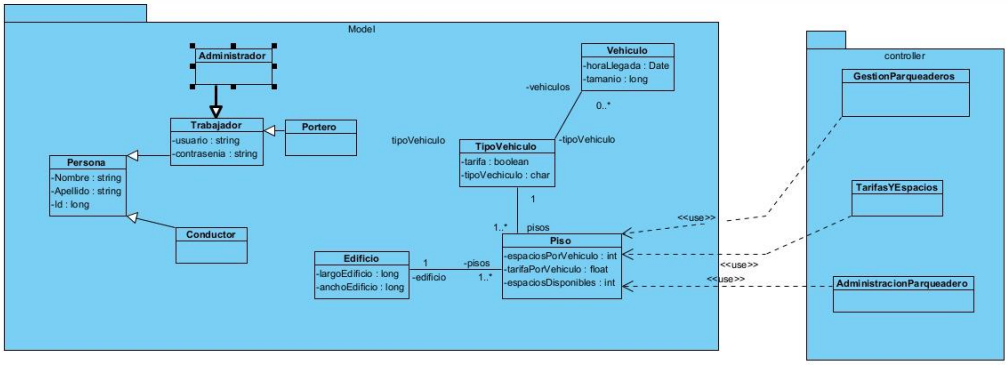
Entrega 1
Desarrollo Web

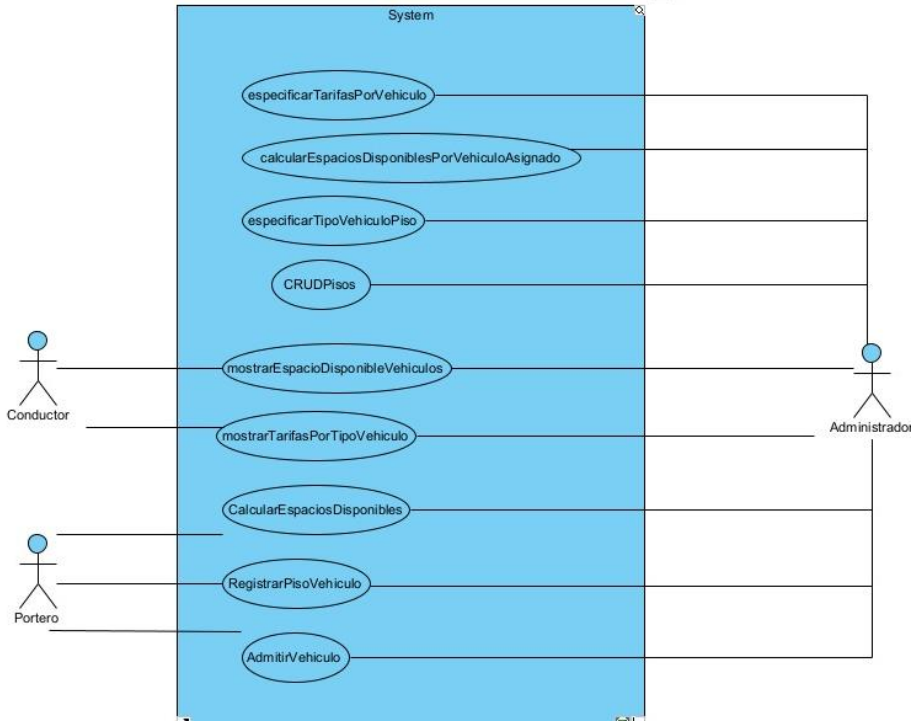
Valentina Rozo Gonzalez
Daniel Stiven Florido
Gabriel Alejandro Martín Sánchez

Septiembre 2023

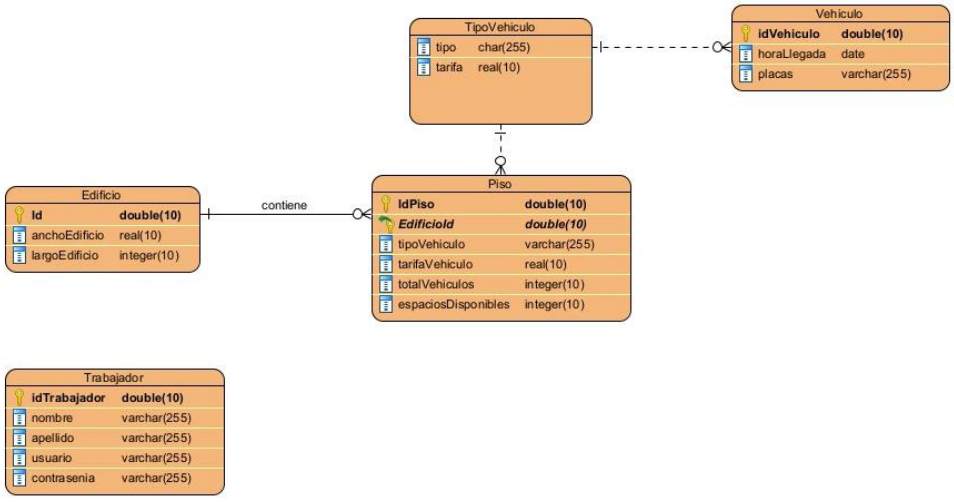
1. Diagrama de clases



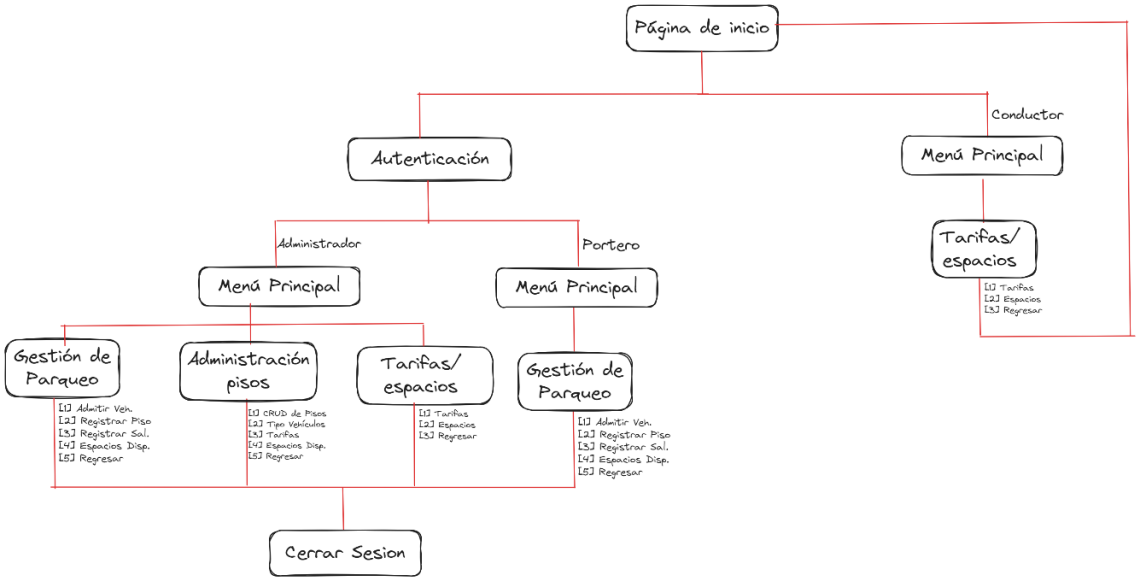
2. Diagrama de Usos



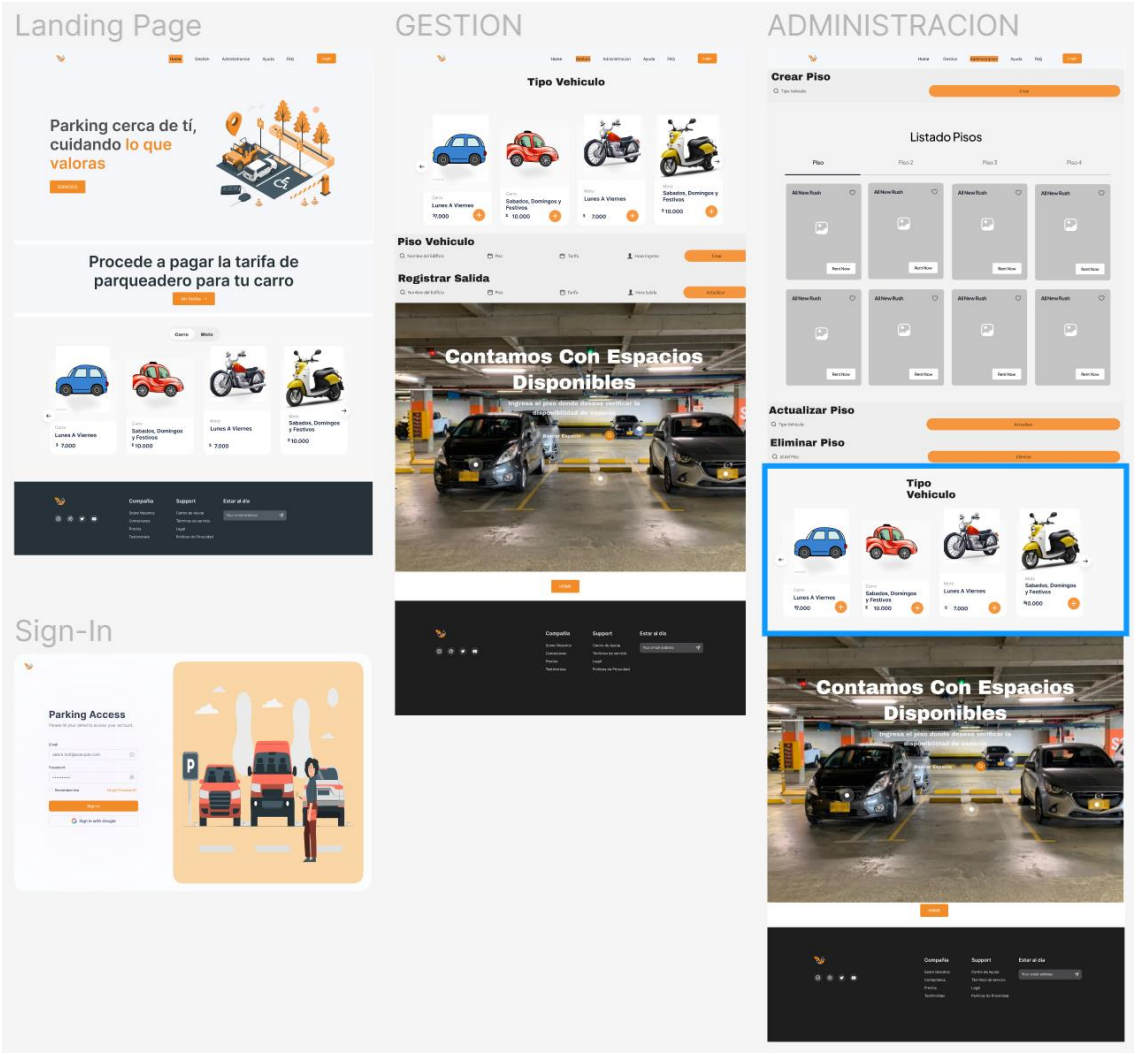
3. Diagrama de Relación



4. Diagrama de Navegación



5. Mockups Aplicación Web



6. **Documentación código utilizado**

Video Youtube: https://youtu.be/cRP8cM_WSFY

- **Controlador TipoVehiculoController**

Este controlador es parte esencial de tu aplicación web de gestión de parqueaderos y se encarga de gestionar la información relacionada con los tipos de vehículos. Los métodos definidos en el controlador manejan las operaciones CRUD (Crear, Leer, Actualizar, Borrar) para los tipos de vehículos y gestionan las vistas correspondientes para interactuar con el usuario.

Anotaciones

- `@Controller`: Esta anotación marca la clase como un controlador de Spring MVC, lo que significa que manejará solicitudes HTTP entrantes.
- `@RequestMapping("/parkingApp/tipoVehiculo") : Esta anotación especifica la ruta base para todas las rutas URL manejadas por este controlador. Todas las rutas URL deben comenzar con "/parkingApp/tipoVehiculo".

Inyeccion de Dependencias

- `@Autowired private TipoVehiculoService tipoVehiculoService;`: Esto inyecta una instancia de `TipoVehiculoService` en el controlador, lo que te permite utilizar los métodos y servicios proporcionados por `TipoVehiculoService` para interactuar con la lógica de negocio relacionada con los tipos de vehículos.

Logger

- `Logger log = LoggerFactory.getLogger(getClass());`: Crea una instancia de `Logger` para habilitar el registro de información y seguimiento (logging) en el controlador. Esto es útil para registrar eventos y depurar la aplicación.

Métodos

- `recuperarTipoVehiculo(@PathVariable Long idTipoVehiculo)`: Este método maneja las solicitudes GET en la ruta "/parkingApp/tipoVehiculo/view/{idTipoVehiculo}". Recupera un objeto `TipoVehiculo` basado en el ID proporcionado y lo pasa a una vista llamada "tipoVehiculo-view" utilizando `ModelAndView`.
- `listarTipoVehiculo()`: Este método maneja las solicitudes GET en la ruta "/parkingApp/tipoVehiculo/list". Recupera una lista de objetos `TipoVehiculo` y la pasa a una vista llamada "tipoVehiculo-list" utilizando `ModelAndView`.
- `editarPiso(@PathVariable Long idTipoVehiculo)`: Este método maneja las solicitudes GET en la ruta "/parkingApp/tipoVehiculo/edit/{idTipoVehiculo}". Al igual que el primer método, recupera un objeto `TipoVehiculo` basado en el ID proporcionado y lo pasa a una vista llamada "tipoVehiculo-edit" utilizando `ModelAndView`.
- `guardarTipoVehiculo(@ModelAttribute TipoVehiculo tipoVehiculo)`: Este método maneja las solicitudes POST en la ruta "/parkingApp/tipoVehiculo/save". Recibe un objeto `TipoVehiculo` enviado desde un formulario web y lo guarda utilizando el servicio `tipoVehiculoService`. Luego, redirige al usuario a la lista de tipos de vehículos.
- `mostrarTipoVehiculoNuevo(Model model)`: Este método maneja las solicitudes GET en la ruta "/parkingApp/tipoVehiculo/create". Prepara un nuevo objeto `TipoVehiculo` y lo agrega al modelo, luego devuelve la vista "tipoVehiculo-create".

- `borrarTipoVehiculo(@PathVariable Long tipoVehiculo)`: Este método maneja las solicitudes GET en la ruta "/parkingApp/tipoVehiculo/delete/{idTipoVehiculo}". Recibe el ID de un tipo de vehículo que se va a borrar y llama al servicio `tipoVehiculoService` para realizar la eliminación. Luego, redirige al usuario a la lista de tipos de vehículos y muestra un mensaje de éxito o error en función del resultado.

- **Controlador `PisoController`**
  Este controlador es una parte esencial de tu aplicación web de gestión de parqueaderos y se encarga de gestionar la información relacionada con los pisos. Los métodos definidos en el controlador manejan las operaciones CRUD (Crear, Leer, Actualizar, Borrar) para los pisos y gestionan las vistas correspondientes para interactuar con el usuario.
  Anotaciones
  - `@Controller`: Esta anotación marca la clase como un controlador de Spring MVC, lo que significa que manejará solicitudes HTTP entrantes.
  - `@RequestMapping("/parkingApp/piso")`: Esta anotación especifica la ruta base para todas las rutas URL manejadas por este controlador. Todas las rutas URL deben comenzar con "/parkingApp/piso".
  Inyección de Dependencias
  - `@Autowired private PisoService pisoService;`: Esto inyecta una instancia de `PisoService` en el controlador, lo que te permite utilizar los métodos y servicios proporcionados por `PisoService` para interactuar con la lógica de negocio relacionada con los pisos.
  Logger
  - `Logger log = LoggerFactory.getLogger(getClass());`: Crea una instancia de `Logger` para habilitar el registro de información y seguimiento (logging) en el controlador. Esto es útil para registrar eventos y depurar la aplicación
  Métodos
  - `recuperarPiso(@PathVariable Long idPiso)`: Este método maneja las solicitudes GET en la ruta "/parkingApp/piso/view/{idPiso}". Recupera un objeto `Piso` basado en el ID proporcionado y lo pasa a una vista llamada "piso-view" utilizando `ModelAndView`.
  - `listarPisos()`: Este método maneja las solicitudes GET en la ruta "/parkingApp/piso/list". Recupera una lista de objetos `Piso` y la pasa a una vista llamada "piso-list" utilizando `ModelAndView`.
  - `editarPiso(@PathVariable Long idPiso)`: Este método maneja las solicitudes GET en la ruta "/parkingApp/piso/edit/{idPiso}". Al igual que el primer método, recupera un objeto `Piso` basado en el ID proporcionado y lo pasa a una vista llamada "piso-edit" utilizando `ModelAndView`.
  - `guardarPiso(@ModelAttribute Piso piso)`: Este método maneja las solicitudes POST en la ruta "/parkingApp/piso/save". Recibe un objeto `Piso` enviado desde un formulario web y lo guarda utilizando el servicio `pisoService`. Luego, redirige al usuario a la lista de pisos.
  - `mostrarPisoNuevo(Model model)`: Este método maneja las solicitudes GET en la ruta "/parkingApp/piso/create". Prepara un nuevo objeto `Piso` y lo agrega al modelo, luego devuelve la vista "piso-create".

- `borrarPiso(@PathVariable Long idPiso)`: Este método maneja las solicitudes GET en la ruta "/parkingApp/piso/delete/{idPiso}". Recibe el ID de un piso que se va a borrar y llama al servicio `pisoService` para realizar la eliminación. Luego, redirige al usuario a la lista de pisos y muestra un mensaje de éxito o error en función del resultado.

- **Controlador `homeController`**
  El controlador `homeController` está diseñado para redirigir a los usuarios a la página de administración de pisos (presumiblemente relacionada con la gestión de pisos en un parqueadero) cuando acceden a la página de inicio de la aplicación ("/parkingApp"). La vista "admin-pisos" será cargada y mostrada al usuario como respuesta a esta solicitud. Este controlador sirve como punto de entrada o página de inicio de la aplicación web.
  Anotaciones
  - `@Controller`: Esta anotación marca la clase como un controlador de Spring MVC, lo que significa que manejará solicitudes HTTP entrantes.
  Métodos
  - `home()`: Este método maneja las solicitudes GET en la ruta "/parkingApp". Cuando un usuario accede a la ruta de inicio de la aplicación, este método se ejecuta. Dentro de este método, se devuelve una cadena "admin-pisos", que representa el nombre de la vista que se mostrará como respuesta.

- **Clase Edificio**
  Esta clase representa la entidad "Edificio" en tu sistema. Esta entidad es fundamental para la gestión de espacios de parqueadero en la aplicación.

  Un edificio se asocia con un parqueadero y tiene propiedades como su identificador único (`id`), ancho (`ancho`), largo (`largo`), y nombre (`name`). Además, esta clase tiene una relación `@OneToMany` con la clase `Piso`, lo que significa que un edificio puede tener varios pisos.
  Atributos
  - `id`: Representa el identificador único del edificio. Se genera automáticamente mediante una estrategia de generación de secuencia (`GenerationType.SEQUENCE`).
  - `ancho`: Representa el ancho del edificio. Es un valor numérico de tipo `long`.
  - `largo`: Representa el largo del edificio. Es un valor numérico de tipo `long`.
  - `name`: Representa el nombre del edificio. Es una cadena de texto (`String`).
  Relación
  `pisos`: Esta relación indica que un edificio puede tener varios pisos. La anotación `@OneToMany` se utiliza para mapear esta relación bidireccional con la clase `Piso`. El atributo `mappedBy` especifica que la relación está mapeada por la propiedad `edificio` en la clase `Piso`.
  Constructores
  `Edificio()`: Constructor por defecto sin argumentos. Se utiliza en la creación de instancias de la clase.

`Edificio(String name, long ancho, long largo)`: Constructor que acepta el nombre, ancho y largo del edificio como argumentos para inicializar sus propiedades.

Métodos Getter y Setter

La clase proporciona métodos getter y setter para acceder y modificar sus atributos, como `getId()`, `getAncho()`, `getLargo()`, `getName()`, `setId()`, `setAncho()`, `setLargo()`, y `setName()`.

- **Clase Piso**

Esta clase representa la entidad "Piso" en tu sistema de parqueadero. Un piso es una parte del edificio y tiene propiedades como su identificador único (`id`), el edificio al que pertenece (`edificio`), el número total de vehículos que puede albergar (`totalVehiculos`), la cantidad de espacios disponibles (`espaciosDisponibles`), y el tipo de vehículo que puede estacionarse en el piso (`tipoVehiculo`). Esta entidad es esencial para la gestión de los espacios de estacionamiento en tu aplicación de parqueadero.

Atributos

`id`: Representa el identificador único del piso. Se genera automáticamente mediante una estrategia de generación de secuencia (`GenerationType.SEQUENCE`).

`edificio`: Representa la relación `@ManyToOne` con la clase `Edificio`, lo que significa que un piso pertenece a un edificio específico. La anotación `@JoinColumn` se utiliza para mapear esta relación y el atributo `name` indica la columna en la tabla de la base de datos que almacena la referencia al edificio.

`totalVehiculos`: Representa el número total de vehículos que puede albergar el piso. Es un valor numérico de tipo `int`.

`espaciosDisponibles`: Representa la cantidad de espacios disponibles en el piso para estacionar vehículos. También es un valor numérico de tipo `int`.

`tipoVehiculo`: Representa la relación `@ManyToOne` con la clase `TipoVehiculo`, lo que indica el tipo de vehículo que puede estacionarse en el piso. La anotación `@JoinColumn` se utiliza para mapear esta relación y el atributo `name` indica la columna en la tabla de la base de datos que almacena la referencia al tipo de vehículo.

Constructores

`Piso(Edificio edificio, TipoVehiculo tipoVehiculo)`: Constructor que acepta un edificio y un tipo de vehículo como argumentos para inicializar las propiedades `edificio` y `tipoVehiculo`.

`Piso()`: Constructor por defecto sin argumentos. Se utiliza en la creación de instancias de la clase.

Métodos Getter

La clase proporciona métodos getter para acceder a sus atributos, como `getId()`, `getEdificio()`, `getTipoVehiculo()`, `getEspaciosDisponibles()`, y `getTotalVehiculos()`.

Método `toString()`

La clase sobrescribe el método `toString()` para proporcionar una representación de cadena de un objeto `Piso`, que incluye información sobre el número total de vehículos, espacios disponibles y el tipo de vehículo asociado.

- **Clase TipoVehiculo**

Esta clase representa la entidad "TipoVehiculo" en tu sistema de parqueadero. Un tipo de vehículo tiene propiedades como su identificador único (`id`), el tipo de vehículo (`tipo`), y la tarifa asociada a ese tipo de vehículo (`tarifa`). Almacena información sobre el tipo de

vehículo y la tarifa correspondiente, lo que es esencial para calcular el costo de estacionamiento y gestionar las tarifas para diferentes tipos de vehículos en tu aplicación de parqueadero.

Atributos

- `id`: Representa el identificador único del tipo de vehículo. Se genera automáticamente mediante una estrategia de generación de secuencia (`GenerationType.SEQUENCE`).
- `tipo`: Representa el tipo de vehículo, que se almacena como un carácter (`char`). Por ejemplo, puede ser 'C' para automóviles, 'M' para motocicletas, etc.
- `tarifa`: Representa la tarifa asociada a ese tipo de vehículo, que es un valor numérico de tipo `double`. Esta tarifa indica el costo de estacionamiento por unidad de tiempo para este tipo de vehículo.

Constructores

- `TipoVehiculo(char tipo, double tarifa): Constructor que acepta el tipo de vehículo y la tarifa como argumentos para inicializar las propiedades `tipo` y `tarifa`.
- `TipoVehiculo()`: Constructor por defecto sin argumentos. Se utiliza en la creación de instancias de la clase.

Métodos Getter

La clase proporciona métodos getter para acceder a sus atributos, como `getId()`, `getTipoPlaca()`, y `getTarifa()`.

Método `toString()`

La clase sobrescribe el método `toString()` para proporcionar una representación de cadena de un objeto `TipoVehiculo`, que incluye información sobre el tipo de vehículo y su tarifa asociada.

- **Clase `Vehiculo`**

La clase `Vehiculo` representa un vehículo estacionado en tu sistema de parqueadero. Cada instancia de esta clase almacena información sobre un vehículo, incluyendo su identificador único (`id`), tipo de vehículo (`tipoVehiculo`), hora de llegada (`horaLlegada`), y placa (`placa`).. Esta clase es fundamental para el seguimiento y la gestión de vehículos dentro de tu aplicación de parqueadero.

Atributos

- `id`: Es el identificador único del vehículo y se genera automáticamente utilizando una estrategia de generación de secuencia (`GenerationType.SEQUENCE`).
- `tipoVehiculo`: Representa el tipo de vehículo al que pertenece el vehículo estacionado. Esta propiedad está relacionada con la clase `TipoVehiculo` mediante una relación `@ManyToOne`. Permite identificar el tipo de vehículo al que pertenece este vehículo.
- `horaLlegada`: Indica la hora de llegada del vehículo al parqueadero. Se almacena como un valor entero (`int`) y es obligatorio (no nulo).
- `placa`: Almacena la placa del vehículo. La placa es un identificador único para vehículos y se almacena como una cadena de caracteres (`String`).

Constructores

- `Vehiculo(TipoVehiculo tipoVehiculo, String placa, int horaLlegada)`: Constructor que acepta un tipo de vehículo, una placa y la hora de llegada para inicializar las propiedades del vehículo.
- `Vehiculo()`: Constructor por defecto sin argumentos. Se utiliza en la creación de instancias de la clase.

Métodos Getter y Setter

La clase proporciona métodos getter y setter para acceder y modificar sus atributos, como `getId()`, `getTipoVehiculo()`, `getPlaca()`, `getHoraLlegada()`, `setTipoVehiculo()`, `setPlaca()`, y `setHoraLlegada()`.

- **Interfaz EdificioRepository**

Es un repositorio de Spring Data JPA que se utiliza para acceder y administrar entidades de tipo `Edificio` en la base de datos. Spring Data JPA proporciona métodos predefinidos para realizar operaciones comunes en la base de datos, y también permite definir consultas personalizadas mediante anotaciones o consultas JPQL (Java Persistence Query Language).

Métodos predefinidos

La interfaz `EdificioRepository` hereda de `JpaRepository<Edificio, Long>`, lo que significa que hereda una serie de métodos predefinidos para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en entidades de tipo `Edificio`. Algunos de los métodos predefinidos incluyen:

- `save(Edificio edificio)`: Guarda un objeto `Edificio` en la base de datos.
- `findById(Long id)`: Busca un `Edificio` por su identificador único (`id`).
- `findAll()`: Recupera todos los objetos `Edificio` de la base de datos.
- `delete(Edificio edificio)`: Elimina un objeto `Edificio` de la base de datos.

Métodos personalizados

Además de los métodos predefinidos, la interfaz `EdificioRepository` define algunos métodos personalizados mediante consultas derivadas de nombres y consultas JPQL personalizadas:

- `List<Edificio> findAllByName(String name)`: Busca todos los edificios cuyo nombre coincida exactamente con el valor proporcionado.
- `List<Edificio> findAllByNameStartingWith(String name)`: Busca todos los edificios cuyos nombres comiencen con el valor proporcionado.
- `List<Edificio> findAllByNameStartingWithIgnoreCase(String name)`: Busca todos los edificios cuyos nombres comiencen con el valor proporcionado, sin importar las diferencias de mayúsculas y minúsculas.
- `@Query("SELECT e FROM Edificio e WHERE e.name LIKE :text%")`: Define una consulta JPQL personalizada que busca edificios cuyos nombres comiencen con un texto específico. Se utiliza la anotación `@Query` para especificar la consulta JPQL, y se pasa un parámetro `:text` que se vincula con el valor proporcionado en el método.

Estos métodos personalizados permiten realizar consultas específicas en la base de datos para buscar edificios según diferentes criterios, como el nombre o una coincidencia parcial del nombre.

- **Interfaz PisoRepository**

La interfaz `PisoRepository` es un repositorio de Spring Data JPA que se utiliza para acceder y administrar entidades de tipo `Piso` en la base de datos. Al igual que en la explicación anterior, Spring Data JPA proporciona métodos predefinidos para realizar operaciones comunes en la base de datos y también permite definir consultas personalizadas mediante anotaciones o consultas JPQL (Java Persistence Query Language).

Métodos predefinidos

La interfaz `PisoRepository` hereda de `JpaRepository<Piso, Long>`, lo que significa que hereda una serie de métodos predefinidos para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en entidades de tipo `Piso`. Algunos de los métodos predefinidos incluyen:

- `save(Piso piso)`: Guarda un objeto `Piso` en la base de datos.
- `findById(Long id)`: Busca un `Piso` por su identificador único (`id`).
- `findAll()`: Recupera todos los objetos `Piso` de la base de datos.
- `delete(Piso piso)`: Elimina un objeto `Piso` de la base de datos.

Métodos personalizados

Además de los métodos predefinidos, la interfaz `PisoRepository` define algunos métodos personalizados mediante consultas derivadas de nombres y consultas JPQL personalizadas:

- `List<Piso> findAllById(int id)`: Busca todos los pisos cuyo identificador coincida exactamente con el valor proporcionado.
- `@Query("SELECT p FROM Piso p WHERE p.id = :id")`: Define una consulta JPQL personalizada que busca pisos cuyo identificador coincida con un valor específico. Se utiliza la anotación `@Query` para especificar la consulta JPQL y se pasa un parámetro `:id` que se vincula con el valor proporcionado en el método.
- `void deleteById(int id)`: Elimina un piso de la base de datos según su identificador.

Estos métodos personalizados permiten realizar consultas específicas en la base de datos para buscar pisos según diferentes criterios, como el identificador. También proporcionan la capacidad de eliminar un piso por su identificador.

- **Interfaz TipoVehiculoRepository**

La interfaz `TipoVehiculoRepository` es un repositorio de Spring Data JPA que se utiliza para acceder y administrar entidades de tipo `TipoVehiculo` en la base de datos. Al igual que en las explicaciones anteriores, Spring Data JPA proporciona métodos predefinidos para realizar operaciones comunes en la base de datos y también permite definir consultas personalizadas mediante anotaciones o consultas JPQL (Java Persistence Query Language).

Métodos predefinidos

La interfaz `TipoVehiculoRepository` hereda de `JpaRepository<TipoVehiculo, Long>`, lo que significa que hereda una serie de métodos predefinidos para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en entidades de tipo `TipoVehiculo`. Algunos de los métodos predefinidos incluyen:

- `save(TipoVehiculo tipoVehiculo)`: Guarda un objeto `TipoVehiculo` en la base de datos.
- `findById(Long id)`: Busca un `TipoVehiculo` por su identificador único (`id`).
- `findAll()`: Recupera todos los objetos `TipoVehiculo` de la base de datos.

- `delete(TipoVehiculo tipoVehiculo)`: Elimina un objeto `TipoVehiculo` de la base de datos.

Consultas personalizadas

Aunque en la interfaz `TipoVehiculoRepository` no se han definido métodos personalizados explícitamente, es importante destacar que puedes definir consultas personalizadas según tus necesidades específicas utilizando anotaciones `@Query` o métodos derivados de nombres en esta interfaz. Las consultas personalizadas te permiten buscar y filtrar datos de manera más específica en la base de datos.

- **Interfaz VehiculoRepository**

La interfaz `VehiculoRepository` es un repositorio de Spring Data JPA que se utiliza para acceder y administrar entidades de tipo `Vehiculo` en la base de datos. Al igual que en los ejemplos anteriores, Spring Data JPA proporciona métodos predefinidos para realizar operaciones comunes en la base de datos y también permite definir consultas personalizadas mediante anotaciones o consultas JPQL (Java Persistence Query Language).

Métodos predefinidos

La interfaz `VehiculoRepository` hereda de `JpaRepository<Vehiculo, Long>`, lo que significa que hereda una serie de métodos predefinidos para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en entidades de tipo `Vehiculo`. Algunos de los métodos predefinidos incluyen:

- `save(Vehiculo vehiculo)`: Guarda un objeto `Vehiculo` en la base de datos.
- `findById(Long id)`: Busca un `Vehiculo` por su identificador único (`id`).
- `findAll()`: Recupera todos los objetos `Vehiculo` de la base de datos.
- `delete(Vehiculo vehiculo)`: Elimina un objeto `Vehiculo` de la base de datos.

Consultas personalizadas

Además de los métodos predefinidos, puedes definir consultas personalizadas según tus necesidades específicas utilizando anotaciones `@Query` o métodos derivados de nombres en esta interfaz. En el ejemplo proporcionado, se ha definido una consulta JPQL personalizada mediante la anotación `@Query`:

```java @Query("SELECT p FROM Vehiculo p WHERE p.id = :id")
List<Vehiculo> findVehiculoById(int id);```

Esta consulta personalizada permite buscar un `Vehiculo` por su identificador (`id`). Puedes definir consultas adicionales según tus requerimientos de búsqueda.

- **Clase `PisoService`**

La clase `PisoService` es un servicio de Spring que encapsula la lógica de negocio relacionada con los pisos de estacionamiento en tu aplicación. Proporciona métodos para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en entidades de tipo `Piso` y aplica ciertas reglas de negocio relacionadas con la eliminación de pisos que contienen vehículos.

**Métodos**

1. `recuperarPiso(Long id)`: Este método recibe un identificador único (`id`) como parámetro y utiliza el repositorio `PisoRepository` para recuperar un objeto `Piso` de la base de datos
```

según ese identificador. Si no se encuentra un piso con el `id` especificado, se lanza una excepción.
2. `listarPisos()`: Este método utiliza el repositorio `PisoRepository` para recuperar una lista de todos los objetos `Piso` almacenados en la base de datos y la devuelve.
3. `guardarPiso(Piso piso)`: Este método recibe un objeto `Piso` como parámetro y utiliza el repositorio `PisoRepository` para guardar dicho piso en la base de datos.
4. `borrarPiso(Long id)`: Este método recibe un identificador único (`id`) como parámetro y realiza dos comprobaciones:
  - Verifica si existen vehículos en el piso especificado. Si hay vehículos en el piso, devuelve un mensaje indicando que no se puede borrar el piso debido a la presencia de vehículos.
  - Si no hay vehículos en el piso, utiliza el repositorio `PisoRepository` para eliminar el piso de la base de datos y devuelve un mensaje de "Borrado Exitoso".
La clase `PisoService` permite interactuar con los pisos de estacionamiento en tu aplicación, ya sea para recuperar información, listar pisos, guardar nuevos pisos o borrar pisos bajo ciertas condiciones. Esta capa de servicio encapsula la lógica de negocio y se comunica con el repositorio para realizar operaciones en la base de datos.

- **Clase TipoVehiculoService**

La clase `TipoVehiculoService` es un servicio de Spring que encapsula la lógica de negocio relacionada con los tipos de vehículos en tu aplicación. Proporciona métodos para realizar operaciones CRUD (Crear, Leer, Actualizar, Eliminar) en entidades de tipo `TipoVehiculo`.
Métodos
1. `recuperarTipoVehiculo(Long id)`: Este método recibe un identificador único (`id`) como parámetro y utiliza el repositorio `TipoVehiculoRepository` para recuperar un objeto `TipoVehiculo` de la base de datos según ese identificador. Si no se encuentra un tipo de vehículo con el `id` especificado, se lanza una excepción.
2. `listarTipoVehiculo()`: Este método utiliza el repositorio `TipoVehiculoRepository` para recuperar una lista de todos los objetos `TipoVehiculo` almacenados en la base de datos y la devuelve.
3. `guardarTipoVehiculo(TipoVehiculo tipoVehiculo)`: Este método recibe un objeto `TipoVehiculo` como parámetro y utiliza el repositorio `TipoVehiculoRepository` para guardar dicho tipo de vehículo en la base de datos.
4. `borrarTipoVehiculo(Long id)`: Este método recibe un identificador único (`id`) como parámetro y utiliza el repositorio `TipoVehiculoRepository` para eliminar el tipo de vehículo correspondiente de la base de datos. Luego, devuelve un mensaje de "Borrado Exitoso".
La clase `TipoVehiculoService` permite interactuar con los tipos de vehículos en tu aplicación, ya sea para recuperar información, listar tipos de vehículos, guardar nuevos tipos de vehículos o eliminar tipos de vehículos. Esta capa de servicio encapsula la lógica de negocio y se comunica con el repositorio para realizar operaciones en la base de datos.

- **Clase ParkingAppApplication**

Esta clase es la entrada principal de tu aplicación Spring Boot y contiene el método `main`. Aquí se realiza la configuración e inicio de la aplicación Spring Boot.

Métodos

1. `main(String[] args)`: Este es el punto de entrada principal de tu aplicación. Es el método que se ejecuta cuando lanzas la aplicación. Dentro de este método, se utiliza `SpringApplication.run()` para iniciar la aplicación Spring Boot. La clase `ParkingAppApplication.class` se pasa como argumento, lo que le dice a Spring Boot que esta clase es la principal y debe cargarse para iniciar la aplicación.

`@SpringBootApplication`

La anotación `@SpringBootApplication` se coloca en la clase principal de la aplicación Spring Boot. Esta anotación combina varias anotaciones clave que configuran la aplicación de la siguiente manera:

- `@Configuration`: Indica que la clase contiene configuración de Spring.
- `@EnableAutoConfiguration`: Habilita la configuración automática de Spring Boot, que configura automáticamente la aplicación en función de las dependencias y las clases disponibles en el proyecto.
- `@ComponentScan`: Escanea el paquete actual y sus subpaquetes en busca de componentes de Spring para registrarlos y hacerlos disponibles para la aplicación.