

can you create without JpaRepository

ChatGPT said:

ChatGPT

Sure! To create the project without using `JpaRepository`, we can directly work with JDBC by writing SQL queries for the CRUD operations. Below is a complete guide:

Project Structure

The project will include:

1. **Spring Boot Configuration** for H2 database.
 2. **Entity**: Define the `User` class.
 3. **DAO Layer**: Use `JdbcTemplate` to interact with the database.
 4. **Service**: Handle business logic and data masking.
 5. **Controller**: Expose APIs for CRUD operations.
-

Step 1: `application.properties`

properties

Copy code

```
# H2 Database configuration
spring.datasource.url=jdbc:h2:mem:testdb
spring.datasource.driverClassName=org.h2.Driver
spring.datasource.username=sa
spring.datasource.password=password

# H2 Console
spring.h2.console.enabled=true
spring.h2.console.path=/h2-console
```

Step 2: Entity Class

java

Copy code

```
package com.example.datamasking.entity;

public class User {
    private Long id;
    private String name;
```

```
private String email;
private String phoneNumber;

// Getters and setters
public Long getId() { return id; }
public void setId(Long id) { this.id = id; }
public String getName() { return name; }
public void setName(String name) { this.name = name; }
public String getEmail() { return email; }
public void setEmail(String email) { this.email = email; }
public String getPhoneNumber() { return phoneNumber; }
public void setPhoneNumber(String phoneNumber) {
this.phoneNumber = phoneNumber; }
}
```

Step 3: Data Masking Utility

java

Copy code

```
package com.example.datamasking.util;

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

    public static String maskPhoneNumber(String phoneNumber) {
        if (phoneNumber == null || phoneNumber.length() < 4) return
phoneNumber;
        return phoneNumber.replaceAll("(?<=.{4})", "*");
    }
}
```

Step 4: DAO Layer

java

Copy code

```
package com.example.datamasking.dao;

import com.example.datamasking.entity.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Repository;

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.List;

@Repository
public class UserDAO {
    @Autowired
    private JdbcTemplate jdbcTemplate;

    // RowMapper to map SQL rows to User objects
    private static final class UserRowMapper implements
RowMapper<User> {
        @Override
        public User mapRow(ResultSet rs, int rowNum) throws
SQLException {
            User user = new User();
            user.setId(rs.getLong("id"));
            user.setName(rs.getString("name"));
            user.setEmail(rs.getString("email"));
            user.setPhoneNumber(rs.getString("phone_number"));
            return user;
        }
    }

    // Create User
    public int createUser(User user) {
        String sql = "INSERT INTO users (name, email, phone_number)
VALUES (?, ?, ?)";
        return jdbcTemplate.update(sql, user.getName(),
user.getEmail(), user.getPhoneNumber());
    }

    // Get All Users
```

```
        public List<User> getAllUsers() {
            String sql = "SELECT * FROM users";
            return jdbcTemplate.query(sql, new UserRowMapper());
        }
    }
}
```

Step 5: Service Layer

java

Copy code

```
package com.example.datamasking.service;

import com.example.datamasking.dao.UserDAO;
import com.example.datamasking.entity.User;
import com.example.datamasking.util.DataMaskingUtil;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import java.util.List;
import java.util.stream.Collectors;

@Service
public class UserService {
    @Autowired
    private UserDAO userDAO;

    public int saveUser(User user) {
        return userDAO.createUser(user);
    }

    public List<User> getAllUsers() {
        return userDAO.getAllUsers().stream().map(user -> {

            user.setEmail(DataMaskingUtil.maskEmail(user.getEmail()));

            user.setPhoneNumber(DataMaskingUtil.maskPhoneNumber(user.getPhoneNumber()));

            return user;
        }).collect(Collectors.toList());
    }
}
```

Step 6: Controller

java

Copy code

```
package com.example.datamasking.controller;

import com.example.datamasking.entity.User;
import com.example.datamasking.service.UserService;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.*;

import java.util.List;

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

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

    @PostMapping
    public String createUser(@RequestBody User user) {
        int result = userService.saveUser(user);
        return result > 0 ? "User created successfully" : "Failed to
create user";
    }
}
```

Step 7: Initialize H2 Database

Use a `CommandLineRunner` to initialize the database:

java

Copy code

```
package com.example.datamasking;
```

```
import org.springframework.boot.CommandLineRunner;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.jdbc.core.JdbcTemplate;

@Configuration
public class DatabaseInitializer {
    @Bean
    CommandLineRunner initDatabase(JdbcTemplate jdbcTemplate) {
        return args -> {
            jdbcTemplate.execute("CREATE TABLE users (id BIGINT
AUTO_INCREMENT PRIMARY KEY, name VARCHAR(100), email VARCHAR(100),
phone_number VARCHAR(15))");
        };
    }
}
```

Step 8: Run the Application

1. Start the Spring Boot application.
 2. Use `POST /api/users` to add users.
 3. Use `GET /api/users` to retrieve masked user data.
-

Example JSON for Testing

POST Request:

json

Copy code

```
{
  "name": "Jane Doe",
  "email": "jane.doe@example.com",
  "phoneNumber": "9876543210"
}
```

GET Response:

json

Copy code

```
[
  {
    "id": 1,
    "name": "Jane Doe",
    "email": "ja****@example.com",
    "phoneNumber": "*****3210"
  }
]
```

Would you like a downloadable version of this project?

40

You said:

like anyother alternate JbcTemplate

ChatGPT said:

ChatGPT

If you prefer an alternative to `JdbcTemplate` for working with raw SQL in Spring Boot, you can use **Spring's `SimpleJdbcInsert`** or **`NamedParameterJdbcTemplate`**. These are both part of the Spring JDBC framework and provide additional flexibility for interacting with databases.

Here's an example of how to use **`NamedParameterJdbcTemplate`**, which allows named parameters in SQL queries, making them easier to read and maintain.

Using `NamedParameterJdbcTemplate` Instead of `JdbcTemplate`

Step 1: DAO Layer with `NamedParameterJdbcTemplate`

java

Copy code

```
package com.example.datamasking.dao;

import com.example.datamasking.entity.User;
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.jdbc.core.NamedParameterJdbcTemplate;
import org.springframework.jdbc.core.RowMapper;
import org.springframework.stereotype.Repository;
```

```

import java.sql.ResultSet;
import java.sql.SQLException;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

@Repository
public class UserDAO {
    @Autowired
    private NamedParameterJdbcTemplate namedParameterJdbcTemplate;

    // RowMapper to map SQL rows to User objects
    private static final class UserRowMapper implements
RowMapper<User> {
        @Override
        public User mapRow(ResultSet rs, int rowNum) throws
SQLException {
            User user = new User();
            user.setId(rs.getLong("id"));
            user.setName(rs.getString("name"));
            user.setEmail(rs.getString("email"));
            user.setPhoneNumber(rs.getString("phone_number"));
            return user;
        }
    }

    // Create User
    public int createUser(User user) {
        String sql = "INSERT INTO users (name, email, phone_number)
VALUES (:name, :email, :phoneNumber)";
        Map<String, Object> params = new HashMap<>();
        params.put("name", user.getName());
        params.put("email", user.getEmail());
        params.put("phoneNumber", user.getPhoneNumber());
        return namedParameterJdbcTemplate.update(sql, params);
    }

    // Get All Users
    public List<User> getAllUsers() {
        String sql = "SELECT * FROM users";

```



```
        return namedParameterJdbcTemplate.query(sql, new
UserRowMapper());
    }
}
```

Step 2: Service Layer (No Changes)

The service layer remains the same as the previous example.

Step 3: Controller (No Changes)

The controller remains the same as the previous example.

Step 4: Initialize H2 Database (No Changes)

The database initialization step remains the same.

Key Advantages of `NamedParameterJdbcTemplate`:

1. **Readable Queries:** Named parameters (e.g., `:name`, `:email`) make SQL queries more readable than using `?` placeholders.
 2. **Flexibility:** Allows mapping parameters to `Map<String, Object>` for easy handling of dynamic values.
 3. **Compatibility:** Works seamlessly with Spring Boot.
-

Example JSON for Testing

POST Request:

json

Copy code

```
{
    "name": "Alice Doe",
    "email": "alice.doe@example.com",
    "phoneNumber": "9876543210"
}
```

GET Response:

json

Copy code

```
[  
  {  
    "id": 1,  
    "name": "Alice Doe",  
    "email": "al****@example.com",  
    "phoneNumber": "*****3210"  
  }  
]
```