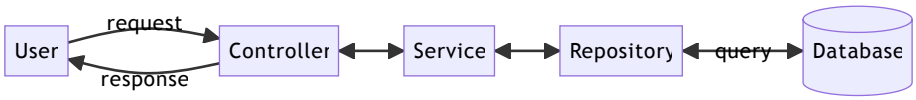
Parking lot management system - Class design and REST APIs

Three-layered architecture

The three layered architecture is a popular architecture for designing software systems. It is also commonly referred to as the **MVC** architecture, or specifically the **MVT** architecture in the case of Django. The three layers are:

1. **Controller layer** - This layer is responsible for handling the user requests and returning the response. It is also responsible for request level validations and transformations.
2. **Service layer** - This layer contains the business logic of the application. In MVC, the service layer is also referred to as the **Model** layer.

3. **Repository layer** - This layer is responsible for interacting with the database and translating the database objects to service objects and vice versa.



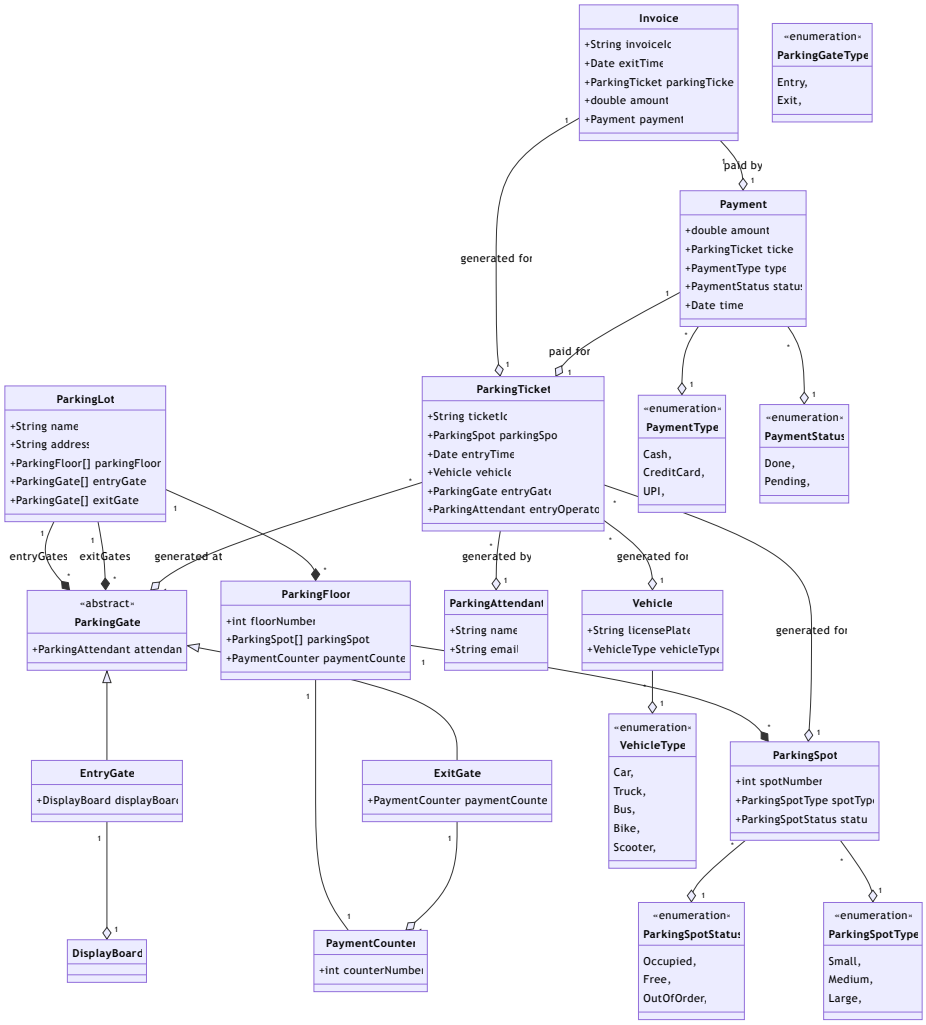
The advantages of this architecture are:

1. **Separation of concerns** - Each layer is responsible for a specific set of functionalities. This makes the code more modular and easier to maintain.
2. **Code reusability** - The code in each layer can be reused in other layers. For example, the service layer can be reused in other classes that need the same business logic.
3. **Testability** - The code in each layer can be tested independently. This makes it easier to write unit tests for the code.
4. **Extensibility** - The code in each layer can be extended independently. For example, if we want to change the database, we can do so without affecting the other layers.

> **Note:** The three layered architecture is a very popular architecture for designing software systems. However, it is not the only architecture. There are other architectures like the **Hexagonal architecture** and the **Clean architecture**. The idea behind all these architectures is the same - **separation of concerns**. The difference is in the way the layers are separated.
> ou can read more about them **here** (https://www.freecodecamp.org/news/a-quick-introduction-to-clean-architecture-990c014448d2/) and **here** (https://reflectoring.io/spring-hexagonal/).

Data models

Before we start implementing the different layers, let us implement the data models that we will be using in the application. This is the class diagram we came up with in the previous session:



Enums

Let us start by quickly implementing all the enums in the class diagram.

```
from enum import Enum

class ParkingSpotType(Enum):
    SMALL = 1
    MEDIUM = 2
    LARGE = 3

class ParkingSpotStatus(Enum):
    OCCUPIED = 1
    FREE = 2
    OUT_OF_ORDER = 3

class PaymentType(Enum):
    CASH = 1
    CREDIT_CARD = 2
    UPI = 3

class PaymentStatus(Enum):
    DONE = 1
    PENDING = 2

class VehicleType(Enum):
    CAR = 1
    TRUCK = 2
    BUS = 3
    BIKE = 4
    SCOOTER = 5
```

Above we used the `Enum` class from the `enum` module to implement the enums. The right hand side of the `=` operator is the value of the enum. The left hand side is the name of the enum. For example, `SMALL` is the name of the enum and `1` is the value of the enum. We can access the value of the enum using the `value` attribute. For example, `ParkingSpotType.SMALL.value` will return `1` and `ParkingSpotType.SMALL.name` will return `SMALL`.

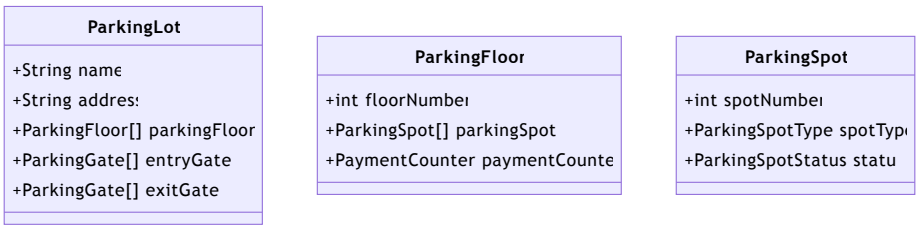
Base class

A lot of the times, there are multiple fields that are duplicated across different classes. For example, the `id` field is present in almost all the classes. Similarly, the `created_at` and `updated_at` fields are also present in most of the classes. It is a common practice to create a base class that contains all these fields and then inherit from this base class in all the other classes. Let us create a base class called `BaseModel` that contains the `id`, `created_at` and `updated_at` fields.

```
from datetime import datetime
from dataclasses import dataclass
from abc import ABC

@dataclass
class BaseModel(ABC):
    id: int
    created_at: datetime
    updated_at: datetime
```

Structural classes

We will now create some important classes in the class diagram. Let us start with the structural classes i.e. `ParkingLot`, `ParkingFloor` and `ParkingSpot`.

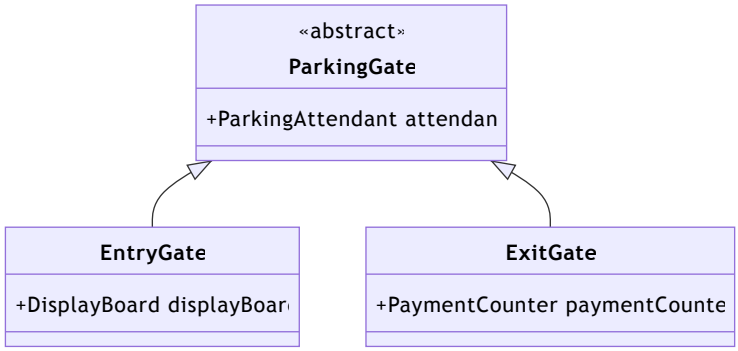


```
from dataclasses import dataclass, field
from typing import List

@dataclass
class ParkingLot(BaseModel):
    name: str
    address: str
    parking_floors: List[ParkingFloor] = field(default_factory=list)
    entry_gates: List[ParkingGate] = field(default_factory=list)
    exit_gates: List[ParkingGate] = field(default_factory=list)
    display_board: DisplayBoard

@dataclass
class ParkingFloor(BaseModel):
    floor_number: int
    parking_spots: List[ParkingSpot] = field(default_factory=list)
    payment_counter: PaymentCounter
    display_board: DisplayBoard

@dataclass
class ParkingSpot(BaseModel):
    spot_number: int
    spot_type: ParkingSpotType
    status: ParkingSpotStatus
```

We have used the `dataclass` decorator from the `dataclasses` module to create the classes. The `field` decorator is used to provide default values for the fields. The `default_factory` argument is used to provide a callable that will be called to create the default value. In the above example, we have used the `list` class as the default factory. This means that the default value of the field will be an empty list. We have also used the `typing` module to specify the type of the fields. This is not mandatory, but it is a good practice to specify the types of the fields.

Gate classes

We can create a single `Gate` class to represent both the entry and exit gates. However, there are some differences between the two. For example, the entry gate has a display board that displays the number of free spots in the parking lot. The exit gate does not have a display board. Similarly, the exit gate has a payment counter where the customer can pay the parking fee. The entry gate does not have a payment counter. So it makes sense to create separate classes for the entry and exit gates, else we will end up with a lot of `if` statements in the `Gate` class due to null values.
We will first create a base class called `ParkingGate` and then inherit from this class to create the `EntryGate` and `ExitGate` classes.



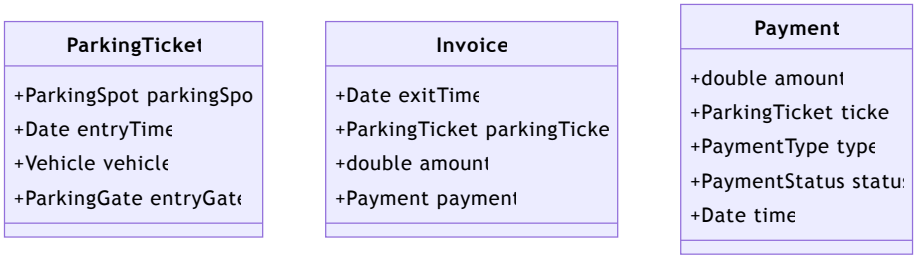

```
from dataclasses import dataclass
from abc import ABC

@dataclass
class ParkingGate(BaseModel, ABC):
    attendant: ParkingAttendant
```

Since both the entry and exit gates will have a parking attendant, we keep it in the base class. We have also used the `ABC` class from the `abc` module to make the `ParkingGate` class an abstract class. This means that we cannot create an instance of the `ParkingGate` class. We can only create instances of the `EntryGate` and `ExitGate` classes.

```
@dataclass
class EntryGate(ParkingGate):
    display_board: DisplayBoard

@dataclass
class ExitGate(ParkingGate):
    payment_counter: PaymentCounter
```

Ticket classes

The `ParkingTicket` class represents the ticket that is generated when a vehicle enters the parking lot. The `Invoice` class represents the invoice that is generated when a vehicle exits the parking lot. These are extremely important classes in the system. Let us implement them.



```
from dataclasses import dataclass
from datetime import datetime

@dataclass
class ParkingTicket(BaseModel):
    parking_spot: ParkingSpot
    entry_time: datetime
    vehicle: Vehicle
    entry_gate: ParkingGate

@dataclass
class Invoice(BaseModel):
    exit_time: datetime
    exit_gate: ParkingGate
    parking_ticket: ParkingTicket
    amount: float
    payment: Payment

@dataclass
class Payment(BaseModel):
    amount: float
    payment_type: PaymentType
    status: PaymentStatus
    time: datetime
```

If you look closely at the invoice class, you will notice that it has a `payment` field of type `Payment`. The cardinality of this relationship is `1:1`. This means that a single payment can be associated with only one invoice and vice versa. However, a real system that accepts payments need to accommodate partial payments. For example, if the parking fee is 100 rupees, the customer might pay 50 rupees in cash and 50 rupees using a credit card. In this case, we will have two payments for a single invoice. To accommodate this, we can change the cardinality of the relationship to `1:N`.

```
@dataclass
class Invoice(BaseModel):
    exit_time: datetime
    exit_gate: ParkingGate
    parking_ticket: ParkingTicket
    amount: float
    payments: List[Payment] = field(default_factory=list)
```

REST APIs

What are APIs?

An **API** is an **Application Programming Interface**. It is a set of functions that allows one application to interact with another application. For example, the Google Maps API allows us to interact with the Google Maps application. We can use the Google Maps API to get the directions from one place to another. We can also use the Google Maps API to

get the distance between two places. Similarly, the Twitter API allows us to interact with the Twitter application. We can use the Twitter API to post tweets, follow other users, etc. **An API allows us to call the functions of another application over the internet.**

What are REST APIs?

REST stands for **REpresentational State Transfer**. It is an architectural style for designing APIs. It is just a set of guidelines that we can follow to design our APIs. The REST architectural style is based on the following principles:

1. **Client-server architecture** - The client and the server are two separate applications. The client is responsible for sending the requests and the server is responsible for sending the responses.
2. **Stateless** - The server does not store any state. This means that the server does not store any information about the client. Each request is independent of the other requests.
3. **Resource-based** - The server exposes resources to the client. A resource is an object that can be accessed using a unique identifier. For example, a user is a resource. We can access a user using the user id. A user can have multiple addresses. An address is also a resource. We can access an address using the address id.

To understand the resource-based principle, let us take see how would we create APIs over the parking spot data model that we created in the previous session. First thing we need to realise is the type of operations that we can perform on the parking spot data model. We can perform the following operations:

1. **Create a parking lot** - The user will send a request to create a parking lot. The request will contain the name and address of the parking lot. The server will create a new parking lot and return the id of the parking lot.
2. **Read a parking lot** - The user will send a request to read a parking lot. The request will contain the id of the parking lot. The server will return the details of the parking lot.

3. **Update a parking lot**
4. **Delete a parking lot**

The above operations are called **CRUD** operations. **CRUD** stands for **Create, Read, Update and Delete**. These are the four basic operations that we can perform on any data model. We can create, read, update and delete a parking lot. We can also create, read, update and delete operations for the other data models.

HTTP Verbs

The HTTP protocol defines a set of verbs that we can use to perform the CRUD operations. The HTTP verbs are:

1. **POST** - This verb is used to create a resource. For example, we can use the POST verb to create a parking lot.
2. **GET** - This verb is used to read a resource. For example, we can use the GET verb to read a parking lot.
3. **PUT** - This verb is used to update a resource. For example, we can use the PUT verb to update a parking lot.
4. **DELETE** - This verb is used to delete a resource. For example, we can use the DELETE verb to delete a parking lot.
5. **PATCH** - This verb is used to update a part of a resource. For example, we can use the PATCH verb to update the name of a parking lot. This is a partial update operation.

REST API design

Let us now design the CRUD APIs for a parking lot. Apart from the verbs, we also need to know which URL to use for each operation. The URL is called the **endpoint**. The CRUD APIs for a parking lot will be:

1. **Create a parking lot** - `POST /parking-lot`
2. **Read a parking lot** - `GET /parking-lot/{id}`
3. **Update a parking lot** - `PUT /parking-lot/{id}`
4. **Delete a parking lot** - `DELETE /parking-lot/{id}`

Apart from the POST API, all the other APIs have an id in the endpoint. This is because we need to specify the id of the parking lot that we want to read, update or delete. The id is passed as a **path parameter** and is a part of the URL. The POST API does not have an id in the endpoint because we are creating a new parking lot, and the server will generate the id for the parking lot.

A three-layered example

In this session, we will not be creating fully fledged REST APIs. We will instead set up the three different layers for creating REST APIs. Let us start with creating the parking lot APIs.

Controller layer

The controller layer is responsible for handling the user requests and returning the response. It is also responsible for request level validations and transformations. Let us create a controller class called `ParkingLotController` that will handle the parking lot requests.

```
class ParkingLotController:
    def create_parking_lot(self, name: str, address: str) -> ParkingLot:
        pass

    def get_parking_lot(self, id: int) -> ParkingLot:
        pass

    def update_parking_lot(self, id: int, name: str, address: str) -> None:
        pass

    def delete_parking_lot(self, id: int) -> None:
        pass
```

The above is the skeleton of a controller class. It contains the four CRUD methods. To make the POST method more extensible, we can create a data transfer object (DTO) called `CreateParkingLotRequest` that will contain the name and address of the parking lot.
If we need to add more fields to the request, we can add them to the DTO.

```
from dataclasses import dataclass

@dataclass
class CreateParkingLotRequest:
    name: str
    address: str
```

We can also validate if the name and address are not empty strings. Request validation is one of the additional responsibilities of the controller layer.

```
class ParkingLotController:
    def create_parking_lot(self, request: CreateParkingLotRequest) -> ParkingLot:
        self.validate_create_request(request)
        pass

    def validate_create_request(self, request: CreateParkingLotRequest) -> None:
        if not request.name:
            raise ValueError("Name cannot be empty")

        if not request.address:
            raise ValueError("Address cannot be empty")
```

It is also preferred to convert the request to a domain object before passing it to the service layer. This is called **request transformation** and is also the other additional responsibility of the controller layer. We'll define a simple method in the controller itself to convert the request to a domain object.

```
class ParkingLotController:
    def create_parking_lot(self, request: CreateParkingLotRequest) -> ParkingLot:
        self.validate_create_request(request)
        parking_lot = self.to_parking_lot(request)
        pass

    def to_parking_lot(self, request: CreateParkingLotRequest) -> ParkingLot:
        return ParkingLot(
            name=request.name,
            address=request.address,
        )
```

Now that we have the parking lot object, we can pass it to the service layer to create the parking lot. We will create a service class called `ParkingLotService` that will contain the business logic of the application.

```
class ParkingLotController:

    def __init__(self):
        self.parking_lot_service = ParkingLotService()

    def create_parking_lot(self, request: CreateParkingLotRequest) -> ParkingLot:
        self.validate_create_request(request)
        parking_lot = self.to_parking_lot(request)
        return self.parking_lot_service.create_parking_lot(parking_lot)
```

Service layer

The service layer is responsible for the business logic of the application. It is also responsible for calling the repository layer to interact with the database. Let us create a service class called `ParkingLotService` that will contain the business logic of the application.

```
class ParkingLotService:
    def create_parking_lot(self, parking_lot: ParkingLot) -> ParkingLot:
        pass

    def get_parking_lot(self, id: int) -> ParkingLot:
        pass

    def update_parking_lot(self, id: int, name: str, address: str) -> None:
        pass

    def delete_parking_lot(self, id: int) -> None:
        pass
```

For now our service class is going to be extremely simple, it will just call the repository layer to create, read, update and delete the parking lot. We will create a repository class called `ParkingLotRepository` that will interact with the database.

```
class ParkingLotService:
    def __init__(self):
        self.parking_lot_repository = ParkingLotRepository()

    def create_parking_lot(self, parking_lot: ParkingLot) -> ParkingLot:
        return self.parking_lot_repository.create_parking_lot(parking_lot)

    def get_parking_lot(self, id: int) -> ParkingLot:
        return self.parking_lot_repository.get_parking_lot(id)
```

Repository layer

The repository layer is responsible for interacting with the database and translating the database objects to service objects and vice versa. Let us create a repository class called `ParkingLotRepository` that will interact with the database. For this session, our database is simply going to be a list of parking lots in memory.

```
class ParkingLotRepository:
    def __init__(self):
        self.parking_lots = []

    def create_parking_lot(self, parking_lot: ParkingLot) -> ParkingLot:
        parking_lot.id = len(self.parking_lots) + 1
        self.parking_lots.append(parking_lot)
        return parking_lot

    def get_parking_lot(self, id: int) -> ParkingLot:
        for parking_lot in self.parking_lots:
            if parking_lot.id == id:
                return parking_lot
        return None
```

Conclusion

In this session, we learnt about the three layered architecture and how to design REST APIs. We also implemented the three layers for the parking lot

management system. In the next session, we will implement the transactional APIs for the parking lot management system.