

Create PostLoad annotation based spring boot project for data masking more uniquely and less code than less complexity

ChatGPT said:

ChatGPT

Here's a Spring Boot project showcasing a **PostLoad annotation-based approach for data masking**. This approach ensures data masking is applied at the entity level after the data is fetched from the database, with a focus on **unique implementation**, **reduced code complexity**, and **maintainability**.

---

## Project Overview

- **Tools:** Spring Boot, JPA (Hibernate), MySQL (or H2 for testing).
  - **Features:**
    - Automatic data masking using `@PostLoad`.
    - Customizable masking logic encapsulated in utility methods.
    - Clean and concise implementation.
- 

## Project Structure

css

Copy code

```
src/main/java/com/example/masking/  
├── MaskingApplication.java  
├── entity/  
│   ├── User.java  
├── util/  
│   ├── MaskingUtil.java  
├── repository/  
│   ├── UserRepository.java  
├── controller/  
│   ├── UserController.java  
├── dto/  
│   ├── UserDTO.java  
├── service/  
│   ├── UserService.java
```

---

## Implementation

### 1. Main Application Class

java

Copy code

```
package com.example.masking;

import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;

@SpringBootApplication
public class MaskingApplication {
    public static void main(String[] args) {
        SpringApplication.run(MaskingApplication.class, args);
    }
}
```

---

## 2. Entity Class with @PostLoad

java

Copy code

```
package com.example.masking.entity;

import com.example.masking.util.MaskingUtil;
import jakarta.persistence.*;

@Entity
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;

    private String name;
    private String email;
    private String phone;

    // Getters and Setters

    @PostLoad
    public void maskSensitiveData() {
        this.email = MaskingUtil.maskEmail(this.email);
        this.phone = MaskingUtil.maskPhone(this.phone);
    }
}
```

---

### 3. Masking Utility Class

java

Copy code

```
package com.example.masking.util;

public class MaskingUtil {
    public static String maskEmail(String email) {
        if (email == null || !email.contains("@")) {
            return email;
        }
        String[] parts = email.split("@");
        return parts[0].substring(0, 2) + "****@" + parts[1];
    }

    public static String maskPhone(String phone) {
        if (phone == null || phone.length() < 4) {
            return phone;
        }
        return "****" + phone.substring(phone.length() - 4);
    }
}
```

---

### 4. User Repository

java

Copy code

```
package com.example.masking.repository;

import com.example.masking.entity.User;
import org.springframework.data.jpa.repository.JpaRepository;

public interface UserRepository extends JpaRepository<User, Long> {
}
```

---

### 5. Service Layer

java

Copy code

```
package com.example.masking.service;
```

```
import com.example.masking.entity.User;
import com.example.masking.repository.UserRepository;
import org.springframework.stereotype.Service;

import java.util.List;

@Service
public class UserService {
    private final UserRepository userRepository;

    public UserService(UserRepository userRepository) {
        this.userRepository = userRepository;
    }

    public List<User> getAllUsers() {
        return userRepository.findAll();
    }
}
```

---

## 6. User Controller

java

Copy code

```
package com.example.masking.controller;

import com.example.masking.entity.User;
import com.example.masking.service.UserService;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import java.util.List;

@RestController
@RequestMapping("/users")
public class UserController {
    private final UserService userService;

    public UserController(UserService userService) {
        this.userService = userService;
    }
}
```

```
@GetMapping
public List<User> getUsers() {
    return userService.getAllUsers();
}
}
```

---

## 7. Sample Data (Data.sql)

For testing purposes, you can preload the database with sample data:

sql

Copy code

```
INSERT INTO user (name, email, phone) VALUES
('John Doe', 'john.doe@example.com', '1234567890'),
('Jane Smith', 'jane.smith@example.com', '9876543210');
```

---

## 8. Application Configuration

Use either **MySQL** or **H2** for database configuration.

**application.properties:**

properties

Copy code

```
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driver-class-name=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password
spring.jpa.hibernate.ddl-auto=create
spring.h2.console.enabled=true
```

---

## How It Works

### 1. Entity-Level Masking:

- The `@PostLoad` annotation ensures masking logic is applied **immediately after data is fetched** from the database.
- This keeps raw data untouched while providing a masked view to the application.

### 2. Custom Utility:

- `MaskingUtil` provides reusable methods for masking.
  - Adding new masking types is straightforward (e.g., mask SSNs, credit card numbers, etc.).
3. **Centralized Logic:**
    - Masking logic is centralized in the utility class, reducing code duplication and improving maintainability.
  4. **Seamless Integration:**
    - The service layer and controller remain clean, focusing on business logic without handling masking explicitly.
- 

## Example Output

### Database Data:

json

Copy code

```
[
  {
    "name": "John Doe",
    "email": "john.doe@example.com",
    "phone": "1234567890"
  }
]
```

### API Response (GET /users):

json

Copy code

```
[
  {
    "name": "John Doe",
    "email": "jo****@example.com",
    "phone": "****7890"
  }
]
```

---

## Why This Approach?

1. **Efficient:** Masking happens automatically without extra code in services or controllers.
2. **Reusable:** Easy to extend for new masking rules.

3. **Clean:** Keeps code clean and maintainable.
4. **Secure:** Raw data is masked only when exposed, ensuring internal systems have access to unmasked data if needed.

Would you like additional improvements or customization?