**UNIVERSIDAD ABIERTA INTERAMERICANA**

**FACULTAD DE TECNOLOGÍA INFORMÁTICA**

**ASIGNATURA: PROGRAMACIÓN WEB**

**Programación Web**

**Proyecto de desarrollo de sistema de turnos para peluquerías.**

**Alumnos:**

Luparia Guillermo Raul

Zacharczuk Mauro Andres

**Profesor**

Escandell Gustavo

**Buenos Aires - Argentina**

**Año 2025**

**ÍNDICE**

# Introducción

El presente documento describe el desarrollo e implementación de un sistema de turnos para una peluquería, elaborado como proyecto de desarrollo web. El sistema busca facilitar la gestión de servicios, personal y reservas, optimizando el tiempo tanto de los clientes como del personal administrativo. Este trabajo contempla tanto el backend desarrollado en Python con Flask como el frontend implementado con React.

# Tecnologías Utilizadas

Las tecnologías empleadas en este proyecto son:

* - Backend: Python
* - Frontend: React.js, SCSS
* - Base de datos: SQL Server (entorno Azure)
* - Control de versiones y despliegue: Git, Docker, Visual Studio Code

# Backend

El backend está desarrollado en Python. Está compuesto por múltiples módulos organizados en carpetas, donde cada una contiene servicios, repositorios y modelos. Se utilizan controladores (routes) para manejar las solicitudes HTTP. Los modelos representan las entidades principales como Usuario, Persona, Servicio y Turno. El sistema incluye autenticación mediante tokens JWT, control de roles (ADMINISTRADOR, EMPLEADO, CLIENTE), gestión de turnos, servicios, empleados y estados de turnos. El backend expone endpoints RESTful que permiten crear, listar, actualizar y eliminar entidades, además de realizar reservas de turnos y consultar disponibilidad.

# Módulos incluídos:

# Autenticación (auth)

# auth.py

Maneja el inicio de sesión y generación de tokens JWT.

# decorators.py

`decorators.py` define funciones decoradoras personalizadas para asegurar la autenticación y autorización en las rutas del backend.

- `token\_required(f)`: Verifica que el encabezado HTTP contenga un token JWT válido. Si el token está vencido o ausente, devuelve error 401.

- `role\_required(role)`: Verifica que el usuario autenticado posea un rol específico (ej. ADMINISTRADOR) antes de ejecutar la función.

- `admin\_required`: Decorador especializado que exige el rol "ADMINISTRADOR".

Estas funciones son esenciales para proteger los endpoints críticos y se aplican en todas las rutas del sistema (turnos, pagos, servicios, etc.).

# Configuración (config)

# config/base.py

`base.py` define la clase `BaseConfig`, que contiene la configuración común a todos los entornos.

- Usa `dotenv` para cargar las variables de entorno desde un archivo `.env`.

- Atributos configurables:

- `SECRET\_KEY`: Clave utilizada para firmar tokens JWT y sesiones.

- `DB\_USER`, `DB\_PASSWORD`, `DB\_SERVER`, `DB\_NAME`, `DB\_PORT`: Datos de conexión a la base de datos.

- Si alguna de las variables requeridas falta, lanza una excepción `ValueError`.

Esta configuración es la base para los entornos de desarrollo y producción, y garantiza seguridad mediante parámetros externos.

# config/dev.py

Define la clase `DevConfig`, que hereda de `BaseConfig`.

- Activa los modos `DEBUG = True` y `TESTING = True`, lo que permite ver errores detallados y usar herramientas de desarrollo.

Esta configuración se usa cuando el entorno necesita feedback inmediato, como durante pruebas locales.

# config/prod.py

Define `ProdConfig`, que también hereda de `BaseConfig`.

- Establece `DEBUG = False` y `TESTING = False`, lo cual es adecuado para despliegues en producción, donde se busca estabilidad y seguridad.

Este archivo permite cambiar rápidamente entre entornos, sin modificar el resto del sistema.

# Modelos (models)

# cliente.py

`cliente.py` define la clase `Cliente`, que extiende de `Persona` utilizando herencia polimórfica en SQLAlchemy.

- Atributos heredados: nombre, apellido, email, etc.

- Relación:

- `telefonos`: Lista de teléfonos asociados al cliente.

- `turnos`: Lista de turnos tomados por el cliente.

- Métodos:

- `\_\_init\_\_`: Establece automáticamente `tipo\_persona` como CLIENTE.

- `nombre\_completo`: Propiedad que devuelve el nombre y apellido concatenado.

Esta clase permite distinguir a los clientes de otras personas del sistema, como empleados o administradores.

# empleado.py

`empleado.py` define la clase `Empleado`, también derivada de `Persona`.

- Relaciones:

- `servicios`: Relación muchos-a-muchos con `Servicio` a través de la tabla `SERVICIO\_EMPLEADO`.

- `servicio\_empleado\_rel`: Relación directa con los registros intermedios.

- `turnos`: Lista de turnos asignados a este empleado.

- Usa `polymorphic\_identity` para distinguir el tipo EMPLEADO dentro de la jerarquía de personas.

La clase representa a los peluqueros, barberos o asistentes disponibles para brindar servicios.

# pago.py

`pago.py` contiene el modelo `Pago`, que representa los registros de pagos realizados.

- Campos importantes:

- `MONTO`: Monto abonado.

- `FECHA\_PAGO`: Fecha efectiva de pago.

- `COMPROBANTE`: Código o referencia del comprobante.

- `METODO\_ID`: Clave foránea a `MetodoPago`.

- `ESTADO\_ID`: Estado del pago según `EstadoPago`.

- Relación con `Turno`: Un pago puede estar vinculado a uno o varios turnos mediante la relación `turnos`.

Incluye trazabilidad mediante campos de alta y baja, y permite almacenar datos del medio de pago con flexibilidad.

# persona.py

Este archivo define la clase base `Persona`, de la cual heredan `Cliente` y `Empleado`.

- Atributos clave:

- `nombre`, `apellido`, `email`: Datos básicos del individuo.

- `tipo\_persona`: Enum que define si es CLIENTE, EMPLEADO o ADMINISTRADOR.

- `tipo\_empleado`: Campo opcional para empleados (barbero, peluquero, asistente).

- Campos administrativos como `fecha\_alta`, `usuario\_alta`, `fecha\_baja`.

- Relaciones posibles con `Usuario`, `Telefono`, `Turno`.

La arquitectura basada en herencia facilita la reutilización de lógica común entre tipos de persona.

# servicio.py

El modelo `Servicio` representa los distintos tratamientos o actividades que ofrece la peluquería.

- Atributos:

- `nombre`, `descripcion`: Identificadores del servicio.

- `precio`: Costo del servicio.

- `duracion\_estimada`: Duración en minutos (default 30).

- `tipo\_servicio`: Enum que categoriza (ej. corte, tintura, barbería).

- Relaciones:

- `empleados`: Relación muchos-a-muchos con `Empleado`.

- `servicio\_empleado\_rel`: Relación directa con tabla intermedia.

El modelo está diseñado para admitir ABM completo y soporta estadísticas de uso, asignación y reserva.

# servicioEmpleado.py

Este archivo define el modelo `ServicioEmpleado`, que representa la relación entre empleados y servicios.

- Atributos:

- `SERVICIO\_ID`: Clave foránea hacia la tabla `SERVICIO`.

- `EMPLEADO\_ID`: Clave foránea hacia la tabla `PERSONA`, usada aquí como empleado.

- Ambos campos conforman una clave primaria compuesta.

- Relaciones:

- `servicio`: Relación con el modelo `Servicio`.

- `empleado`: Relación con el modelo `Empleado`.

- `serialize(include\_details=False)`: Método que permite exportar el objeto como diccionario. Si se indica `include\_details=True`, agrega datos anidados del servicio y empleado.

Este modelo permite determinar qué empleados están habilitados para realizar cada servicio, y forma parte esencial del flujo de asignación en el backend.

# telefono.py

`telefono.py` define el modelo `Telefono`, asociado a un cliente específico.

- Atributos:

- `NUMERO`: Teléfono registrado.

- `TIPO`: Enum que indica si es celular, fijo, etc.

- `CLIENTE\_ID`: Clave foránea hacia `Persona`, filtrada por tipo CLIENTE.

- Relaciones:

- `cliente`: Relación inversa hacia `Cliente`.

- Métodos:

- `\_\_repr\_\_`: Representación textual para depuración.

- `serialize`: Exporta la entidad como diccionario.

Permite que cada cliente tenga múltiples números de contacto y facilita la gestión desde el frontend.

# turno.py

El archivo `turno.py` contiene el modelo `Turno`, el cual representa las reservas realizadas por los clientes.

- Atributos:

- `ID`: Identificador único del turno.

- `FECHA\_HORA`: Fecha y hora de inicio del turno.

- `DURACION`: Duración en minutos del servicio (por defecto 30).

- `ESTADO`: Estado actual del turno, definido por el enum `EstadoTurno` (ej. PENDIENTE, CANCELADO).

- `NOTAS`: Comentarios adicionales opcionales.

- `CLIENTE\_ID`, `EMPLEADO\_ID`: Relaciones a personas con roles correspondientes.

- `SERVICIO\_ID`: Servicio que se va a prestar.

- Relaciones:

- Utiliza `ForeignKey` para vincularse con las tablas `PERSONA` y `SERVICIO`.

- Es probable que haya una relación con `Pago` en otros modelos, para enlazar turnos con sus pagos correspondientes.

Este modelo es utilizado intensamente por `turnoService.py` y los repositorios relacionados, ya que representa el núcleo de la funcionalidad

de la reserva de turnos del sistema.

# usuario.py

define el modelo `Usuario`, que representa a los usuarios del sistema en la base de datos.

- Atributos:

- `usuario`: Nombre de usuario (clave primaria).

- `contrasena`: Contraseña encriptada.

- `activo`: Indica si el usuario está habilitado para iniciar sesión.

- `rol`: Rol de acceso (ej. ADMINISTRADOR, CLIENTE).

- `persona\_id`: Relación con una persona en la tabla `PERSONA`.

- `fecha\_alta`, `fecha\_baja`, `usuario\_alta`, `usuario\_baja`: Información administrativa para auditoría.

- Relaciones:

- `persona`: Relación bidireccional con el modelo `Persona`.

- Método especial `\_\_repr\_\_()`: Devuelve una representación string útil para debugging o logs.

Este modelo es clave para el sistema de autenticación y autorización, ya que permite asociar una identidad a cada persona del sistema.

Es utilizado por `authService.py`, `usuarioRepository.py` y la lógica de login.

# Enumeraciones (models/enumerations)

# estadoPagoEnum.py

Enum para los estados del pago (pendiente, pagado, etc).

# estadoTurnoEnum.py

Enum para el estado del turno (reservado, finalizado, etc).

# medioPagoEnum.py

Enum de medios de pago (efectivo, tarjeta, etc).

# tipoEmpleadoEnum.py

Enum que define roles como PELUQUERO, BARBERO, ASISTENTE.

# tipoPersonaEnum.py

Enum que clasifica personas como CLIENTE, EMPLEADO, ADMINISTRADOR.

# tipoServicioEnum.py

Enum para categorizar los servicios.

# tipoTelefonoEnum.py

Enum para categorizar los tipos de teléfono.

# Rutas (routes)

# authRoute.py

Define las rutas para autenticación y login.

# clienteRoute.py

`clienteRoute.py` define la ruta `/get` del blueprint `cliente\_bp`, encargado de devolver la lista de clientes registrados.

- `get\_clientes()`: Recupera los clientes mediante `ClienteService.listar\_clientes()`, los serializa y retorna en una respuesta JSON.

- Aplica `@token\_required` para asegurar que solo usuarios autenticados puedan acceder.

Este endpoint es clave para las interfaces administrativas que gestionan clientes y se conecta directamente con la base de datos a través del servicio.

# empleadoRoute.py

`empleadoRoute.py` define rutas asociadas a empleados mediante el blueprint `empleado\_bp`.

- `crear\_empleado()`: Recibe un JSON desde el cuerpo de la petición, valida su presencia y delega la creación al `EmpleadoService`. En caso de éxito, retorna un mensaje JSON con el nombre del empleado creado.

- Aplica `@token\_required` para protección.

La ruta permite incorporar nuevos empleados al sistema, incluyendo su asignación posterior a servicios disponibles.

# metodoPagoRoute.py

Define la ruta para obtener los métodos de pago disponibles, sin necesidad de autenticación.

- `listar\_metodos\_pago()`: Llama al `MetodoPagoService` para obtener los métodos (efectivo, tarjeta, etc.) y los devuelve como una lista JSON.

Este archivo se relaciona directamente con el módulo de pagos y es utilizado, por ejemplo, al momento de cargar un formulario de pago en el frontend.

# pagoRoute.py

Este archivo define múltiples rutas para acceder a los pagos realizados, agrupadas bajo el blueprint `pago\_bp`.

- `obtener\_pago(pago\_id)`: Devuelve un pago específico consultando al `PagoService` y lo serializa en JSON.

- `obtener\_pago\_por\_turno(turno\_id)`: Permite consultar el pago asociado a un turno específico.

- Ambos endpoints están protegidos por `@token\_required`.

Estas rutas son esenciales para recuperar información financiera vinculada a turnos y clientes, tanto desde el panel administrativo como del cliente.

# servicioRoute.py

Define el blueprint de Flask para gestionar los endpoints HTTP asociados a los servicios.

- `listar\_servicios()`: Maneja la ruta GET para consultar servicios. Permite filtrar por activos mediante un parámetro query. Utiliza decoradores como `@token\_required` para exigir autenticación y `@cross\_origin` para habilitar CORS. La respuesta es un JSON con la lista de servicios serializados.

El archivo define el blueprint `servicio\_bp`, que se registra en el `app.py`. Interactúa con `ServicioService` para obtener los datos, y controla errores con `APIError`. Esta arquitectura separa claramente la capa de presentación de la lógica de negocio.

# turnoRoute.py

`turnoRoute.py` define los endpoints HTTP relacionados a los turnos, utilizando Flask y decoradores de autenticación.

- `listar\_turnos()`: Ruta GET que permite obtener turnos aplicando filtros opcionales como cliente\_id, empleado\_id, estado y fecha. Utiliza el servicio `TurnoService` para ejecutar la lógica. Se protege con `@token\_required`.

- Otros endpoints (no visibles en este fragmento) probablemente incluyan la creación de turnos, modificación de estado o cancelación.

Este archivo conecta directamente con `turnoService.py` y `pagoService.py`, estructurando las respuestas en formato JSON. Facilita la interacción cliente-servidor desde el frontend.

# Servicios (services)

# authService.py

El archivo `authService.py` contiene la clase `AuthService`, responsable de la autenticación de usuarios en el sistema.

Utiliza JWT para emitir tokens de acceso y emplea el algoritmo `check\_password\_hash` para validar contraseñas encriptadas.

- `login(usuario, contrasena)`: Verifica que los datos no sean nulos. Luego busca en la base de datos un usuario con el nombre recibido.

Si el usuario no existe o la contraseña es incorrecta, lanza una excepción de API personalizada. Si todo es válido, genera un token JWT

con la información del usuario y fecha de expiración.

Este archivo se comunica directamente con el repositorio de usuarios (`usuarioRepository`) y depende del modelo `Usuario`.

También utiliza la clase `Config` para acceder a la clave secreta del token. Está diseñado para ser consumido desde el endpoint `authRoute.py`.

# clienteService.py

`clienteService.py` define la clase `ClienteService`, encargada de la lógica relacionada a la gestión de clientes.

- `listar\_clientes()`: Recupera todos los clientes registrados usando el repositorio correspondiente. Maneja errores con excepciones personalizadas.

- `crear\_cliente(data)`: Valida que se proporcionen los datos necesarios. Verifica la presencia de campos obligatorios como nombre, apellido, email y DNI. Si falta alguno, lanza un error. Una vez validados, probablemente persiste el nuevo cliente en la base de datos.

Este servicio interactúa directamente con `clienteRepository.py` y se expone mediante rutas como las de `clienteRoute.py`. La validación estricta y manejo de errores mejora la robustez del sistema.

# empleadoService.py

`empleadoService.py` implementa la clase `EmpleadoService`, encargada de procesar la lógica específica para empleados.

- `listar\_empleados()`: Recupera todos los empleados desde `EmpleadoRepository`, registrando en el log cuántos se encontraron. Captura errores y lanza excepciones API.

- `crear\_empleado(data)`: Aunque no se visualiza completo, se anticipa que implementa validaciones similares al cliente y crea un objeto empleado asociado a una persona existente.

Este servicio es invocado por las rutas que manejan empleados, y se conecta con los modelos para validar reglas de negocio como tipos de empleado o relaciones con servicios.

# servicioService.py

`servicioService.py` contiene la clase `ServicioService`, que provee métodos para gestionar los servicios ofrecidos por la peluquería.

- `obtener\_servicio\_por\_id(servicio\_id)`: Consulta el repositorio `ServicioRepository` y devuelve un objeto `Servicio` si existe.

Registra en el log si el servicio no se encuentra.

- Otros métodos que pueden estar presentes incluyen: `crear\_servicio`, `modificar\_servicio` y `eliminar\_servicio`, aunque su presencia debe

confirmarse en el código completo. Normalmente estarían encargados de validar parámetros, evitar duplicados, y realizar operaciones de ABM.

Este servicio centraliza la interacción entre entidades de negocio (como `ServicioEmpleado`) y la capa de persistencia.

Se utiliza desde `servicioRoute.py`, donde se exponen los endpoints de administración de servicios.

# turnoService.py

El archivo `turnoService.py` define la clase `TurnoService`, que centraliza toda la lógica de negocio relacionada con los turnos.

- `crear\_turno\_reserva(cliente\_id, empleado\_id, servicio\_id, fecha\_hora, notas)`: Crea un nuevo turno validando la disponibilidad del horario

para el empleado seleccionado, y asociándolo al cliente, servicio y notas especificadas. Verifica que el turno no se superponga con otros.

La clase también maneja la relación con los pagos, ya que en algunos casos el turno puede registrarse con un estado de pago automático.

Utiliza los repositorios de empleado, turno, servicio y pago para verificar consistencia de datos. Incluye logging para trazabilidad.

El servicio es utilizado por el archivo `turnoRoute.py` y está preparado para emitir excepciones personalizadas si ocurre algún conflicto.

# pagoService.py

`pagoService.py` contiene la clase `PagoService`, que gestiona los pagos realizados por los clientes.

- `obtener\_pago\_por\_id(pago\_id)`: Busca un pago específico por ID. Si no existe, lanza una advertencia en logs y devuelve error.

- Además, es probable que contenga métodos como `registrar\_pago()` y `confirmar\_pago()` que permiten asociar un pago a un turno, cambiar su estado o emitir comprobantes.

Utiliza `PagoRepository` para acceder a los pagos y `TurnoRepository` para vincular el pago con un turno existente. También recurre a enums como `EstadoPago` para controlar el estado del pago (pendiente, aprobado, etc.). Esta clase es fundamental para cerrar el flujo de reserva y pago de turnos.

# Repositorios (repositories)

# clienteRepository.py

Acceso a datos de clientes en base de datos.

# empleadoRepository.py

`empleadoRepository.py` define la clase `EmpleadoRepository`, utilizada para acceder a los datos de empleados desde la base de datos.

- `get\_all()`: Consulta todos los empleados registrados en la tabla correspondiente. Si ocurre un error durante la consulta, registra los detalles completos del error incluyendo el tipo, mensaje y stack trace. También indica qué consulta se ejecutó, lo cual es útil para depuración avanzada.

Este repositorio es utilizado por `EmpleadoService` y forma parte esencial de la lógica de visualización y administración de empleados.

.

# metodoPagoRepository.py

Este archivo implementa la clase `MetodoPagoRepository`, que permite consultar los métodos de pago disponibles.

- `get\_all()`: Devuelve todos los métodos de pago definidos, utilizando `db.session.query`.

- `get\_by\_id(metodo\_id)`: Recupera un método de pago específico por su ID.

Este repositorio es utilizado por el `MetodoPagoService` y sus datos son expuestos mediante `metodoPagoRoute.py`, permitiendo que el frontend cargue dinámicamente las opciones disponibles.

# pagoRepository.py

`pagoRepository.py` implementa funciones para gestionar pagos en la base de datos.

- `get\_by\_id(pago\_id)`: Recupera un objeto `Pago` por su ID.

- `create(\*\*kwargs)`: Crea un nuevo objeto de pago desde un diccionario de parámetros, lo inserta en la base y confirma. Maneja errores con rollback y lanza `APIError`.

- `update()`: Confirma cualquier cambio pendiente a un objeto pago existente.

Este repositorio es esencial en el flujo de pagos asociado a turnos, y es consumido desde `pagoService.py`. Se destaca por su manejo robusto de errores y uso del modelo `Pago`.

# servicioEmpleadoRepository.py

Este repositorio gestiona la relación muchos-a-muchos entre servicios y empleados.

- `asignar\_empleado(servicio\_id, empleado\_id, usuario\_alta=None)`: Inserta un nuevo registro en la tabla intermedia `SERVICIO\_EMPLEADO`, asociando un servicio a un empleado. Registra el usuario que realizó la asignación. En caso de error, se realiza rollback y se lanza una excepción.

Esta lógica es crítica para definir qué empleados están habilitados para ofrecer qué servicios. El repositorio se comunica con el modelo `servicioEmpleado` y es utilizado por `servicioService.py`.

# servicioRepository.py

Consulta y persistencia de servicios.

# turnoRepository.py

`turnoRepository.py` es responsable del acceso a la base de datos para la entidad `Turno`.

- `get\_all()`: Devuelve todos los turnos registrados.

- `get\_by\_id(turno\_id)`: Obtiene un turno específico por su ID.

- `get\_by\_empleado\_fecha(empleado\_id, fecha)`: Retorna todos los turnos de un empleado desde una fecha determinada, excluyendo los cancelados.

- `create(turno)`: Inserta un nuevo turno en la base de datos y confirma la transacción.

- `update()`: Realiza un commit de los cambios pendientes.

El repositorio encapsula la lógica de consulta SQLAlchemy y es utilizado principalmente desde `turnoService.py`. Al centralizar el acceso a datos, se mantiene la separación de responsabilidades.

# usuarioRepository.py

`usuarioRepository.py` provee funciones de acceso directo a la base de datos para la entidad `Usuario`.

- `obtener\_por\_usuario(usuario)`: Busca un usuario específico por su nombre.

- `crear\_usuario(usuario)`: Inserta un nuevo usuario en la base de datos y realiza commit de la transacción.

- `existe\_usuario(usuario)`: Verifica si ya existe un usuario con ese nombre, útil para validaciones previas al registro.

Este repositorio es utilizado por `authService.py` y posiblemente `usuarioService.py`. Centraliza todas las interacciones con la tabla de usuarios, permitiendo aislar la lógica de acceso a datos del resto de la aplicación.

# Aplicación principal

# app.py

`app.py` es el punto de entrada principal del backend. Contiene la función que crea e inicializa la aplicación Flask.

- Importa y configura extensiones como SQLAlchemy (`db`) y CORS.

- Carga la configuración desde `Config`, que puede ser `DevConfig` o `ProdConfig`.

- `check\_database\_connectivity(server)`: Verifica que el servidor de base de datos esté accesible mediante un intento de conexión de bajo nivel por socket.

- Configura el logger con rotación de archivos (`RotatingFileHandler`).

- Registra las rutas importadas desde los distintos módulos de `routes/`.

- También maneja errores globales y excepciones con `register\_error\_handlers`.

Este archivo centraliza la inicialización de la aplicación y sus dependencias, siendo el núcleo del backend Flask.

# extensions.py

Este archivo define las extensiones de Flask que serán utilizadas a lo largo del proyecto.

- `db = SQLAlchemy()`: Crea una instancia global de SQLAlchemy, usada para definir y consultar los modelos de base de datos.

- Esta instancia se inicializa luego dentro de `app.py` mediante `db.init\_app(app)`.

Mantener las extensiones separadas del archivo principal ayuda a mantener una arquitectura desacoplada y facilita pruebas unitarias y reutilización.

# Excepciones

# exceptions.py

Define excepciones personalizadas utilizadas en servicios.

# Pruebas

# cliente\_test.py

Test unitarios para el módulo cliente.

# 4. Frontend

El frontend fue implementado con React. Posee un diseño responsive con hojas de estilo en SCSS. Cuenta con componentes reutilizables, formularios de validación, navegación con menú general y layout consistente. Se incluye un sistema de autenticación que bloquea al usuario tras tres intentos fallidos y mantiene la sesión activa mediante almacenamiento local de tokens. El sistema ofrece diferentes vistas según el rol del usuario: cliente, empleado o administrador.

# index.js

Este archivo es el punto de entrada de la aplicación React. Renderiza el componente `<App />` dentro del elemento HTML raíz (`#root`). Se encarga también de envolver la aplicación con los `Provider` necesarios para manejar Redux y otras configuraciones globales.

# App.js

`App.js` es el componente principal que contiene el layout general de la aplicación. Dentro de él se cargan las rutas definidas en `routes.jsx`. También puede incluir validaciones globales, carga de temas o inicializaciones comunes como la detección de idioma o usuario logueado.

# routes.jsx

Este archivo define todas las rutas de navegación de la aplicación usando React Router. Asocia URLs específicas con los componentes correspondientes. Por ejemplo, `/login` carga el componente de inicio de sesión, `/turnos` muestra la página de reserva de turnos, etc. Puede incluir rutas protegidas usando `PrivateRoute`.

# store.js

Configura el store de Redux para la aplicación. Importa todos los reducers combinados y los aplica como middleware al store, permitiendo manejar el estado global compartido entre múltiples componentes. Es una pieza central en la arquitectura React + Redux del sistema.

# Layout General (layout/)

Dentro de `layout/` se encuentran los componentes que conforman la estructura visual permanente del sitio:  
- `header.jsx`: Barra superior que incluye navegación, logo, usuario activo, etc.  
- `footer.jsx`: Pie de página con información general.  
Ambos usan SCSS propios para definir su apariencia. Se integran en el `App.js` para permanecer visibles en toda la navegación.

# Componentes Compartidos (shared/components/)

Contiene componentes reutilizables que no dependen de una página específica:  
- `global-loader.component.jsx`: Muestra un indicador de carga animado.  
- `toast-notifier.component.jsx`: Sistema de notificaciones emergentes para feedback visual (éxito, error, etc.).  
  
Estos componentes son invocados desde diferentes páginas o procesos asincrónicos como peticiones API.

# Sistema de Rutas Protegidas

El archivo `private-route.js` implementa una ruta protegida que impide el acceso a páginas sin estar autenticado. Se basa en la validación de tokens y redirecciona a `/login` si el usuario no está autorizado.

# Módulo de Autenticación y Registro de Usuarios

# login.component.jsx

Este componente React muestra el formulario de inicio de sesión. Utiliza `useDispatch` para enviar la acción `login` y `useSelector` para obtener el estado de autenticación desde Redux.  
  
- Renderiza un formulario con campos `usuario` y `contraseña`.  
- Muestra mensajes de error o spinner de carga dependiendo del estado.  
- Si el login es exitoso, redirecciona automáticamente a la página principal del sistema.  
  
Estéticamente está estilizado con `login.scss` y se conecta con Redux mediante las acciones del `auth.reducer.js`.

# verify-token.js

Este archivo exporta la función `verifyToken(token)` que valida un token JWT.  
  
- Si no hay token o si está vencido, retorna `false`.  
- Si el token es válido, verifica su fecha de expiración (campo `exp`) contra la hora actual.  
  
Es utilizado en el sistema de rutas protegidas (`PrivateRoute`) y durante la carga inicial para mantener la sesión activa del usuario.

# auth.reducer.js

Este archivo implementa un slice de Redux llamado `auth`, utilizando `createSlice` de Redux Toolkit.  
  
- Estado inicial: guarda si el usuario está autenticado, el token, el usuario, errores y estados de carga.  
- `login`: acción asíncrona que realiza una llamada a la API (`/auth/login`), guarda el token en localStorage y actualiza el estado.  
- `register`: registra un nuevo usuario y también guarda la información en Redux.  
  
Además incluye funciones auxiliares como `verifyInitialToken()` para verificar si el token guardado en localStorage sigue vigente.

# registro.component.jsx

Componente React que muestra un formulario para registrar un nuevo usuario en el sistema.  
  
- Campos: nombre, apellido, email, DNI, usuario y contraseña.  
- Usa `useDispatch` para lanzar la acción `register` definida en `auth.reducer.js`.  
- Si la operación es exitosa, muestra un mensaje y permite redirigir al login.  
  
El formulario está estilizado mediante `register.scss` y estructurado con `reactstrap`. También maneja validaciones básicas y errores visuales.

# Módulo de Turnos, Servicios y Pagos

# reserva-turno.component.jsx

Este componente implementa la lógica para reservar un turno. Está conectado a Redux y utiliza múltiples hooks como `useEffect`, `useDispatch` y `useSelector`.  
  
- Muestra un formulario dividido en pasos: selección de servicio, empleado, horario y método de pago.  
- Llama a `fetchServicioById`, `fetchEmpleadosByServicio` y `fetchDisponibilidad` para cargar datos dinámicos.  
- Al confirmar el turno, lanza `crearReserva` y abre el componente `ModalPago`.  
  
Este componente es clave en el flujo del cliente, ya que gestiona toda la interacción necesaria para concretar una reserva completa.

# servicio-list.component.jsx

Muestra una lista visual de todos los servicios disponibles. Cada servicio se representa con una tarjeta que incluye una imagen, nombre, descripción y botón para solicitar turno.  
  
- Utiliza imágenes temáticas como `servicio-corte.png`, `servicio-barberia.png`, etc.  
- Llama a `fetchServicios()` y `fetchServiciosByEmpleado()` para poblar el estado.  
- Incluye manejo de carga (`Spinner`) y errores (`Alert`).  
  
Es una de las primeras páginas que ve el cliente y permite iniciar el proceso de reserva.

# pago-modal.component.jsx

Componente de tipo modal que se activa durante el proceso de reserva de turno. Permite seleccionar el método de pago y añadir notas.  
  
- Props principales:  
 - `metodosPago`: Lista de opciones (efectivo, tarjeta, etc.).  
 - `monto`, `empleado`, `servicio`: Información contextual mostrada.  
 - `onConfirmarPago`: Función ejecutada al confirmar.  
  
El modal utiliza `reactstrap` para su diseño y valida que se seleccione una opción antes de continuar. Se integra perfectamente al flujo de reserva como paso final.

# reserva-turno.component.jsx

Este componente gestiona la reserva de turnos por parte del cliente. Usa Redux para obtener servicios, empleados y disponibilidad de horarios.  
- Despliega formularios para seleccionar servicio, empleado, fecha y horario.  
- Muestra las franjas horarias disponibles, basadas en la disponibilidad consultada desde el backend.  
- Incluye validaciones y conexión con `ModalPago` para pagar el turno reservado.  
  
Utiliza acciones como `fetchEmpleadosByServicio`, `fetchDisponibilidad`, `crearReserva`, y `fetchMetodosPago`. Es uno de los componentes más importantes del frontend.

# servicio-list.component.jsx

Muestra una galería de servicios ofrecidos (corte, tintura, barbería) con imágenes, nombres, descripciones y botón para reservar.  
  
- Usa Redux para cargar los servicios desde el backend mediante `fetchServicios`.  
- Al hacer clic en "Reservar", redirige al componente de reserva de turno pasando el ID del servicio.  
- Incluye diseño visual atractivo con imágenes y estilos definidos en `servicio.scss`.  
  
Este componente es la puerta de entrada para que los clientes elijan el tipo de atención que desean reservar.

# pago-modal.component.jsx

Este componente muestra un modal para confirmar el pago del turno. Se integra con el componente de reserva y permite seleccionar el método de pago y agregar notas.  
  
- Props esperadas:  
 - `metodosPago`, `monto`, `empleado`, `servicio`, `onConfirmarPago`  
- El usuario selecciona un método de pago y al confirmar se dispara el evento `onConfirmarPago`, que se comunica con el backend.  
  
La estructura está basada en `reactstrap` y permite confirmar o cancelar el pago con botones en el pie del modal.

# Otras Funcionalidades del Frontend

# PersonasPage.jsx

Componente para administración de personas. Permite consultar y modificar el tipo de persona (CLIENTE, EMPLEADO, ADMINISTRADOR) y en caso de ser empleado, también el tipo de empleado.  
  
- Usa `fetch` para consultar datos desde la API.  
- Permite modificar campos como `tipo\_persona` o `tipo\_empleado` y enviarlos nuevamente al backend.  
- Muestra mensajes de éxito o error según el resultado de las operaciones.  
  
Es utilizado por administradores para convertir usuarios registrados en empleados del sistema.

# usuario.component.jsx

Permite administrar usuarios del sistema.  
  
- Consulta la lista de usuarios desde la API.  
- Permite cambiar roles, bloquear/desbloquear usuarios o reiniciar contraseñas.  
- Decodifica el token JWT para identificar si el usuario actual tiene privilegios de administrador.  
  
Es clave para el mantenimiento del sistema por parte de personal administrativo.

# contrasena.component.jsx

Formulario para que el usuario autenticado cambie su contraseña actual.  
  
- Compara si la nueva contraseña coincide con su confirmación.  
- Envia los datos al backend y muestra un mensaje según la respuesta.  
- Usa estilos de `ChangePasswordPage.scss`.  
  
Funcionalidad estándar para mantener segura la cuenta del usuario.

# Register.jsx

Componente alternativo de registro de usuario.  
  
- Permite ingresar nombre de usuario, email y contraseña.  
- Valida que ambas contraseñas coincidan antes de enviar el formulario.  
- Muestra errores en pantalla si ocurren problemas al registrarse.  
  
Puede utilizarse en casos donde se permite el auto-registro sin intervención administrativa.

# Estilos SCSS

El sistema utiliza SCSS para mantener un diseño moderno y consistente. Algunos de los estilos clave son:  
  
- `AuthForm.scss` y `register.scss`: Estilos para login y registro, con fondos, espaciado, formularios centrados y botones estilizados.  
- `servicio.scss`: Galería de servicios con diseño de tarjetas, imágenes de fondo, efectos hover y botones.  
- `turno.scss`: Estilos aplicados a la página de reserva de turnos, incluyendo formulario paso a paso y resaltado de horarios.  
  
Los estilos están organizados y reutilizados para asegurar coherencia visual en todas las páginas.

# 5. Funcionalidades Principales

* - Registro y login de usuarios
* - Gestión de usuarios, servicios y personas (ABM)
* - Sistema de turnos con disponibilidad por fecha y empleado
* - Visualización y filtrado de horarios disponibles
* - Confirmación y resumen del turno reservado
* - Estética unificada con colores suaves, tarjetas, botones estilizados

# 6. Flujo de Reserva de Turnos

El cliente, una vez logueado, accede a la página 'Pedir Turno'. Allí selecciona el servicio, un empleado, una fecha y una hora disponible. El sistema filtra horarios válidos según los turnos ya registrados para ese empleado y fecha. Al confirmar, se guarda el turno en el backend y se muestra una tarjeta de resumen con los datos del turno y el monto a abonar por el servicio. Genera el flujo de pagos para abonar el servicio.

# 7. Estética del Sistema

El sistema posee una interfaz moderna y clara. Se utilizan tarjetas para mostrar datos importantes, formularios con campos espaciados, botones estilizados y una paleta de colores coherente. Las páginas de registro, login, gestión y turnos mantienen un estilo homogéneo. El uso de DatePicker y selectores mejora la experiencia del usuario.

# 8. Conclusiones

Este sistema logra cubrir de forma integral las necesidades de una peluquería moderna: gestión de usuarios, empleados, servicios y turnos. La integración de tecnologías modernas y la división clara entre frontend y backend facilitan su mantenimiento y escalabilidad. El sistema fue desarrollado siguiendo los principios de separación de responsabilidades, donde el backend se encarga de la lógica de negocio, autenticación y persistencia de datos, y el frontend se responsabiliza de la interacción con el usuario final.

Cada componente fue diseñado con el objetivo de ser reutilizable y fácilmente escalable. Las páginas en React están organizadas por responsabilidad, y se emplearon componentes de terceros para mejorar la experiencia del usuario. Además, se utilizó SCSS para lograr un diseño moderno y coherente entre pantallas.

Los formularios en el sistema manejan validaciones básicas desde el frontend y se comunican con el backend usando `fetch`, incluyendo token cuando es necesario para endpoints protegidos.

El sistema fue probado localmente en ambiente de desarrollo utilizando Docker y ejecutándose en puertos separados: 3000 para frontend y 8080 para backend. La comunicación entre servicios se habilitó mediante CORS.

La estética general se basa en bloques con márgenes, botones estilizados, formularios centrados y feedback inmediato en pantalla. La estructura modular permite añadir fácilmente nuevas funcionalidades.