Smart Parking Lot Management System (IoT)

You are tasked with developing a simple backend for a **smart parking lot management system** that interacts with IoT sensors in the parking spaces. The system should:

1. **Detect** when a car enters or leaves a parking spot (simulated by API calls).
2. **Track** the number of available parking spots.
3. **Expose a RESTful API** that provides the status of parking spots (e.g., occupied, free).
4. Handle basic CRUD operations for parking spaces and devices (i.e., IoT sensors) to track them.

Requirements:

1. **API Endpoints**:
    - `POST /api/parking-spots/{id}/occupy`: Mark a parking spot as occupied.
    - `POST /api/parking-spots/{id}/free`: Mark a parking spot as free.
    - `GET /api/parking-spots`: Return the status of all parking spots.
    - `POST /api/parking-spots`: Add a new parking spot.
    - `DELETE /api/parking-spots/{id}`: Remove a parking spot.
2. **Business Rules**:
    - A parking spot can either be `free` or `occupied`.
    - The number of available (free) spots should be tracked dynamically.
    - Only IoT devices registered with the system should be able to occupy or free a parking spot.
3. **Simulated IoT Devices**:
    - IoT devices can be represented by simple identifiers (e.g., a GUID for each device).
    - Only registered devices should be allowed to send requests to occupy or free a parking spot.
4. **Basic Validation**:
    - Ensure parking spots cannot be occupied twice without being freed.
    - Ensure parking spots cannot be freed if they are already free.

Objectives for the Candidate:

1. **Design Patterns & Principles**:
    - Apply **SOLID principles** in structuring the application.
    - Utilize **dependency injection** for service management.
    - Ensure separation of concerns through the use of **layers** (e.g., Controllers, Services, Repositories).

    - Implement **DTOs (Data Transfer Objects)** to expose clean and stable APIs.
2. **Clean and Scalable Code**:
    - Ensure the code is readable, maintainable, and follows **clean code** practices.
    - Use meaningful names for methods, classes, and variables.
    - Keep the logic in controllers minimal, delegating business logic to service classes.
3. **Data Storage**:
    - Simulate data storage in-memory (for simplicity) using a repository pattern. Optionally, use a simple SQLite or file-based storage if the candidate prefers.
4. **Error Handling**:
    - Handle edge cases gracefully, such as invalid parking spot IDs, invalid requests from non-registered IoT devices, or when a spot is already occupied/free.
5. **Unit Tests**:
    - Write unit tests for the core business logic (e.g., adding/removing parking spots, changing status, device validation).
    - Focus on testing key scenarios like parking spot occupancy and freeing, ensuring IoT devices interact correctly.
6. **Bonus (Optional)**:
    - Implement a **Rate Limiting** feature to simulate IoT device communication limits (e.g., one action per device per 10 seconds).
    - Return **pagination** when retrieving the status of a large number of parking spots.

Submission Guidelines:

- Source code should be shared as a Git repository (GitHub, GitLab, etc.).
- Include a README with instructions on how to run the solution, including any dependencies or setup steps.
- Use .NET 6 or .NET 8 for the project.

Key Points to Evaluate:

1. **Architecture**: The structure of the solution, separation of concerns, and use of design patterns.
2. **Clean Code**: Is the code well-organized, easy to follow, and aligned with modern .NET practices?
3. **Testability**: Has the candidate written tests for key parts of the application logic?
4. **Performance Awareness**: Does the candidate avoid unnecessary complexity and follow good performance practices for a backend?
5. **Extensibility**: Is the solution designed to be easily extendable with minimal modifications?