Parking Management System - Project Overview

What Does This System Do?

A REST API system that manages parking slots, vehicles, and parking tickets. It automatically assigns available slots based on vehicle type and tracks parking duration.

Tech Stack

- **Framework:** Spring Boot 3.2
- **Language:** Java 17
- **Database:** H2 (In-Memory)
- **ORM:** Spring Data JPA
- **Build Tool:** Maven
- **Libraries:** Lombok, Validation

Architecture - Layered Design

```
Request Flow:
Client → Controller → Service → Repository → Database
          ↓
     Exception Handler
```

1. Controller Layer (REST APIs)

- Receives HTTP requests
- Validates input using `@Valid`
- Calls service layer
- Returns HTTP responses

2. Service Layer (Business Logic)

- Validates business rules
- Handles parking logic
- Manages transactions
- Logs important events

3. Repository Layer (Data Access)

- JPA repositories
- CRUD operations
- Custom queries

4. Entity Layer (Data Models)

- Vehicle, ParkingSlot, ParkingTicket
- Relationships between entities

5. DTO Layer (Data Transfer)

- Request/Response objects
- Input validation

6. Exception Layer (Error Handling)

- Global exception handler
- Custom exceptions
- Consistent error responses

Core Entities & Relationships

```
Vehicle (1) -------- (N) ParkingTicket (N) -------- (1) ParkingSlot
```

Vehicle

- Stores vehicle information
- Types: CAR, BIKE, TRUCK
- Unique license plate

ParkingSlot

- Represents parking space
- Type-specific (CAR/BIKE/TRUCK)
- Tracks availability

ParkingTicket

- Links vehicle to slot

- Tracks entry/exit time
- Calculates parking duration

Code Flow Examples

Flow 1: Register a Vehicle

```
1. POST /api/vehicles → VehicleController.registerVehicle()
2. @Valid validates VehicleRequest
3. VehicleService.registerVehicle()
  - Check if license plate exists (business rule)
  - Create Vehicle entity
  - Save to database
4. Return Vehicle object (201 Created)
```

Key Code:

```java
// Controller receives request
@PostMapping
public ResponseEntity<Vehicle> registerVehicle(@Valid @RequestBody VehicleRequest request)

// Service validates and saves
if (vehicleRepository.existsByLicensePlate(request.getLicensePlate())) {
  throw new BusinessException("Vehicle already exists");
}
Vehicle vehicle = new Vehicle();
// ... set fields
return vehicleRepository.save(vehicle);
```

Flow 2: Park a Vehicle

```
1. POST /api/park → ParkingController.parkVehicle()
2. ParkingService.parkVehicle()
  - Fetch vehicle by ID
  - Check if already parked (business rule)
  - Find available slot matching vehicle type
  - Mark slot as occupied
  - Create parking ticket
  - Save ticket
3. Return ParkingTicket (201 Created)
```

Key Code:

java

```java
// Find vehicle
Vehicle vehicle = vehicleRepository.findById(vehicleId)
    .orElseThrow(() -> new ResourceNotFoundException("Vehicle not found"));

// Check if already parked
ticketRepository.findByVehicleIdAndExitTimeIsNull(vehicleId)
    .ifPresent(t -> throw new BusinessException("Already parked"));

// Find matching slot
ParkingSlot slot = slotRepository
    .findFirstBySlotTypeAndIsAvailableTrue(slotType)
    .orElseThrow(() -> new BusinessException("No slots available"));

// Occupy slot
slot.setIsAvailable(false);

// Create ticket
ParkingTicket ticket = new ParkingTicket();
ticket.setVehicle(vehicle);
ticket.setSlot(slot);
return ticketRepository.save(ticket);
```

Flow 3: Unpark a Vehicle

```
1. POST /api/unpark/{ticketId} → ParkingController.unparkVehicle()
2. ParkingService.unparkVehicle()
   - Fetch ticket by ID
   - Check if already unparked (business rule)
   - Set exit time
   - Free the slot
   - Update ticket
3. Return updated ParkingTicket (200 OK)
```

Key Code:

java

```
// Get ticket
ParkingTicket ticket = ticketRepository.findById(ticketId)
    .orElseThrow(() -> new ResourceNotFoundException("Ticket not found"));

// Check if already unparked
if (ticket.getExitTime() != null) {
    throw new BusinessException("Already unparked");
}

// Set exit time
ticket.setExitTime(LocalDateTime.now());

// Free slot
ParkingSlot slot = ticket.getSlot();
slot.setIsAvailable(true);
slotRepository.save(slot);

return ticketRepository.save(ticket);
```

Error Handling Flow

```
Exception Occurs → GlobalExceptionHandler catches it → Returns JSON error

Exceptions:
- ResourceNotFoundException → 404
- BusinessException → 400
- MethodArgumentNotValidException → 400 (validation)
```

Example:

```
java

@ExceptionHandler(ResourceNotFoundException.class)
public ResponseEntity<Map<String, Object>> handleNotFound(ResourceNotFoundException ex) {
    log.error("Resource not found: {}", ex.getMessage());
    return buildErrorResponse(ex.getMessage(), HttpStatus.NOT_FOUND);
}
```

Business Rules Implemented

1. ✅ **Unique License Plate** - No duplicate vehicles

2. ✅ **Type Matching** - Slot type must match vehicle type
3. ✅ **No Double Parking** - Vehicle can't be parked twice
4. ✅ **Automatic Slot Assignment** - First available slot assigned
5. ✅ **No Double Unparking** - Can't unpark twice
6. ✅ **Required Fields** - All mandatory fields validated
7. ✅ **Slot Availability** - Tracks occupied/free slots

Validation Strategy

1. Input Validation (DTOs)

java

```
@NotBlank(message = "License plate is required")
private String licensePlate;

@NotNull(message = "Vehicle type is required")
private VehicleType vehicleType;
```

2. Business Validation (Services)

java

```
// Check business rules
if (vehicleRepository.existsByLicensePlate(licensePlate)) {
    throw new BusinessException("Already exists");
}
```

3. Global Error Handling

java

```
@RestControllerAdvice
public class GlobalExceptionHandler {
    // Catches all exceptions
    // Returns consistent error format
}
```

Database Design

Tables Created:

```sql
VEHICLE (id, license_plate, owner_name, vehicle_type, registered_at)
PARKING_SLOT (id, slot_number, slot_type, is_available)
PARKING_TICKET (id, vehicle_id, slot_id, entry_time, exit_time)
```

Relationships:

- ParkingTicket.vehicle_id → Vehicle.id (Many-to-One)
- ParkingTicket.slot_id → ParkingSlot.id (Many-to-One)

Logging Strategy

```java
@Slf4j
public class VehicleService {

    log.info("Important business events");    // INFO
    log.debug("Detailed flow information");   // DEBUG
    log.error("Error scenarios");             // ERROR
}
```

Examples:

- `INFO`: "Vehicle registered with ID: 123"
- `DEBUG`: "Fetching vehicle with ID: 123"
- `ERROR`: "Vehicle not found: 123"

Transaction Management

```java
@Transactional  // Ensures atomic operations
public ParkingTicket parkVehicle(ParkRequest request) {
    // Multiple DB operations
    // All succeed or all rollback
}
```

Why needed?

- Parking involves: updating slot + creating ticket

- Both must succeed or both must fail
- Prevents inconsistent state

Key Design Decisions

1. Why DTOs?

- Separate API contracts from entities
- Add validation at API boundary
- Hide internal entity structure

2. Why Service Layer?

- Centralize business logic
- Reusable across controllers
- Easier to test

3. Why Global Exception Handler?

- Consistent error format
- Single place for error handling
- Clean controller code

4. Why Enums for Types?

- Type safety
- No invalid values
- Clear options

5. Why H2 Database?

- No installation needed
- Fast for development/testing
- Easy to reset

Project Structure

```
com.parking/
├── ParkingSystemApplication.java    # Main class
├── controller/          # REST endpoints
│   ├── VehicleController
│   ├── ParkingSlotController
│   └── ParkingController
├── service/             # Business logic
│   ├── VehicleService
│   ├── ParkingSlotService
│   └── ParkingService
├── repository/          # Data access
│   ├── VehicleRepository
│   ├── ParkingSlotRepository
│   └── ParkingTicketRepository
├── entity/              # Database models
│   ├── Vehicle
│   ├── ParkingSlot
│   └── ParkingTicket
├── dto/                 # Request/Response
│   ├── VehicleRequest
│   ├── ParkingSlotRequest
│   └── ParkRequest
└── exception/           # Error handling
    ├── GlobalExceptionHandler
    ├── ResourceNotFoundException
    └── BusinessException
```

API Design Principles

1. **RESTful** - Proper HTTP methods and status codes

2. **Consistent** - All responses follow same format

3. **Validated** - Input checked at boundary

4. **Documented** - Clear error messages

5. **Stateless** - Each request is independent

Testing Strategy

Unit Tests (Service Layer)

```
java
```

```
@ExtendWith(MockitoExtension.class)
class VehicleServiceTest {
    @Mock VehicleRepository repository;
    @InjectMocks VehicleService service;

    @Test
    void registerVehicle_Success() {
        // Mock dependencies
        // Call service method
        // Assert results
    }
}
```

What's Tested:

- ✅ Business logic
- ✅ Error scenarios
- ✅ Edge cases

Key Takeaways

What Makes This Code Good?

1. **Clean Separation** - Each layer has clear responsibility
2. **Proper Validation** - Input checked, business rules enforced
3. **Error Handling** - Consistent, informative error messages
4. **Logging** - Track important events
5. **Transaction Safety** - Data consistency maintained
6. **Type Safety** - Enums prevent invalid data
7. **Testable** - Service layer easily unit tested
8. **RESTful** - Follows REST principles
9. **Documented** - Clear code structure
10. **Maintainable** - Easy to extend and modify

How to Understand the Code?

Start Here:

1. **Entities** - Understand data models
2. **Repositories** - See database operations
3. **Services** - Read business logic
4. **Controllers** - Check API endpoints
5. **DTOs** - View request/response format
6. **Exception Handler** - See error handling

Follow a Request:

1. Pick an API (e.g., POST /api/park)
2. Start at Controller
3. Follow to Service
4. See Repository calls
5. Check error handling
6. View response format

Time to Understand: ~30 minutes

Lines of Code: ~800 (clean, readable)

Complexity: Medium (well-structured)

Perfect for interview assignments! ✅