



Parking lot management system - Ticketing functionality

- Parking lot management system - Ticketing functionality
 - Issue a ticket
 - Request DTO
 - Controller
 - Services and Repositories
 - Gate Service
 - Vehicle Service
 - Spot Allocation
 - Ticket Service
 - Conclusion
 - Next steps

Issue a ticket

In our previous session, we looked the models required for the parking lot management system, along with learning about the three layered architecture. In this session, we will leverage this knowledge to implement the ticketing and invoicing functionalities.

Let's start with the ticketing functionality. The ticketing functionality is responsible for issuing a ticket to the customer when they enter the parking lot.

Request DTO

The API for issuing a ticket would be `POST /ticket`. The ticket data model looks as follows:

```
ParkingTicket
- ParkingSpot parkingSpot
- Date entryTime
- Vehicle vehicle
- ParkingGate entryGate
```

For the client of our API passing the parking spot, vehicle, and entry gate would neither be feasible nor required. The gate should already be persisted in the system and the parking spot should be allocated by the backend. Instead of passing the whole object, we can pass the ID of the gate. The vehicle could be present in the system or not. We can pass the vehicle number and vehicle type. If the vehicle is not present in the system, we can create a new vehicle object and persist it in the system. We also would have to pass the parking lot ID since there can be multiple parking lots in the system.

The new request DTO would look as follows:

```
@dataclass
class IssueTicketRequest:
    parking_lot_id: int
    vehicle_number: str
    vehicle_type: VehicleType
    entry_gate_id: int
    entry_time: datetime
```

Controller

Now that we know the API we have to expose and the format of the request, let's implement the controller. The `TicketController` would look as follows:

```

class TicketController:
    def __init__(self, ticket_service: TicketService):
        self.ticket_service = ticket_service

    def issue_ticket(self, request: IssueTicketRequest) -> ParkingTicket:
        self.validate_request(request)
        return self.ticket_service.issue_ticket(request)

    def validate_request(self, request: IssueTicketRequest):
        if request.parking_spot_id is None:
            raise ValueError("Parking lot ID is required")

        if request.vehicle_number is None:
            raise ValueError("Vehicle number is required")

        if request.entry_gate_id is None:
            raise ValueError("Entry gate ID is required")

```

The `TicketController` takes the `TicketService` as a dependency. The `issue_ticket` method validates the request and delegates the request to the `TicketService`. The `validate_request` method validates the request and raises an exception if the request is invalid.

You will notice that we did not transform the request DTO to the domain model. This is because we would require access to the database to fetch the parking spot and entry gate. The controller does not have nor should access to the database. All the resolvers should be in the service layer. We will see how to transform the request DTO to the domain model in the service layer.

Services and Repositories

Before we implement the `TicketService`, let's write down the different steps involved in issuing a ticket:

1. Check if a parking spot is available
2. Update the parking spot to mark it as occupied
3. Save the spot in the database
4. Fetch the entry gate from the database

5. Create a new vehicle object if the vehicle is not present in the system, else fetch the vehicle from the database
6. Create a new ticket object
7. Save the ticket in the database

Gate Service

To implement the flow, you can see we would need other entities as well. A service class should interact with other service classes and not directly with the database. This is because the service methods have business logic that should be run before interacting with the database. Let us create the classes associated with fetching a gate first.

```
class GateService:
    def __init__(self, gate_repository: GateRepository):
        self.gate_repository = gate_repository

    def get_gate(self, gate_id: int) -> ParkingGate:
        return self.gate_repository.get_gate(gate_id)

class GateRepository:
    def __init__(self):
        self.gates = []

    def get_gate(self, gate_id: int) -> ParkingGate:
        for gate in self.gates:
            if gate.id == gate_id:
                return gate
        raise ValueError("Gate not found")
```

The `GateService` takes the `GateRepository` as a dependency. The `get_gate` method fetches the gate from the repository. The `GateRepository` is an in-memory repository. The `get_gate` method iterates over the gates and returns the gate with the given ID. If the gate is not found, it raises an exception.

Vehicle Service

Let's create the `VehicleService` and `VehicleRepository` as well. Now, the vehicle should be created if it is not present in the system.

```

class VehicleService:
    def __init__(self, vehicle_repository: VehicleRepository):
        self.vehicle_repository = vehicle_repository

    def get_vehicle(self, vehicle_number: str) -> Vehicle:
        return self.vehicle_repository.get_vehicle(vehicle_number)

    def create_vehicle(self, vehicle_number: str, vehicle_type: VehicleType) -> Vehicle:
        vehicle = Vehicle(vehicle_number, vehicle_type)
        return self.vehicle_repository.create_vehicle(vehicle)

    def find_or_create_vehicle(self, vehicle_number: str, vehicle_type: VehicleType) -> Vehicle:
        existing_vehicle = self.get_vehicle(vehicle_number)
        if existing_vehicle is not None:
            return existing_vehicle

        return self.create_vehicle(vehicle_number, vehicle_type)

class VehicleRepository:
    def __init__(self):
        self.vehicles = []

    def get_vehicle(self, vehicle_number: str) -> Vehicle:
        for vehicle in self.vehicles:
            if vehicle.vehicle_number == vehicle_number:
                return vehicle
        return None

    def create_vehicle(self, vehicle: Vehicle) -> Vehicle:
        self.vehicles.append(vehicle)
        return vehicle

```

The `find_or_create_vehicle` method first tries to fetch the vehicle from the repository. If the vehicle is not present, it creates a new vehicle object and persists it in the repository. The `VehicleRepository` is an in-memory repository. The `get_vehicle` method iterates over the vehicles and returns the vehicle with the given vehicle number. If the vehicle is not found, it returns `None`. The `create_vehicle` method appends the vehicle to the list of vehicles and returns the vehicle.

Spot Allocation

We also need to allocate a parking spot to the vehicle. Let's start by creating a simple method in the service to fetch the first available spot according to the spot type.

```
class ParkingSpotService:
    def __init__(self):
        self.spot_repository = ParkingSpotRepository()

    def get_first_available_spot(self, lot_id: int, spot_type: ParkingSpotType) -> ParkingSpot:
        parking_spots = self.spot_repository.get_parking_spots(lot_id)
        for spot in parking_spots:
            if spot.spot_type == spot_type and spot.status == ParkingSpotStatus.FREE:
                return spot
        return None
```

The `get_first_available_spot` method iterates over the parking spots and returns the first available spot with the given spot type. If no spot is available, it returns `None`. This approach works, but it is not extensible. What if we want to allocate the spot nearest to the entry gate? What if we want to allocate the spot nearest to the exit gate? Since there can be multiple ways to allocate a spot, we can use the strategy pattern to make the allocation extensible. Let's create an interface for the spot allocation strategy.

```
class SpotAllocationStrategy(ABC):
    @abstractmethod
    def get_spot(self, parking_spots: List[ParkingSpot], spot_type: ParkingSpotType) -> ParkingSpot:
        pass
```

The `SpotAllocationStrategy` is an abstract class with an abstract method `get_spot`. The `get_spot` method takes the list of parking spots and the spot type and returns the allocated spot. Let's create a strategy that allocates the first available spot.

```

class FirstAvailableSpotAllocationStrategy(SpotAllocationStrategy):
    def get_spot(self, parking_spots: List[ParkingSpot], spot_type: ParkingSpotType) -> ParkingSpot:
        for spot in parking_spots:
            if spot.spot_type == spot_type and spot.status == ParkingSpotStatus.FREE:
                return spot
        return None

```

Now we can modify the `ParkingSpotService` to use the strategy.

```

class ParkingSpotService:
    def __init__(self, spot_allocation_strategy: SpotAllocationStrategy):
        self.spot_repository = ParkingSpotRepository()
        self.spot_allocation_strategy = spot_allocation_strategy

    def allocate_spot(self, lot_id: int, spot_type: ParkingSpotType) -> ParkingSpot:
        parking_spots = self.spot_repository.get_parking_spots(lot_id)
        return self.spot_allocation_strategy.get_spot(parking_spots, spot_type)

```

Ticket Service

Now that we have all the required services, let's implement the `TicketService`. Remember the steps again:

1. Check if a parking spot is available
2. Update the parking spot to mark it as occupied
3. Save the spot in the database
4. Fetch the entry gate from the database
5. Create a new vehicle object if the vehicle is not present in the system, else fetch the vehicle from the database
6. Create a new ticket object
7. Save the ticket in the database

```

class TicketService:
    def __init__(self, spot_service: ParkingSpotService, gate_service: GateService, vehicle_service: VehicleService, ticket_repository: TicketRepository):
        self.spot_service = spot_service
        self.gate_service = gate_service
        self.vehicle_service = vehicle_service
        self.ticket_repository = ticket_repository

    def issue_ticket(self, request: IssueTicketRequest) -> ParkingTicket:
        # Check if a parking spot is available
        parking_spot = self.spot_service.allocate_spot(request.parking_lot_id, decide_spot_type(request.vehicle_type))
        if parking_spot is None:
            raise ValueError("No parking spot available")

        # Update the parking spot to mark it as occupied
        parking_spot.status = ParkingSpotStatus.OCCUPIED

        # Save the spot in the database
        self.spot_service.spot_repository.save_parking_spot(parking_spot)

        # Fetch the entry gate from the database
        entry_gate = self.gate_service.get_gate(request.entry_gate_id)

        # Create a new vehicle object if the vehicle is not present in the system, else fetch it
        vehicle = self.vehicle_service.find_or_create_vehicle(request.vehicle_number, request.vehicle_type)

        # Create a new ticket object
        ticket = ParkingTicket(parking_spot, request.entry_time, vehicle, entry_gate)

        # Save the ticket in the database
        self.ticket_repository.save_ticket(ticket)
        return ticket

    def decide_spot_type(self, vehicle_type: VehicleType) -> ParkingSpotType:
        return {
            VehicleType.CAR: ParkingSpotType.MEDIUM,
            VehicleType.TRUCK: ParkingSpotType.LARGE,
            VehicleType.BUS: ParkingSpotType.LARGE,
            VehicleType.BIKE: ParkingSpotType.SMALL,
            VehicleType.SCOOTER: ParkingSpotType.SMALL
        }

```



```
}[vehicle_type]
```

This is the complete implementation of the `TicketService`. The `issue_ticket` method implements the flow we discussed earlier. The `decide_spot_type` method returns the spot type based on the vehicle type. The `TicketService` takes the `ParkingSpotService`, `GateService`, `VehicleService`, and `TicketRepository` as dependencies.


Conclusion

In this session, we implemented the ticketing functionality. We learned about the strategy pattern and how to use it to make the code extensible.

Next steps

- Implement the invoicing functionality. The invoicing functionality is responsible for calculating the parking fee and generating the invoice for the customer.
- Try to implement the fee calculation logic in an extensible manner.
- Implement an API for the display boards. The display boards should display the number of available spots in the parking lot or the floor.



 You can find the associated code [here](#).