. El Dashboard actualiza el badge a En Cocina en azul.

El personal de cocina comienza preparación y actualiza desde la interfaz de cocina invocando PATCH kitchen/rounds/id/in_progress. El servicio valida SUBMITTED, actualiza a IN_KITCHEN. Publica ROUND_IN_KITCHEN a todos los canales incluyendo el de sesión. Los comensales en pwaMenu ven que su pedido está siendo preparado.

Cuando la cocina termina, marca el pedido como listo invocando PATCH kitchen/rounds/id/ready. El servicio valida IN_KITCHEN, actualiza a READY. Invoca write_round_outbox_event por ser evento crítico. Publica ROUND_READY a todos los canales incluyendo sesión. Los comensales reciben notificación de que su pedido está listo. El Dashboard muestra badge Listo en verde. Si hay otros pedidos de la misma mesa aún en cocina, muestra badge combinado Listo + Cocina en naranja con animación continua para alertar al mozo.

El mozo asignado al sector recibe notificación prominente, recoge el pedido de la cocina, y lo entrega a la mesa. Confirma entrega desde pwaWaiter invocando PATCH waiter/rounds/id/served. El servicio actualiza a SERVED. Publica ROUND_SERVED. Las interfaces muestran el pedido como completado. Las métricas de tiempo de servicio se calculan comparando timestamps y almacenan para reportes operacionales.

Capítulo 15: El Sistema de Pagos

El procesamiento de pagos implementa patrones de resiliencia para manejar las complejidades de operaciones financieras donde los fallos deben manejarse cuidadosamente y la pérdida de eventos es inaceptable.

El modelo de asignación FIFO implementado en allocation.py distribuye pagos a cargos en orden cronológico. Cuando llega un pago, el sistema obtiene los cargos de la cuenta que tienen saldo pendiente, ordenados por fecha de creación. Itera asignando el monto del pago al cargo más antiguo primero, creando un registro Allocation que vincula el pago con el cargo por el monto asignado. Si queda excedente después de saldar el primer cargo, continúa con el siguiente. El proceso repite hasta agotar el monto del pago o saldar todos los cargos. Si queda saldo de pago sin asignar, se registra como crédito a favor.

El circuit breaker implementado en circuit_breaker.py protege contra fallos en cascada cuando el proveedor de pagos o Redis experimentan problemas. Mantiene un contador de fallos consecutivos. Cuando se alcanzan cinco fallos, el breaker transiciona de estado CLOSED a OPEN. En estado abierto, rechaza nuevas operaciones inmediatamente retornando error sin intentar la operación, evitando acumular timeouts que degradarían el sistema completo. Después de treinta segundos en estado abierto, transiciona a HALF_OPEN y permite una operación de prueba. Si tiene éxito, vuelve a CLOSED y resume operación normal. Si falla, vuelve a OPEN reiniciando el timer.

La integración con Mercado Pago maneja el flujo completo de pagos electrónicos. Cuando el comensal solicita pagar con tarjeta, el frontend invoca el endpoint para crear una preferencia de pago. El servicio construye la preferencia con los items de la cuenta, URLs de retorno para éxito y fallo, y configuración de webhooks. Invoca la API de Mercado Pago con el access token configurado. Retorna la URL de checkout donde el comensal completa el pago en la interfaz de Mercado Pago.

Cuando el pago se procesa, Mercado Pago envía un webhook POST al endpoint configurado. El middleware permite este path específico sin validación de Content-Type ya que el formato lo determina Mercado Pago. El handler verifica la autenticidad del webhook mediante la firma HMAC incluida en headers, comparando con el secreto configurado en MERCADOPAGO_WEBHOOK_SECRET. Si la firma no coincide, rechaza el webhook con 401.

El handler obtiene los detalles completos del pago invocando la API de Mercado Pago con el payment_id incluido en el webhook. Determina el estado: approved, rejected, o pending. Localiza el pago interno mediante el external_reference que vincula con la cuenta. Actualiza el estado del Payment y registra los detalles de la transacción.

Si el pago está aprobado, invoca el algoritmo FIFO para asignar a cargos pendientes. Invoca write_billing_outbox_event con tipo PAYMENT_APPROVED para garantizar que el evento de confirmación llegue a todas las interfaces. Si el pago es rechazado, registra el motivo de rechazo e invoca write_billing_outbox_event con PAYMENT_REJECTED.

Si el webhook falla temporalmente por problemas de red o base de datos, el reintentador de webhooks lo procesará nuevamente. Mantiene una cola de webhooks pendientes con backoff exponencial entre intentos, comenzando en un segundo y duplicando hasta treinta segundos máximo, con jitter aleatorio para evitar thundering herd. Después de cinco reintentos fallidos, marca el webhook para intervención manual y genera alerta.

Capítulo 16: El Sistema RAG de Chatbot

El sistema de Retrieval-Augmented Generation proporciona un chatbot inteligente que responde consultas sobre el menú utilizando documentos ingresados y búsqueda semántica mediante embeddings vectoriales.

Los documentos de conocimiento se almacenan en KnowledgeDocument con su contenido textual y su embedding vectorial generado por un modelo de lenguaje. El embedding es un vector de alta dimensionalidad que captura el significado semántico del texto, permitiendo búsqueda por similitud conceptual en lugar de coincidencia exacta de palabras clave. La extensión pgvector de PostgreSQL almacena estos vectores y proporciona operadores de similitud eficientes.

Las recetas pueden ingresarse automáticamente al sistema de conocimiento mediante el endpoint recipes/id/ingest. El sistema extrae el contenido relevante incluyendo nombre, descripción, ingredientes con cantidades, instrucciones de preparación, tiempos de cocción, y rendimientos. Genera el embedding de este texto concatenado. Crea o actualiza el KnowledgeDocument asociado. Esto permite que el chatbot responda preguntas sobre preparación de platos basándose en las fichas técnicas de cocina.

Cuando un usuario hace una pregunta al chatbot mediante el endpoint de chat, el sistema genera un embedding de la consulta usando el mismo modelo. Ejecuta una búsqueda de similitud en pgvector para encontrar los documentos más cercanos semánticamente, típicamente los cinco más relevantes. La distancia coseno entre vectores determina la similitud donde menor distancia indica mayor relevancia.

El sistema construye un prompt que incluye los documentos recuperados como contexto, instrucciones sobre el rol del asistente como experto en el menú del restaurante, y la pregunta del usuario. Envía este prompt al modelo de lenguaje configurado, que puede ser Ollama corriendo localmente o un servicio cloud.

El modelo genera una respuesta informada por el contexto de los documentos recuperados. Esta respuesta se registra en ChatLog junto con la pregunta original para análisis posterior y mejora del sistema. Se retorna al usuario como respuesta del endpoint.

Este enfoque RAG combina las ventajas de búsqueda semántica que encuentra información relevante aunque las palabras exactas no coincidan, con las capacidades de generación de lenguaje natural que produce respuestas coherentes y contextuales en lugar de simplemente retornar fragmentos de documentos.

Capítulo 17: Paginación y Consultas Optimizadas

La API implementa paginación consistente mediante un modelo Pagination y una dependencia get_pagination que los routers utilizan uniformemente para endpoints de listado.

El modelo de paginación define parámetros offset y limit con valores por defecto razonables y máximos configurados. El offset por defecto es cero, iniciando desde el primer registro. El límite por defecto varía según la entidad, típicamente cincuenta para entidades ligeras como categorías y veinte para entidades con muchas relaciones cargadas como productos con precios y alérgenos. Los límites máximos previenen que clientes soliciten conjuntos de datos excesivamente grandes que sobrecargarían memoria y red. Para productos el máximo es quinientos, para staff doscientos.

La respuesta de endpoints paginados incluye los items solicitados junto con metadatos de paginación. Los metadatos indican el offset actual desde donde se inició, el limit aplicado que puede diferir del solicitado si excedía el máximo, y el total de registros disponibles que cumplen los filtros independientemente de la paginación. Esta información permite a los clientes implementar navegación por páginas mostrando número de página y total, o scroll infinito cargando más registros al llegar al final.

La paginación se aplica a nivel de base de datos mediante cláusulas OFFSET y LIMIT en las consultas SQL, no filtrando resultados en memoria después de cargarlos. Esto es crucial para

rendimiento con grandes volúmenes de datos ya que solo se transfieren de la base de datos los registros que se retornarán.

El conteo total se obtiene con una consulta COUNT separada y optimizada que no carga los datos de los registros. Algunos endpoints omiten el conteo total cuando no es necesario para la interfaz, evitando esta consulta adicional.

Los índices de base de datos optimizan las consultas más frecuentes. Un índice sobre tenant_id e is_active acelera los listados filtrados por tenant que son prácticamente todas las consultas. Un índice sobre branch_id e is_active optimiza listados por sucursal. Un índice sobre status en Round acelera las consultas de cocina por estado. Un índice sobre submitted_at permite ordenamiento eficiente de la cola de cocina. Un índice compuesto sobre branch_id y status optimiza las consultas de rondas pendientes por sucursal que son muy frecuentes.

Capítulo 18: Soft Delete y Restauración

El sistema de soft delete preserva datos eliminados para auditoría y potencial recuperación, marcando registros como inactivos en lugar de eliminarlos físicamente. Esto cumple con requisitos de retención de datos, permite recuperación ante errores, y mantiene integridad referencial histórica.

La función soft_delete recibe la entidad a eliminar junto con el identificador y email del usuario que realiza la operación. Establece is_active en falso haciendo la entidad invisible para consultas normales. Registra el timestamp actual en deleted_at para saber cuándo se eliminó. Almacena el usuario en deleted_by_id y deleted_by_email para saber quién lo eliminó. No realiza commit, permitiendo que la operación sea parte de una transacción mayor que podría incluir otras operaciones relacionadas.

La función cascade_soft_delete extiende esto para manejar entidades relacionadas manteniendo consistencia referencial. El sistema mantiene un diccionario de relaciones de cascada que define qué entidades deben eliminarse junto con cada tipo de entidad. Cuando se elimina una categoría, también se eliminan sus subcategorías. Cuando se elimina una subcategoría, se eliminan sus productos. Cuando se elimina un producto, se eliminan sus precios por sucursal, asociaciones con alérgenos, y otras relaciones dependientes. La función retorna un diccionario describiendo todas las entidades afectadas por tipo y sus identificadores para logging de auditoría.

La función restore_entity revierte un soft delete, reactivando una entidad previamente eliminada. Establece is_active en verdadero haciéndola visible nuevamente. Limpia deleted_at, deleted_by_id, y deleted_by_email. Registra la restauración actualizando updated_at, updated_by_id, y updated_by_email con el usuario que restaura. Para entidades con cascada, ofrece la opción de restaurar también las entidades relacionadas que fueron eliminadas en la misma operación de cascada, verificando el timestamp de deleted_at para identificar cuáles fueron eliminadas juntas.

El endpoint administrativo admin/restore permite a administradores recuperar entidades eliminadas. El listado GET admin/restore muestra entidades eliminadas con sus metadatos incluyendo quién las eliminó, cuándo, y el tipo de entidad. Los filtros permiten buscar por tipo de entidad, rango de fechas de eliminación, y usuario que eliminó. La restauración POST admin/restore/type/id ejecuta la restauración con opción de incluir cascada. Solo usuarios con rol ADMIN pueden acceder a este endpoint ya que la restauración puede tener implicaciones significativas en el estado del sistema.

Conclusión

La REST API del ecosistema Integrador representa una implementación madura de principios de arquitectura limpia adaptados a las necesidades específicas de la industria de restauración. Su diseño en capas separa responsabilidades de manera que cada componente puede evolucionar, testearse, y mantenerse independientemente.

El sistema de modelos con AuditMixin proporciona capacidades uniformes de auditoría y soft delete que cumplen con requisitos de trazabilidad empresarial. Más de cincuenta modelos organizados en veinte archivos modulares mantienen el código navegable mientras soportan la complejidad de relaciones entre entidades de un restaurante moderno.

Los repositorios encapsulan consultas con aislamiento multi-tenant automático mediante TenantRepository y BranchRepository, eliminando la posibilidad de fugas de datos entre restaurantes. La jerarquía de tres niveles proporciona el grado apropiado de filtrado para cada tipo de entidad.

Los servicios de dominio concentran la lógica de negocio con hooks de ciclo de vida que permiten personalización sin duplicación. Cada servicio mantiene las invariantes de su dominio, valida reglas complejas, y orquesta operaciones que involucran múltiples entidades y la publicación de eventos.

Los routers permanecen delgados delegando toda la lógica a servicios, facilitando el testing y manteniendo clara la separación entre preocupaciones HTTP y lógica de negocio. La organización por grupos funcionales permite que diferentes audiencias de la API tengan endpoints dedicados con políticas de seguridad apropiadas.

El sistema de permisos basado en estrategias proporciona control de acceso granular que se adapta automáticamente al rol del usuario. Los mixins permiten componer comportamientos comunes evitando duplicación mientras mantienen flexibilidad para casos especiales de cada rol.

El sistema de eventos de dominio desacopla la notificación de cambios de la lógica que los produce. El patrón Outbox garantiza consistencia para eventos críticos financieros y de workflow donde la pérdida sería inaceptable, mientras eventos no críticos usan publicación directa para menor latencia.

Los middlewares de seguridad implementan defensa en profundidad con headers que mitigan vectores de ataque comunes. La validación de Content-Type, el rate limiting, la configuración restrictiva de CORS, y las cookies HttpOnly para refresh tokens completan una postura de seguridad robusta.

Esta arquitectura sustenta las operaciones diarias de restaurantes, procesando desde la gestión de catálogos de productos hasta el flujo completo de pedidos desde el escaneo del QR por el comensal hasta la entrega en mesa, con trazabilidad completa de cada operación y notificaciones en tiempo real a todos los actores involucrados mediante la integración con el WebSocket Gateway.

estado a través de Redis. Esta arquitectura permite que un restaurante con múltiples sucursales pueda gestionar simultáneamente cientos de mesas activas sin degradación perceptible del rendimiento.

Principios Arquitectónicos Fundamentales

La arquitectura del sistema Integrador se fundamenta en cinco principios rectores que guían todas las decisiones de diseño y que han sido refinados a

través de múltiples ciclos de auditoría y optimización durante enero y febrero de dos mil veintiséis.

El principio de Separación de Responsabilidades establece que cada componente del sistema posee una responsabilidad claramente definida y acotada. El Dashboard gestiona operaciones administrativas, pwaMenu maneja la experiencia del cliente, pwaWaiter optimiza el flujo de trabajo del mesero, el REST API procesa lógica de negocio, y el WebSocket Gateway orquesta comunicación en tiempo real. Esta separación no es meramente organizativa sino que se refleja en la estructura física del código, donde cada componente reside en su propio directorio con sus propias dependencias y configuraciones.

El principio de Arquitectura Limpia se implementa estrictamente en el backend, donde los routers actúan como controladores delgados que únicamente manejan concerns HTTP, los servicios de dominio encapsulan toda la lógica de negocio, y los repositorios abstraen completamente el acceso a datos. Esta estructura garantiza que los cambios en una capa no propaguen efectos colaterales a otras, facilitando tanto el testing como la evolución del sistema.

El Diseño Reactivo permea todos los frontends mediante Zustand como gestor de estado con patrones de suscripción selectiva que previenen re-renderizados innecesarios. La arquitectura de eventos del WebSocket Gateway permite que los cambios se propaguen instantáneamente a todos los clientes conectados sin polling, manteniendo sincronización en

tiempo real entre múltiples dispositivos que interactúan con la misma sesión de mesa.

La Seguridad en Profundidad implementa múltiples capas de protección incluyendo autenticación JWT con tokens de corta duración de quince minutos para access tokens, validación de origen en WebSockets, rate limiting por endpoint y por conexión, validación exhaustiva de entrada para prevenir ataques de inyección, y un patrón fail-closed donde cualquier error de seguridad resulta en denegación de acceso.

La Escalabilidad Horizontal se logra mediante sharding de locks, broadcast paralelo con batching, y pools de conexiones Redis configurables. El sistema implementa worker pools para broadcasting y circuit breakers para resiliencia ante fallos de dependencias externas.

Stack Tecnológico

El stack tecnológico ha sido seleccionado para maximizar productividad de desarrollo, rendimiento en producción, y mantenibilidad a largo plazo. En la capa frontend, el Dashboard utiliza React versión diecinueve con Zustand y TailwindCSS cuatro, aprovechando el React Compiler para memoización automática y los selectores de Zustand versión cinco. La aplicación pwaMenu también emplea React diecinueve con Zustand e i18next para internacionalización completa en español, inglés y portugués, funcionando como PWA offline-first. La aplicación pwaWaiter implementa React diecinueve con Zustand y Push API

para notificaciones nativas, con agrupación por sectores.

En la capa de servicios backend, el REST API está construido con FastAPI, SQLAlchemy dos punto cero y Pydantic versión dos, aprovechando async nativo, tipado fuerte y validación automática. El WebSocket Gateway utiliza FastAPI WebSocket con Redis Streams para conexiones bidireccionales y entrega garantizada de eventos.

La capa de datos emplea PostgreSQL dieciséis con la extensión pgvector para soporte vectorial necesario en RAG, proporcionando garantías ACID completas. Redis siete funciona como capa de cache y mensajería, soportando Pub/Sub, Streams, rate limiting mediante scripts Lua, y blacklist de tokens. Docker Compose orquesta todos los servicios en ambiente local de desarrollo.

Topología de Componentes

La arquitectura del sistema se organiza en tres capas horizontales que comunican mediante protocolos bien definidos. La capa de presentación comprende el Dashboard operando en el puerto cinco mil ciento setenta y siete para administradores y managers, con quince stores Zustand y cien tests Vitest. La aplicación pwaMenu corre en el puerto cinco mil ciento setenta y seis para clientes, con arquitectura de store modular e internacionalización trilingüe. La aplicación pwaWaiter opera en el puerto cinco mil ciento setenta y ocho para meseros, con tres stores y

agrupación por sectores. Las integraciones externas como Mercado Pago comunican mediante webhooks.

Todos los frontends se conectan a la capa de servicios mediante HTTP REST y WebSocket. El REST API en el puerto ocho mil contiene nueve grupos de routers con dieciséis routers administrativos, diez servicios de dominio siguiendo Clean Architecture, y repositorios TenantRepository y BranchRepository. El WebSocket Gateway en el puerto ocho mil uno comprende doce mil seiscientas cinco líneas organizadas en cincuenta y un archivos Python, implementando el Connection Manager, Redis Subscriber con sistema híbrido de Streams y Pub/Sub, Event Router con filtrado por tenant, y Worker Pool con diez workers para escalabilidad.

La capa de datos incluye PostgreSQL en el puerto cinco mil cuatrocientos treinta y dos con cincuenta y dos modelos SQLAlchemy distribuidos en dieciocho archivos de dominio, todos heredando de AuditMixin para trazabilidad universal. Redis opera en el puerto seis mil trescientos ochenta, soportando Streams para eventos críticos, Pub/Sub para eventos de tiempo real, Token Blacklist para revocación de sesiones, Rate Limiting mediante scripts Lua atómicos, y Sector Cache con TTL de sesenta segundos.

Dashboard: Centro de Control Administrativo

El Dashboard constituye el centro de control administrativo del sistema, implementado como una Single Page Application con React diecinueve y el

React Compiler habilitado para memoización automática. La aplicación gestiona quince stores Zustand con persistencia en localStorage, cada uno especializado en un dominio específico incluyendo autenticación, sucursales, categorías, productos, alérgenos, personal, promociones y mesas.

La arquitectura interna se organiza en diecinueve páginas funcionales que cubren todo el espectro administrativo: gestión de sucursales y sectores, catálogo completo con precios por sucursal, sistema de alérgenos con reacciones cruzadas, personal con roles por sucursal, y promociones multi-sucursal. El sistema de mesas implementa un workflow de cinco estados con animaciones visuales que reflejan el estado de cada orden en tiempo real.

Los componentes UI, organizados en veinticinco primitivos reutilizables con React.memo optimizado, logran una reducción del treinta y cinco por ciento en re-renderizados innecesarios. El patrón useFormModal y useConfirmDialog ha sido adoptado en nueve de once páginas con formularios, consolidando la gestión de estado modal en hooks reutilizables.

El Dashboard implementa sincronización multi-pestaña mediante BroadcastChannel para autenticación, garantizando que el logout en una pestaña se propague instantáneamente a todas las demás. Los cien tests Vitest cubren stores, hooks personalizados, y flujos críticos de negocio.

El sistema de mesas implementa un workflow de estados con animaciones CSS específicas. La animación animate-pulse-warning muestra pulso amarillo para

órdenes pendientes, animate-pulse-urgent pulso púrpura para cuenta solicitada, animate-status-blink parpadeo azul para cambios de estado, y animate-ready-kitchen-blink parpadeo naranja para el estado combinado de listo más cocina. Los WebSocket events se manejan mediante useTableWebSocket que actualiza el store con debounce para eventos de cambio de estado de mesa, evitando llamadas API duplicadas.

---

pwaMenu: Interfaz de Clientes

La aplicación pwaMenu representa la interfaz principal de interacción con clientes, diseñada como una Progressive Web App completamente funcional offline. La arquitectura de service workers con Workbox implementa estrategias de cache diferenciadas: CacheFirst para imágenes de productos con treinta días de retención, NetworkFirst con timeout de cinco segundos para APIs, y SPA fallback para navegación offline.

El sistema soporta internacionalización completa en español, inglés y portugués mediante i18next, con detección automática del idioma del navegador y persistencia de la preferencia del usuario. Cada error de validación y mensaje de la interfaz posee una clave i18n correspondiente en los tres archivos de traducción.

El flujo de usuario comienza cuando el cliente escanea un código QR en la mesa, lo que inicia una sesión vinculada al dispositivo mediante un UUID persistido en localStorage. Múltiples comensales

pueden unirse a la misma sesión usando códigos de cuatro dígitos, habilitando ordenamiento colaborativo donde cada comensal mantiene su propio carrito pero visualiza los pedidos de todos. El sistema de confirmación grupal, implementado mediante el componente RoundConfirmationPanel, requiere que todos los comensales aprueben antes de enviar la ronda a cocina.

La arquitectura de estado utiliza un tableStore modular dividido en cuatro archivos: store.ts con la definición Zustand y acciones, selectors.ts con hooks memoizados, helpers.ts con funciones puras para cálculos, y types.ts con interfaces TypeScript. Este patrón ha probado ser altamente mantenible y testeable.

El sistema de filtrado avanzado permite a los clientes excluir productos por alérgenos con detección de reacciones cruzadas, preferencias dietéticas como vegetariano, vegano, keto y bajo en sodio, y métodos de cocción. Estas preferencias se persisten como preferencias implícitas vinculadas al device_id, permitiendo que clientes recurrentes encuentren sus filtros ya aplicados mediante el hook useImplicitPreferences que sincroniza cambios al backend con debounce de dos segundos.

El carrito colaborativo implementa React diecinueve useOptimistic para actualizaciones optimistas con rollback automático. Cuando un comensal añade un ítem, la UI se actualiza instantáneamente mientras la operación se confirma en background. La confirmación grupal mediante RoundConfirmationPanel requiere que todos los comensales en la sesión confirmen antes del

envío, con timeout de cinco minutos y capacidad del proponente de cancelar.

---

pwaWaiter: Optimización del Flujo de Meseros

La aplicación pwaWaiter optimiza el flujo de trabajo de meseros mediante una interfaz móvil diseñada para condiciones de conectividad variable. La arquitectura implementa un sistema de cola de reintentos para operaciones fallidas, garantizando que ninguna acción se pierda incluso con conectividad intermitente.

Los meseros visualizan únicamente las mesas de los sectores asignados para el día actual, con la vista organizada en grupos por sector. Cada grupo muestra un encabezado con el nombre del sector, conteo de mesas, e indicadores de urgencia cuando hay órdenes pendientes o llamadas de servicio activas.

La funcionalidad Comanda Rápida permite a meseros tomar pedidos para clientes sin smartphone, utilizando un menú compacto sin imágenes optimizado para rendimiento. El componente AutogestionModal implementa una vista dividida con el catálogo a la izquierda y el carrito a la derecha, facilitando la selección rápida de productos.

El sistema de notificaciones push utiliza Web Push API para alertar a meseros cuando una ronda está lista para servir o cuando un cliente solicita atención. El servicio de notificaciones gestiona permisos, suscripción a push, y reproducción de sonidos de alerta.

La verificación de asignación de sucursal ocurre en dos fases: selección pre-login de sucursal desde el endpoint público de branches, seguida de verificación post-login mediante el endpoint de verificación de asignación. Este flujo garantiza que los meseros solo puedan acceder a sucursales donde están asignados para el día actual.

REST API: Núcleo de Lógica de Negocio

El REST API constituye el núcleo de procesamiento de lógica de negocio del sistema. Implementado con FastAPI, la arquitectura sigue estrictamente el patrón Clean Architecture con cuatro capas bien definidas que han sido refinadas a través de múltiples ciclos de refactorización.

Los routers se organizan en nueve grupos funcionales con dieciséis routers administrativos dedicados. Cada router actúa como controlador delgado que únicamente maneja concerns HTTP: parsing de parámetros mediante Pydantic, inyección de dependencias para autenticación, y construcción de respuestas. La lógica de negocio se delega íntegramente a servicios de dominio.

Los servicios de dominio, implementados en el directorio services/domain, encapsulan todas las reglas de negocio. Los diez servicios activos comprenden CategoryService, SubcategoryService, ProductService, AllergenService, BranchService, SectorService, TableService, StaffService,

PromotionService y TicketService. Todos heredan de clases base que proporcionan operaciones CRUD estándar con hooks de extensión para validación en creación, validación en actualización, acciones post-creación y acciones post-eliminación.

Los repositorios TenantRepository y BranchRepository abstraen completamente el acceso a datos, proporcionando métodos tipados con eager loading preconfigurado que previene el problema N+1. Los repositorios implementan filtrado automático por tenant_id y branch_id según corresponda, garantizando aislamiento multi-tenant a nivel de infraestructura.

Los modelos SQLAlchemy, organizados en veinte archivos por dominio, definen cincuenta y cuatro clases que heredan de AuditMixin para trazabilidad automática. Cada entidad registra quién la creó, modificó o eliminó, junto con timestamps precisos.

La estructura de routers se organiza por dominio de responsabilidad con separación clara entre operaciones públicas, autenticadas y administrativas. El directorio common contiene utilidades compartidas como dependencias comunes y paginación estandarizada. El directorio admin contiene dieciséis routers para operaciones CRUD administrativas. Los demás directorios incluyen auth para autenticación JWT con refresh HttpOnly, public para endpoints sin autenticación, tables para sesiones de mesa y flujo QR, content para catálogos, ingredientes, recetas y RAG, diner para operaciones de comensal con X-Table-Token, waiter para operaciones de mesero con filtrado por sector, kitchen para rounds y tickets de cocina, y billing para pagos y webhooks de Mercado Pago.

Servicios de Dominio y Sistema de Permisos

El sistema implementa diez servicios de dominio que heredan de clases base especializadas. BaseCRUDService proporciona operaciones CRUD estándar con hooks de extensión, mientras BranchScopedService añade filtrado automático por branch_id. Cada servicio recibe en su constructor la sesión de base de datos, el modelo a gestionar, el schema de salida y el nombre de entidad para mensajes de error en español.

Los servicios implementados cubren todos los dominios principales. CategoryService gestiona categorías con validación de unicidad de nombre por branch. SubcategoryService maneja subcategorías validando que la categoría padre exista. ProductService implementa gestión completa de BranchProduct, alérgenos, ingredientes y perfiles dietéticos. AllergenService administra alérgenos con manejo de reacciones cruzadas M:N. BranchService gestiona configuración de sucursal. SectorService valida pertenencia a branch. TableService genera códigos únicos alfanuméricos para mesas. StaffService maneja usuarios con roles por sucursal y restricciones de MANAGER. PromotionService gestiona branches e items asociados con validación de fechas. TicketService valida transiciones de estado de tickets de cocina.

El sistema de permisos utiliza Strategy Pattern con Interface Segregation Principle para manejar las diferencias entre roles de forma extensible. La clase

PermissionContext actúa como facade que simplifica el acceso a permisos, exponiendo propiedades como is_admin, is_management, tenant_id y branch_ids, junto con métodos para requerir acceso de gestión y verificar capacidades según acción, tipo de entidad y contexto.

Las estrategias implementadas cubren cada rol con sus permisos específicos. AdminStrategy proporciona acceso total sin restricciones. ManagerStrategy permite CRUD limitado a Staff, Tables, Allergens y Promotions en branches asignadas, pero sin capacidad de eliminación. KitchenStrategy solo permite lectura de productos y actualización de tickets y rounds. WaiterStrategy permite lectura de mesas del sector asignado y actualización de rounds.

Los mixins NoCreateMixin, NoDeleteMixin, NoUpdateMixin y BranchFilterMixin permiten composición flexible de comportamientos, siguiendo el principio de segregación de interfaces.

WebSocket Gateway: Comunicación en Tiempo Real

El WebSocket Gateway proporciona comunicación bidireccional en tiempo real entre el servidor y todos los clientes conectados. Con doce mil seiscientas cinco líneas de código organizadas en cincuenta y un archivos Python, representa el componente más complejo del sistema desde la perspectiva de concurrencia y resiliencia.

La arquitectura modular se organiza en dos capas principales. El directorio core contiene los módulos extraídos de los archivos monolíticos originales mediante la refactorización ARCH-MODULAR, donde connection_manager.py pasó de novecientas ochenta y siete a cuatrocientas noventa y cinco líneas, y redis_subscriber.py de seiscientas sesenta y seis a trescientas veintiséis líneas. El directorio components implementa la arquitectura de dominios con doce subdirectorios especializados.

El ConnectionManager orquesta el ciclo de vida de conexiones utilizando composición de módulos especializados. ConnectionLifecycle maneja registro y desregistro con locks apropiados. ConnectionBroadcaster implementa envío paralelo con worker pool de diez workers. ConnectionCleanup elimina conexiones muertas y stale. ConnectionStats agrega métricas de operación.

El RedisSubscriber procesa eventos mediante un sistema híbrido de Pub/Sub para eventos de tiempo real y Redis Streams para eventos críticos que requieren entrega garantizada. EventDropRateTracker monitorea tasas de descarte y emite alertas cuando superan el cinco por ciento. StreamConsumer implementa consumer groups con capacidad de rewind y dead letter queue para mensajes irrecuperables.

ConnectionIndex y Sistema de Locks

ConnectionIndex actúa como Value Object que mantiene todos los índices de conexiones y mappings inversos.

Los índices principales incluyen by_user, by_branch, by_session, by_sector, admins_by_branch y kitchen_by_branch. Los mappings inversos como ws_to_user, ws_to_tenant y demás permiten O(1) cleanup durante desconexiones.

LockManager implementa locks sharded para reducir contención en escenarios de alta concurrencia. El orden de adquisición de locks está estrictamente definido para prevenir deadlocks: primero connection_counter_lock como lock global, luego user_lock por usuario en orden ascendente de user_id, seguido de branch_locks por branch en orden ascendente de branch_id, y finalmente sector_lock, session_lock y dead_connections_lock como locks globales para operaciones específicas.

El componente LockSequence valida que este orden se respete y lanza DeadlockRiskError ante violaciones. Este context manager garantiza que los locks se adquieren siempre en el mismo orden, liberándose automáticamente en orden inverso al salir del contexto. Este patrón garantiza que dos coroutines que necesitan los mismos locks nunca se bloqueen mutuamente, ya que siempre los adquieren en el mismo orden.

Broadcast y Worker Pool

BroadcastRouter implementa Strategy Pattern con dos estrategias intercambiables. BatchBroadcastStrategy utiliza batches de tamaño fijo configurado típicamente en cincuenta conexiones.

AdaptiveBatchStrategy ajusta el tamaño según latencia observada, aumentando cuando la latencia es baja y reduciendo cuando aumenta. El patrón Observer permite registrar observadores de métricas sin acoplar la lógica de broadcast, implementado mediante MetricsObserverAdapter que notifica después de cada broadcast completado o cuando se aplica rate limiting.

ConnectionBroadcaster implementa un worker pool de diez workers asincrónicos que procesan tareas de envío desde una cola. Para broadcasts grandes con más de cincuenta conexiones, los envíos se distribuyen entre workers para procesamiento verdaderamente paralelo. Los futures permiten trackear completitud y agregar métricas. Este enfoque logra aproximadamente ciento sesenta milisegundos para broadcast a cuatrocientos usuarios versus cuatro mil milisegundos con envío secuencial.

Endpoints WebSocket y Mixins

La jerarquía de clases de endpoints implementa Template Method para el ciclo de vida y composición mediante mixins para comportamientos reutilizables. WebSocketEndpointBase es la clase abstracta que define el ciclo de vida mediante el método run() que orquesta autenticación, creación de contexto, registro de conexión, loop de mensajes y desregistro. Los métodos abstractos create_context, register_connection y handle_message deben ser implementados por subclases.

JWTWebSocketEndpoint hereda de la base y añade revalidación periódica de JWT cada cinco minutos para conexiones de larga duración. WaiterEndpoint extiende JWTWebSocketEndpoint con el comando especial refresh_sectors para actualizar asignaciones de sector. KitchenEndpoint recibe eventos de rounds y tickets. AdminEndpoint tiene acceso completo a todos los eventos de branch. DinerEndpoint usa TableToken en lugar de JWT para autenticación.

Los mixins disponibles proporcionan comportamientos composables. MessageValidationMixin valida tamaño de mensaje y rate limit. OriginValidationMixin valida el header Origin de la conexión. JWTRevalidationMixin implementa revalidación cada cinco minutos. HeartbeatMixin registra heartbeats para detección de conexiones stale. ConnectionLifecycleMixin proporciona logging estructurado del ciclo de vida de conexión.

Sistema de Eventos

El sistema define eventos tipados para cada flujo de negocio. El ciclo de vida de rondas progresa desde PENDING cuando el diner crea la orden hacia admin y waiters, a CONFIRMED cuando el waiter verifica hacia admin para que pueda enviar a cocina, a SUBMITTED cuando admin envía a cocina hacia admin y kitchen, a IN_KITCHEN cuando cocina comienza hacia todos incluyendo diners, a READY cuando cocina termina hacia todos, y finalmente SERVED cuando se entrega hacia todos. Los eventos CANCELED y

ROUND_ITEM_DELETED manejan cancelaciones y eliminación de items respectivamente.

Los eventos de carrito compartido incluyen CART_ITEM_ADDED cuando un diner agrega producto, CART_ITEM_UPDATED cuando cambia cantidad o notas, CART_ITEM_REMOVED cuando elimina item, CART_CLEARED cuando se envía la ronda, y CART_SYNC para reconexión con estado completo.

Los eventos de llamadas de servicio comprenden SERVICE_CALL_CREATED cuando el diner llama, SERVICE_CALL_ACKED cuando el waiter reconoce, y SERVICE_CALL_CLOSED cuando se atiende y cierra.

Los eventos de facturación incluyen CHECK_REQUESTED cuando el diner pide cuenta, CHECK_PAID cuando se completa el pago, PAYMENT_APPROVED cuando Mercado Pago aprueba, PAYMENT_REJECTED cuando rechaza, y PAYMENT_FAILED cuando hay error.

Los eventos de mesas son TABLE_SESSION_STARTED cuando se escanea QR, TABLE_CLEARED cuando se cierra sesión, y TABLE_STATUS_CHANGED para cambios de estado. Los eventos de cocina incluyen TICKET_IN_PROGRESS, TICKET_READY y TICKET_DELIVERED. Los eventos administrativos ENTITY_CREATED, ENTITY_UPDATED, ENTITY_DELETED y CASCADE_DELETE notifican operaciones CRUD a administradores.

El EventRouter determina destinatarios basándose en tipo de evento, branch_id, sector_id y session_id. Los eventos con sector_id se filtran para enviar solo a meseros asignados a ese sector. Los roles ADMIN y

MANAGER siempre reciben todos los eventos de su branch.

Resiliencia del WebSocket Gateway

CircuitBreaker implementa el patrón homónimo con tres estados para proteger contra fallos de Redis. En estado CLOSED el circuito opera normalmente. Después de cinco fallos consecutivos, transiciona a OPEN donde todas las llamadas fallan inmediatamente por treinta segundos. Luego pasa a HALF_OPEN donde permite hasta tres requests de prueba antes de decidir si cerrar completamente o reabrir.

El sistema de retry con jitter decorrelacionado previene thundering herd en reconexiones masivas. En lugar de exponential backoff puro que sincroniza retries, usa jitter decorrelacionado donde el delay es aleatorio entre el delay previo y el mínimo entre el delay máximo y tres veces el delay previo. Esto distribuye los reintentos en el tiempo evitando picos de carga.

StreamConsumer implementa Redis Streams para eventos críticos que requieren entrega garantizada. Consumer groups permiten que múltiples instancias del gateway compartan carga. PEL (Pending Entries List) recovery reclama mensajes pendientes después de treinta segundos de idle. Dead letter queue retiene mensajes irrecuperables después de tres intentos, almacenándolos en un stream dedicado con metadata del mensaje original, stream de origen, payload, conteo

de reintentos, timestamp de fallo y nombre del consumidor.

Scripts Lua para Operaciones Atómicas

El WebSocket Gateway utiliza Lua scripts ejecutados en Redis para garantizar atomicidad en operaciones críticas como rate limiting. El script de rate limiting recibe la clave de rate limit, máximo de mensajes permitidos, tamaño de ventana en segundos y timestamp actual, retornando si está permitido, conteo actual y TTL restante.

La implementación en Lua primero obtiene el conteo actual, verifica si excede el límite retornando cero si lo hace, incrementa el contador si está permitido, establece el TTL atómicamente si es el primer incremento de la ventana, y retorna uno con el nuevo conteo y TTL.

Las ventajas sobre implementación en Python son significativas. No hay race conditions entre GET e INCR ya que todo ocurre atómicamente en Redis. Solo se requiere un round-trip a Redis en lugar de múltiples. El TTL se establece atómicamente con el primer incremento. El script SHA se cachea para ejecución eficiente con EVALSHA en llamadas subsecuentes.

Modelo de Datos

El sistema define cincuenta y cuatro modelos SQLAlchemy organizados en veinte archivos por dominio coherente. El archivo base.py contiene Base y AuditMixin. El archivo tenant.py define Tenant y Branch. El archivo user.py contiene User y UserBranchRole. El archivo catalog.py define Category, Subcategory, Product y BranchProduct. El archivo allergen.py contiene Allergen, ProductAllergen y AllergenCrossReaction. El archivo ingredient.py define IngredientGroup, Ingredient, SubIngredient y ProductIngredient. El archivo product_profile.py contiene doce modelos de perfiles dietéticos, cocción, sabor y relaciones M:N.

El archivo sector.py define BranchSector y WaiterSectorAssignment. El archivo table.py contiene Table y TableSession. El archivo cart.py define CartItem para el carrito compartido. El archivo customer.py contiene Customer y Diner. El archivo order.py define Round y RoundItem. El archivo kitchen.py contiene KitchenTicket, KitchenTicketItem y ServiceCall. El archivo billing.py define Check usando la tabla app_check para evitar conflicto con palabra reservada SQL, junto con Payment, Charge y Allocation. El archivo knowledge.py contiene KnowledgeDocument y ChatLog para RAG. El archivo promotion.py define Promotion, PromotionBranch y PromotionItem. El archivo exclusion.py contiene BranchCategoryExclusion y BranchSubcategoryExclusion. El archivo recipe.py define Recipe y RecipeAllergen. El archivo audit.py contiene AuditLog. El archivo outbox.py define OutboxEvent para entrega garantizada de eventos.

Jerarquía de Entidades

La jerarquía de entidades refleja la estructura organizativa de un restaurante multi-sucursal. El Tenant representa el restaurante como entidad raíz. Cada Tenant tiene múltiples Branch que a su vez contienen Category con Subcategory y Product. Los productos tienen BranchProduct para precios por sucursal, ProductAllergen para alérgenos con tipo de presencia y nivel de riesgo, ProductIngredient para ingredientes, y relaciones M:N para métodos de cocción, sabores y texturas.

Cada Branch contiene BranchSector que agrupa Table. Las mesas tienen TableSession como sesión activa que contiene CartItem para el carrito compartido y Diner para comensales identificados por device_id. Cada Diner tiene CartItem propios y Round con confirmed_by_user_id para trackear verificación. Los Round contienen RoundItem que a su vez generan KitchenTicketItem.

Los sectores tienen WaiterSectorAssignment para asignaciones diarias de meseros. Las sesiones generan Check que contienen Charge, Payment y Allocation siguiendo el patrón FIFO. Las branches tienen KitchenTicket, ServiceCall y OutboxEvent para eventos pendientes.

Los User se relacionan M:N con Branch mediante UserBranchRole que define roles ADMIN, MANAGER, KITCHEN o WAITER. El Tenant también tiene catálogos scoped como CookingMethod, FlavorProfile,

TextureProfile y CuisineType, además de la jerarquía IngredientGroup, Ingredient y SubIngredient.

AuditMixin Universal

Todos los modelos heredan de AuditMixin, proporcionando trazabilidad completa de todas las operaciones. El mixin incluye is_active como flag de soft delete con índice para filtrado eficiente, deleted_at como timestamp de eliminación, y deleted_by_id junto con deleted_by_email para identificar quién eliminó.

Los timestamps created_at y updated_at se establecen automáticamente en inserción y actualización respectivamente. La trazabilidad de usuario se completa con created_by_id y created_by_email para el creador, y updated_by_id junto con updated_by_email para el último modificador.

Este diseño permite soft delete universal donde ningún dato se elimina físicamente, auditoría completa de quién y cuándo realizó cada operación, y restauración de entidades eliminadas con cascade a sus dependientes.

Relaciones y Constraints

El sistema de productos y precios por sucursal utiliza Product como maestro global mientras BranchProduct contiene el precio específico por

sucursal en centavos donde doce mil quinientos cincuenta representa ciento veinticinco dólares con cincuenta centavos, junto con disponibilidad.

El sistema de alérgenos con reacciones cruzadas usa ProductAllergen para registrar presencia con tres niveles CONTAINS, MAY_CONTAIN y TRACE, junto con riesgo HIGH, MEDIUM o LOW. AllergenCrossReaction implementa una relación auto-referencial M:N para modelar reacciones cruzadas entre alérgenos.

El flujo de órdenes sigue la cadena TableSession hacia Diner hacia Round hacia RoundItem hacia KitchenTicketItem, representando el flujo completo desde que un comensal se sienta hasta que su orden llega a cocina. Round ahora incluye confirmed_by_user_id para trackear qué mesero verificó el pedido.

Los constraints de integridad implementan UniqueConstraint en Category por branch_id y name, Subcategory por category_id y name, todos los catálogos tenant-scoped, y Round por table_session_id e idempotency_key. Los CheckConstraints validan que precios sean no negativos y que cantidades sean positivas.

Patrón Zustand Crítico para React 19

Todos los frontends implementan un patrón estricto de Zustand para evitar loops infinitos de re-renderizado causados por la detección de cambios más agresiva de React diecinueve. El destructuring directo del store

causa loops infinitos porque cada llamada retorna un nuevo objeto. El patrón correcto usa selectores individuales para cada valor o acción necesaria.

Es crítico utilizar referencias estables para arrays fallback, declarando una constante vacía del tipo correcto fuera del selector y usándola en lugar de crear un array vacío inline. Para selectores filtrados, se implementa memoización con cache simple que compara si el source cambió antes de recalcular el filtro.

Para selectores que reciben parámetros dinámicos como filtrado por ID, se usa un cache Map que almacena resultados por cada valor de parámetro. El hook useShallow de Zustand se usa para objetos pero no para arrays, permitiendo extraer múltiples propiedades sin causar re-renders innecesarios.

Los beneficios medidos incluyen reducción de más del noventa por ciento en re-renders de componentes con listas filtradas, eliminación de loops infinitos en React diecinueve strict mode, y estabilidad de referencias para useMemo y useCallback dependientes.

Sistema de Seguridad

El sistema implementa autenticación dual para diferentes contextos. JWT se usa para staff en Dashboard y pwaWaiter con access token de quince minutos de vida, refresh token de siete días almacenado en HttpOnly cookie siguiendo SEC-09, y claims que incluyen sub como user_id, tenant_id,

branch_ids, roles y email. Table Token se usa para diners en pwaMenu, firmado con HMAC-SHA256, con tres horas de vida reducido de ocho horas en CRIT-04, y claims que incluyen table_id, branch_id y session_id.

La revalidación de tokens en conexiones WebSocket verifica JWT cada cinco minutos contra la blacklist de Redis mediante CRIT-WS-01. Si el token fue revocado, la conexión se cierra con código cuatro mil uno. Los table tokens se revalidan cada treinta minutos en conexiones de diners mediante SEC-HIGH-01.

La revocación de tokens se implementa mediante Redis con TTL igual al tiempo restante del token. El patrón fail-closed trata errores de Redis como token blacklisted, garantizando que fallos de infraestructura no resulten en acceso no autorizado.

El rate limiting protege endpoints REST con cinco intentos por sesenta segundos para login implementado mediante script Lua atómico, y diez a veinte por minuto para endpoints de billing. WebSocket tiene límite de veinte mensajes por segundo por conexión y diez broadcasts globales por segundo, cerrando conexiones con código cuatro mil veintinueve cuando se excede.

La validación de entrada previene SSRF en URLs de imagen permitiendo solo esquemas HTTP y HTTPS, bloqueando hosts localhost, rangos IP privados y endpoints de metadata cloud. Previene SQL injection usando siempre bindings de SQLAlchemy y escapando caracteres especiales en patrones LIKE. Previene XSS mediante sanitización de HTML en inputs y headers CSP en producción.

El middleware de security headers añade X-Content-Type-Options nosniff, X-Frame-Options DENY, Content-Security-Policy restrictivo, y Strict-Transport-Security en producción.

Aislamiento Multi-Tenant

Cada restaurante constituye un tenant con aislamiento completo de datos. El tenant_id se propaga a través de todas las capas. En JWT claims el token contiene tenant_id del usuario. En repositorios TenantRepository auto-filtra por tenant_id en todas las queries. En el WebSocket Index las conexiones se indexan por tenant_id. En broadcast TenantFilter valida tenant antes de enviar cualquier evento.

El WebSocket Gateway implementa validación de tenant_id en múltiples puntos. El contexto puede tener tenant_id como None, distinto de cero que es ambiguo. Las conexiones sin tenant_id válido se registran con warning y no reciben eventos de sectores específicos.

La prevención de data leakage garantiza que eventos nunca se envíen a conexiones de otros tenants, queries siempre incluyan filtro por tenant_id, y logs saniticen PII antes de escritura.

Optimizaciones de Rendimiento

La configuración para cuatrocientos a seiscientos usuarios incluye ws_max_connections_per_user en tres para limitar conexiones duplicadas, ws_max_total_connections en mil como límite global, ws_message_rate_limit de veinte por segundo por conexión, ws_broadcast_batch_size de cincuenta para broadcast paralelo, redis_pool_max_connections de cincuenta para el pool async, redis_sync_pool_max_connections de veinte para rate limit y blacklist, y redis_event_queue_size de quinientos como buffer de backpressure.

El broadcast paralelo con worker pool logra aproximadamente ciento sesenta milisegundos para broadcast a cuatrocientos usuarios versus cuatro mil milisegundos con envío secuencial. Los sharded locks reducen contención en noventa por ciento mediante dicts de locks por branch_id y user_id, con locks globales solo para operaciones específicas.

El sector cache con TTL de sesenta segundos almacena asignaciones de sectores por user_id con límite de mil entradas, reduciendo queries de asignación en aproximadamente ochenta por ciento. El eager loading preconfigurado en repositorios usa selectinload y joinedload para prevenir queries N+1 en cadenas de relaciones.

Flujo de Orden Completo

El flujo comienza cuando el diner añade ítems al carrito en pwaMenu. La confirmación grupal requiere que todos los diners aprueben. El POST a la API crea

Round con status PENDING. El REST API publica ROUND_PENDING a Redis. El WS Gateway broadcast a admin y waiters del branch. El waiter verifica en mesa y confirma, transicionando a ROUND_CONFIRMED. El admin envía a cocina, transicionando a ROUND_SUBMITTED. Kitchen comienza y transiciona a ROUND_IN_KITCHEN, broadcast que ahora incluye diners. Kitchen termina y transiciona a ROUND_READY. Staff entrega y transiciona a ROUND_SERVED.

Flujo de Pago con Mercado Pago

El diner inicia pago en pwaMenu. El REST API crea preferencia en Mercado Pago y obtiene URL de checkout. El usuario es redirigido al checkout de MP donde procesa el pago. MP envía webhook al REST API que valida firma y actualiza Check. Se publica PAYMENT_APPROVED o PAYMENT_REJECTED. El WS Gateway notifica a diners de la sesión el resultado.

Flujo de Carrito Compartido

El sistema implementa sincronización en tiempo real del carrito entre múltiples dispositivos de comensales en la misma mesa. Cuando un diner en Chrome agrega Coca-Cola, el diner en Firefox lo ve instantáneamente.

El frontend ejecuta POST a la API de carrito con el item. El REST API valida producto y diner, ejecuta UPSERT en CartItem, incrementa cart_version en la

sesión, y usa BackgroundTask para publicar a Redis. El evento CART_ITEM_ADDED se publica al canal de la sesión. El WS Gateway rutea a diners de esa sesión mediante EventRouter. En cada frontend, el hook useCartSync recibe el evento, compara diner_id con el diner actual, y actualiza estado local si es de otro comensal, ignorando eventos propios ya actualizados optimistamente.

El modelo CartItem incluye tenant_id, branch_id, session_id como FK a TableSession, diner_id como FK a Diner identificando quién agregó, product_id como FK a Product, quantity validado entre uno y noventa y nueve, y notes opcional. El UniqueConstraint en session_id, diner_id y product_id permite UPSERT limpio.

Los eventos de carrito incluyen CART_ITEM_ADDED con payload completo de item_id, product_id, product_name, price_cents, quantity, diner_id, diner_name y diner_color. CART_ITEM_UPDATED tiene el mismo payload. CART_ITEM_REMOVED incluye item_id, product_id y diner_id. CART_CLEARED indica que la ronda fue enviada. CART_SYNC proporciona estado completo para reconexión.

Outbox Pattern para Entrega Garantizada

El sistema implementa el patrón Transactional Outbox para eventos críticos que no pueden perderse incluyendo billing, rounds y service calls. Los eventos se escriben atómicamente con los datos de negocio en la misma transacción de base de datos.

El endpoint ejecuta lógica de negocio como crear Round o Payment, llama a write_outbox_event con los parámetros del evento, y ejecuta db.commit() que guarda atómicamente los datos de negocio junto con el OutboxEvent en estado PENDING.

El Outbox Processor corre como loop que hace poll cada segundo. Selecciona eventos WHERE status = PENDING ORDER BY created_at. Actualiza status a PROCESSING usando FOR UPDATE SKIP LOCKED para evitar procesamiento duplicado. Publica a Redis. Actualiza status a PUBLISHED con processed_at. Si falla, incrementa retry_count hasta MAX_RETRIES de cinco, luego marca como FAILED.

El modelo OutboxEvent incluye tenant_id, event_type como ROUND_SUBMITTED o CHECK_PAID, aggregate_type como round o check o service_call, aggregate_id, payload como JSON serializado, status que puede ser PENDING, PROCESSING, PUBLISHED o FAILED, retry_count, last_error opcional, created_at y processed_at opcional.

Los eventos que usan Outbox por ser críticos incluyen ROUND_SUBMITTED y ROUND_READY para rounds, CHECK_REQUESTED, PAYMENT_APPROVED, PAYMENT_REJECTED y CHECK_PAID para checks, y SERVICE_CALL_CREATED para service calls. Los eventos no críticos como CART_*, TABLE_* y ENTITY_* usan publicación directa con BackgroundTasks.

Infraestructura Docker

El archivo docker-compose.yml orquesta cinco servicios. El servicio db usa la imagen pgvector/pgvector:pg16 en el puerto cinco mil cuatrocientos treinta y dos con volumen persistente pgdata y healthcheck mediante pg_isready.

El servicio redis usa la imagen redis:7-alpine en el puerto seis mil trescientos ochenta mapeado desde seis mil trescientos setenta y nueve para evitar conflictos, ejecutando redis-server con appendonly yes, maxmemory de doscientos cincuenta y seis megabytes y política allkeys-lru, con healthcheck mediante redis-cli ping.

El servicio backend construye desde el directorio padre, expone el puerto ocho mil, depende de db y redis, monta el directorio backend como volumen de solo lectura para hot reload, y ejecuta uvicorn con reload habilitado.

El servicio ws_gateway similar al backend expone el puerto ocho mil uno, depende de db, redis y backend, y monta tanto backend como ws_gateway como volúmenes de solo lectura.

El servicio pgadmin usa la imagen dpage/pgadmin4 en el puerto cinco mil cincuenta para administración web de la base de datos.

Health Checks y Métricas

El REST API expone health en modo sync y health/detailed en modo async con estado de Redis. El WS Gateway expone ws/health en modo sync y ws/health/detailed en modo async. Los health checks detallados incluyen estado de pool Redis async, estado de pool Redis sync, conexiones activas y métricas del subscriber.

El endpoint de métricas Prometheus en ws/metrics expone wsgateway_connections_total para conexiones activas, wsgateway_connections_rejected_total con etiqueta reason para auth o rate_limit, wsgateway_broadcasts_total y wsgateway_broadcasts_failed_total para estadísticas de broadcast, y wsgateway_redis_reconnects_total junto con wsgateway_event_drops_total para monitoreo de Redis.

El sistema usa logging estructurado con formato JSON en producción y coloreado en desarrollo. PII se sanitiza antes de logging con emails parcialmente ocultos, JTI truncados y user_ids hasheados.

EventDropRateTracker monitorea tasas de descarte con ventana de sesenta segundos, umbral de alerta del cinco por ciento y cooldown de alerta de cinco minutos para evitar spam de logs.

Decisiones Arquitectónicas

La adopción de Clean Architecture con servicios de dominio separados añade complejidad inicial pero ha demostrado valor en testing unitario de lógica de

negocio sin dependencias de infraestructura, evolución independiente de capas, y onboarding más rápido de nuevos desarrolladores. El trade-off es aceptado dado el tamaño y complejidad del sistema.

El sistema usa híbrido de Redis Streams y Pub/Sub. Pub/Sub maneja eventos de tiempo real que pueden perderse sin consecuencias graves como TABLE_STATUS_CHANGED. Streams manejan eventos críticos que requieren entrega garantizada como ROUND_* y PAYMENT_*. Streams permiten rewind si el gateway reinicia, previniendo pérdida de órdenes.

Para broadcasts grandes el worker pool ofrece backpressure mediante queue con límite, métricas granulares de latencia, y graceful shutdown con drain timeout. asyncio.gather se mantiene para broadcasts pequeños por simplicidad.

La adopción de React diecinueve con Compiler en Dashboard elimina la mayoría de React.memo manuales. Sin embargo se mantienen en componentes críticos para compatibilidad si el compilador se desactiva y explicitación de intención de optimización.

La persistencia en localStorage permite funcionamiento offline, sesiones que sobreviven refrescos de página, y menor carga al servidor. El trade-off es sincronización más compleja entre tabs, resuelto con BroadcastChannel.

Estadísticas del Código

El componente Backend rest_api comprende ochenta y un archivos con aproximadamente quince mil líneas. El componente Backend shared tiene treinta archivos con aproximadamente cuatro mil quinientas líneas. El WebSocket Gateway tiene cincuenta y un archivos con doce mil seiscientas cinco líneas. El Dashboard tiene más de ochenta y cinco archivos con aproximadamente doce mil líneas. El componente pwaMenu tiene más de setenta archivos con aproximadamente diez mil líneas. El componente pwaWaiter tiene más de cuarenta archivos con aproximadamente seis mil líneas. El total supera trescientos sesenta archivos con aproximadamente sesenta mil líneas de código.

Documento generado: Febrero 2026
Versión: 3.0 - Reescrito en prosa narrativa