



Spring Boot – Transaction Management Using @Transactional Annotation

Last Updated : 17 Apr, 2024

@Transactional annotation is the metadata used for managing transactions in the Spring Boot application. To configure **Spring Transaction**, this annotation can be applied at the class level or method level. In an enterprise application, a transaction is a sequence of actions performed by the application that together pipelined to perform a single operation. For example, booking a flight ticket is also a transaction where the end user has to enter his information and then make a payment to book the ticket.

Why Do We Need Transaction Management?

Let's understand transactions with the above example, if a user has entered his information the user's information gets stored in the user_info table. Now, to book a ticket he makes an online payment and due to some reason(system failure) the payment has been canceled so, the ticket is not booked for him. But,

Java Arrays Java Strings Java OOPs Java Collection Java 8 Tutorial Java Multithreading Java Exception Handling

large scale, more than thousands of these things happen within a single day. So, it is not good practice to store a single action of the transaction(Here, only user info is stored not the payment info).

To overcome these problems, spring provides transaction management, which uses annotation to handle these issues. In such a scenario, spring stores the user information in temporary memory and then checks for payment information if the payment is successful then it will complete the transaction otherwise it will roll back the transaction and the user information will not get stored in the database.

@Transactional Annotation

In Spring Boot, **@Transactional** annotation is used to manage transactions in a Spring boot application and used to define a scope of transaction. This

annotation can be applied to the class level or method level. It provides data reliability and consistency. When a method is indicated with @Transactional annotation, it indicates that the particular method should be executed within the context of that transaction. If the transaction becomes successful then the changes made to the database are committed, if any transaction fails, all the changes made to that particular transaction can be rollback and it will ensure that the database remains in a consistent state.

Note: To use **@Transactional** annotation, you need to configure transaction management by using **@EnableTransactionManagement** to your main class of Spring Boot application.

Configure Transaction in Spring Boot

In this example, we will create an application to store user information along with his address information and will use spring transaction management to resolve the transaction break problem.

Step By Step Implementation of Transaction Management

Step 1: Create A Spring Boot Project

In this step, we will create a spring boot project. For this, we will be using [Spring Initializr](#). To create a spring boot project please refer to [How to Create a Spring Boot Project?](#)

Step 2: Add Dependencies

We will add the required dependencies for our spring boot application.

The screenshot shows the Spring Initializr interface. On the left, project settings are configured: Project (Maven), Language (Java), Spring Boot (3.0.1), and various metadata fields like Group (com.geeksforgeeks), Artifact (transaction-management), Name (transaction-management), Description (Demo project for Spring Boot), Package name (com.geeksforgeeks.transaction-management), Packaging (Jar), Java version (17), and Java modules (11, 8). On the right, a list of dependencies is shown with a red border:

- Spring Boot DevTools** [DEVELOPER TOOLS]: Provides fast application restarts, LiveReload, and configurations for enhanced development experience.
- Lombok** [DEVELOPER TOOLS]: Java annotation library which helps to reduce boilerplate code.
- Spring Web** [WEB]: Build web, including RESTful, applications using Spring MVC. Uses Apache Tomcat as the default embedded container.
- Spring Data JPA** [SQL]: Persist data in SQL stores with Java Persistence API using Spring Data and Hibernate.
- MySQL Driver** [SQL]: MySQL JDBC and R2DBC driver.

An "ADD DEPENDENCIES..." button is located at the top right of the dependency list.

Step 3: Configure Database

Now, we will configure the database in our application. We will be using the following configurations and add them to our **application.properties** file.

```
server.port = 9090
#database configuration
spring.datasource.url=jdbc:mysql://localhost:3306/employee_db
spring.datasource.username=root
spring.datasource.password=root
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL57Dialect
#the ddl-auto:update : It will create the entity schema and map it to db automatically
spring.jpa.hibernate.ddl-auto:update
spring.jpa.show-sql=true
```

Note: Please add your database username & password along with the database path.

Step 4: Create Model Class

In this step, we will create our model class. Here, we will be creating two model classes, **Employee** and **Address**. While creating the model class we will be using [Lombok Library](#).

Employee.java

Java

```
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.Table;
import lombok.AllArgsConstructorConstructor;
import lombok.Getter;
import lombok.NoArgsConstructorConstructor;
import lombok.Setter;
import lombok.ToString;

@Getter
@Setter
@NoArgsConstructorConstructor
@AllArgsConstructorConstructor
@ToString
@Entity
@Table(name="EMP_INFO")
public class Employee {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private int id;
    private String name;

}
```

Address.java

Java

```
import jakarta.persistence.Entity;
import jakarta.persistence.GeneratedValue;
import jakarta.persistence.GenerationType;
import jakarta.persistence.Id;
import jakarta.persistence.OneToOne;
import jakarta.persistence.Table;
import lombok.AllArgsConstructorConstructor;
import lombok.Getter;
import lombok.NoArgsConstructorConstructor;
import lombok.Setter;
```

```

import lombok.ToString;

@Getter
@Setter
@NoArgsConstructor
@AllArgsConstructor
@ToString
@Entity
@Table(name="ADD_INFO")
public class Address {

    @Id
    @GeneratedValue(strategy = GenerationType.AUTO)
    private Long id;
    private String address;

    // one to one mapping means,
    // one employee stays at one address only
    @OneToOne
    private Employee employee;

}

```

Step 5: Create a Database Layer

In this step, we will create a database layer. For this, we will be creating **EmployeeRepository** and **AddressRepository** and will be extending **JpaRepository<T, ID>** for performing database-related queries.

EmployeeRepository.java

Java

```

import org.springframework.data.jpa.repository.JpaRepository;
import com.geeksforgeeks.transactionmanagement.model.Employee;

public interface EmployeeRepository extends JpaRepository<Employee, Integer> {
}

```

AddressRepository.java

Java

```

import org.springframework.data.jpa.repository.JpaRepository;
import com.geeksforgeeks.transactionmanagement.model.Address;

public interface AddressRepository extends JpaRepository<Address, Integer>
{
}

```

Step 6: Create a Service Layer

You can use **@Transactional** annotation in service layer which will result interacting with the database. In this step, we will create a service layer for our application and add business logic to it. For this, we will be creating two classes **EmployeeService** and **AddressService**. In EmployeeService class we are throwing an exception.

EmployeeService.java

Java

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.geeksforgeeks.transactionmanagement.model.Address;
import com.geeksforgeeks.transactionmanagement.model.Employee;
import
com.geeksforgeeks.transactionmanagement.repository.EmployeeRepository;

@Service
public class EmployeeService {

    @Autowired
    private EmployeeRepository employeeRepository;

    @Autowired
    private AddressService addressService;

    @Transactional
    public Employee addEmployee(Employee employee) throws Exception {
        Employee employeeSavedToDB =
this.employeeRepository.save(employee);

        Address address = new Address();
        address.setId(123L);
        address.setAddress("Varanasi");
    }
}

```

```

        address.setEmployee(employee);

        // calling addAddress() method
        // of AddressService class
        this.addressService.addAddress(address);
        return employeeSavedToDB;
    }
}

```

AddressService.java

Java

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;

import com.geeksforgeeks.transactionmanagement.model.Address;
import com.geeksforgeeks.transactionmanagement.repository.AddressRepository;

@Service
public class AddressService {

    @Autowired
    private AddressRepository addressRepository;

    public Address addAddress(Address address) {
        Address addressSavedToDB = this.addressRepository.save(address);
        return addressSavedToDB;
    }

}

```

Step 7: Create Controller

In this step, we will create a controller for our application. For this, we will create a Controller class and add all the mappings to it.

Controller.java

Java

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.http.HttpStatus;
import org.springframework.http.ResponseEntity;

```

```

import org.springframework.web.bind.annotation.PostMapping;
import org.springframework.web.bind.annotation.RequestBody;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.bind.annotation.RestController;

import com.geeksforgeeks.transactionmanagement.model.Employee;
import com.geeksforgeeks.transactionmanagement.service.EmployeeService;

@RestController
@RequestMapping("/api/employee")
public class Controller {

    @Autowired
    private EmployeeService employeeService;

    @PostMapping("/add")
    public ResponseEntity<Employee> saveEmployee(@RequestBody Employee employee) throws Exception{
        Employee employeeSavedToDB =
this.employeeService.addEmployee(employee);
        return new ResponseEntity<Employee>(employeeSavedToDB,
HttpStatus.CREATED);
    }
}

```

Step 8: Running Our Application

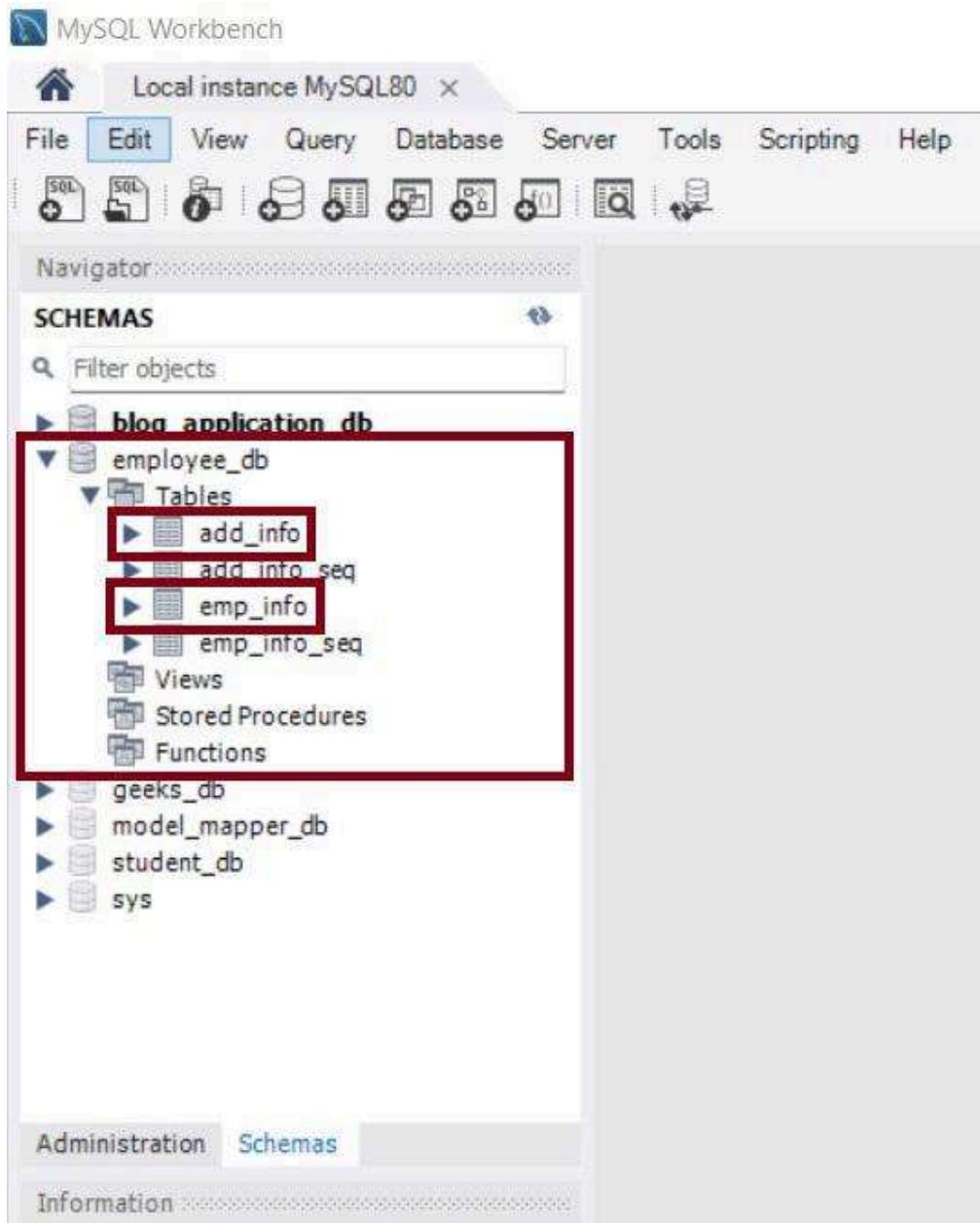
In this step, we will run our application. Once, we run our application using hibernate mapping in our database required tables will be created.

```

2023-01-15T14:10:00.359+05:30  INFO 12048 --- [ restartedMain] org.hibernate.orm.deprecation : HHH0000021: Encountered deprecate
2023-01-15T14:10:00.531+05:30  INFO 12048 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Starting...
2023-01-15T14:10:01.013+05:30  INFO 12048 --- [ restartedMain] com.zaxxer.hikari.pool.HikariPool : HikariPool-1 - Added connection c
2023-01-15T14:10:01.016+05:30  INFO 12048 --- [ restartedMain] com.zaxxer.hikari.HikariDataSource : HikariPool-1 - Start completed.
2023-01-15T14:10:01.048+05:30  INFO 12048 --- [ restartedMain] SQL dialect : HHH000400: Using dialect: org.hib
2023-01-15T14:10:01.048+05:30  WARNN 12048 --- [ restartedMain] org.hibernate.dialect.Dialect : HHH0000024: MySQL5Dialect has b
Hibernate: create table add_info (id bigint not null, address varchar(255), employee_id integer, primary key (id)) engine=InnoDB
Hibernate: create table add_info_seq (next_val bigint) engine=InnoDB
Hibernate: insert into add_info_seq values ( 1 )
Hibernate: create table emp_info (id integer not null, name varchar(255), primary key (id)) engine=InnoDB
Hibernate: create table emp_info_seq (next_val bigint) engine=InnoDB
Hibernate: insert into emp_info_seq values ( 1 )
Hibernate: alter table add_info add constraint FKm3qr2uij5yowcg8lj6vhesii foreign key (employee_id) references emp_info (id)
2023-01-15T14:10:02.834+05:30  INFO 12048 --- [ restartedMain] o.n.e.t.p.l.v.VariactionInitiator : HHH000490: Using StarPlatform impl
2023-01-15T14:10:02.845+05:30  INFO 12048 --- [ restartedMain] j.LocalContainerEntityManagerFactoryBean : Initialized JPA EntityManagerFact
2023-01-15T14:10:03.303+05:30  WARN 12048 --- [ restartedMain] JpaBaseConfiguration$JpaWebConfiguration : spring.jpa.open-in-view is enable
2023-01-15T14:10:03.905+05:30  INFO 12048 --- [ restartedMain] o.s.b.d.a.OptionalLiveReloadServer : LiveReload server is running on p
2023-01-15T14:10:03.993+05:30  INFO 12048 --- [ restartedMain] o.s.b.w.embedded.tomcat.TomcatWebServer : Tomcat started on port(s): 9090 (
2023-01-15T14:10:04.008+05:30  INFO 12048 --- [ restartedMain] c.g.t.TransactionManagementApplication : Started TransactionManagementAppl

```

As we can see in logs, our table has been created. We can also confirm it by looking at our database.



Now, we will request our application for adding an employee to it, using postman. To learn more about postman please refer to [Postman – Working](#), [HTTP Request & Responses](#). Once, we hit the request, **the request moves from the controller to the service layer where our business logic is present.**

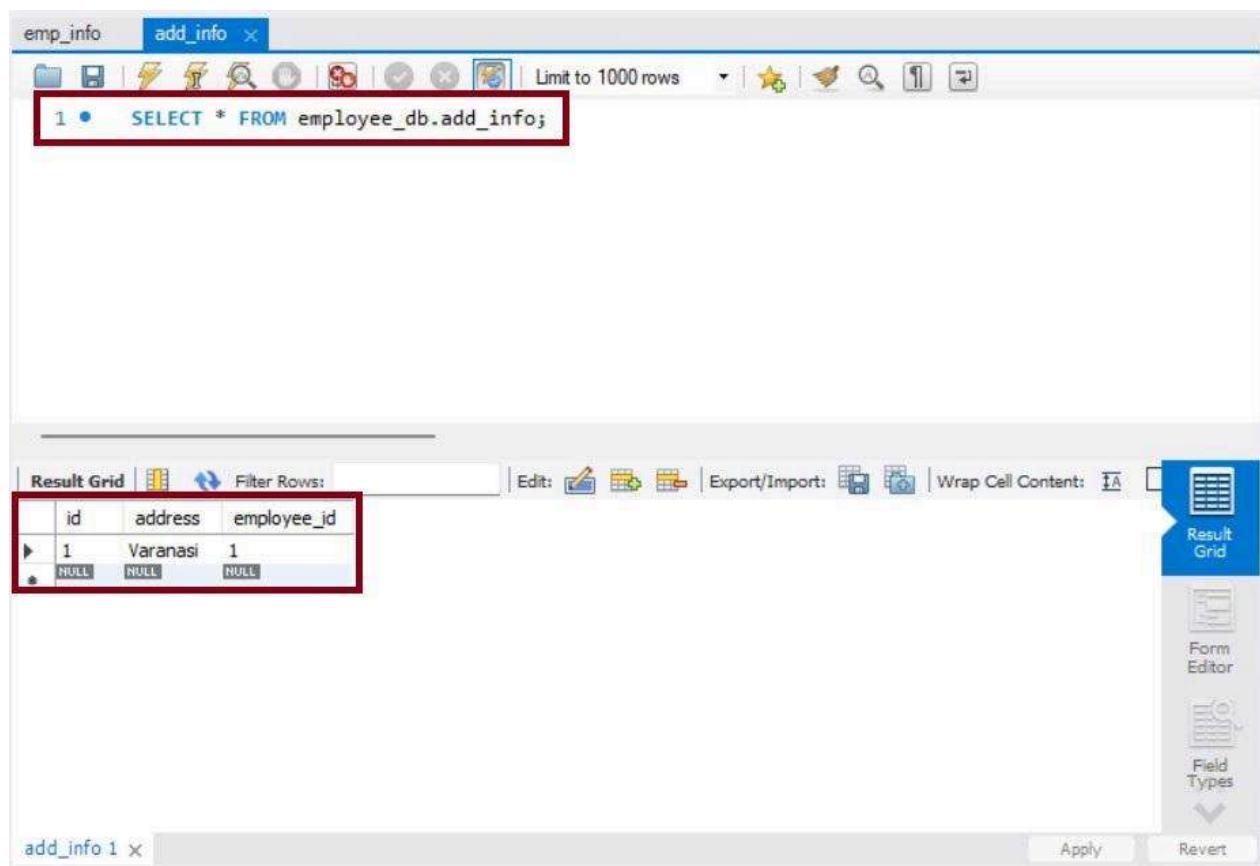
The screenshot shows a Postman interface. In the top left, it says "POST http://localhost:9090/api/employee/add". Below that, the "Body" tab is selected, showing a JSON payload with a single key-value pair: "name": "ankur". The "Test Results" tab at the bottom shows a response with status 201 Created, time 553 ms, and size 192 B. The response body is a JSON object with keys "id" (1) and "name" ("ankur").

As we can see in the above response we have added an employee. We can also check our database for employee data and address data.

The screenshot shows MySQL Workbench. A query window titled "emp_info" contains the SQL command: "SELECT * FROM employee_db.emp_info;". The results grid below shows one row of data:

	id	name
*	1	ankur
*	NULL	NULL

Similarly, we can also check for address data.



Step 9: Problem Without Transaction Management

In the EmployeeService class, we initialize the address object to NULL. Consequently, the employee's details cannot be stored in the database due to the null address object. However, as we are not employing transaction management, the employee basic information will persist in the database. The address details are omitted because of the null value.

Note: Applying the `@Transactional` annotation to a method will not trigger a rollback of any operation if `@EnableTransactionManagement` is not used.

Java

```

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;
import com.geeksforgeeks.transactionmanagement.model.Address;

```

```
import com.geeksforgeeks.transactionmanagement.model.Employee;
import
com.geeksforgeeks.transactionmanagement.repository.EmployeeRepository;

@Service
public class EmployeeService {

    @Autowired
    private EmployeeRepository employeeRepository;

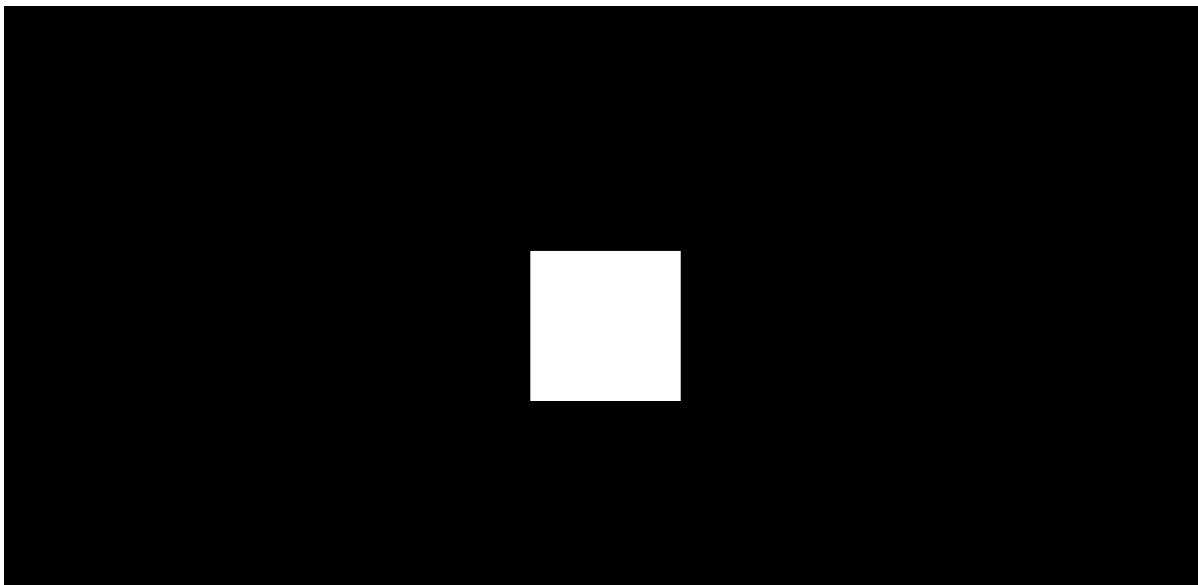
    @Autowired
    private AddressService addressService;

    public Employee addEmployee(Employee employee) throws Exception {
        Employee employeeSavedToDB =
this.employeeRepository.save(employee);

        // we will initialize the
        // address object as null
        Address address = null;
        address.setId(123L);
        address.setAddress("Varanasi");
        address.setEmployee(employee);

        // calling addAddress() method
        // of AddressService class
        this.addressService.addAddress(address);
        return employeeSavedToDB;
    }
}
```

Now, we will delete our table from the database and again run our application and will request the application to create an employee.



00:00

01:01

As we can see in the above media file, we have initialized the address object as null and requested the application, we have an employee created in the database but the address information is not, as we have received a null pointer exception. But, *this is not good practice in transaction management, as employees should be saved only when the address is saved and vice-versa.*

Step 10: Transaction Management

To overcome this problem, we will use **@Transactional** annotation. This will ensure that the transaction should be complete. That is, **either both employee and address data should be stored or nothing will get stored**. For using transaction management, we need to use **@EnableTransactionManagement** in **the main class of our spring boot application** and also, and we need to annotate our **addEmployee()** method in EmployeeService class with **@Transactional** annotation.

TransactionManagementApplication.java

Java

```
import org.springframework.boot.SpringApplication;
import org.springframework.boot.autoconfigure.SpringBootApplication;
import org.springframework.transaction.annotation.EnableTransactionManagement;

@SpringBootApplication
@EnableTransactionManagement
public class TransactionManagementApplication {

    public static void main(String[] args) {
        SpringApplication.run(TransactionManagementApplication.class,
        args);
    }

}
```

EmployeeService.java

Java

```
import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import org.springframework.transaction.annotation.Transactional;

import com.geeksforgeeks.transactionmanagement.model.Address;
import com.geeksforgeeks.transactionmanagement.model.Employee;
import
com.geeksforgeeks.transactionmanagement.repository.EmployeeRepository;

@Service
public class EmployeeService {

    @Autowired
    private EmployeeRepository employeeRepository;

    @Autowired
    private AddressService addressService;

    @Transactional
    public Employee addEmployee(Employee employee) throws Exception {
        Employee employeeSavedToDB =
this.employeeRepository.save(employee);

        // we will initialize the
        // address object as null
        Address address = null;
        address.setId(123L);
        address.setAddress("Varanasi");
        address.setEmployee(employee);

        // calling addAddress() method
        // of AddressService class
        this.addressService.addAddress(address);
        return employeeSavedToDB;
    }
}
```

Step 11: Running Application

Now, we have enabled transaction management for our application. We will again delete the tables from our database and request our application to add an employee.



Conclusion

In this article, we have learned basic configuration of transaction management using in a Spring Boot application. Also we have covered `@Transactional` and `@EnableTransactionManagement` annotation and it's uses with a step by step implementation in a spring boot application.

Feeling lost in the vast world of Backend Development? It's time for a change! Join our [Java Backend Development - Live Course](#) and embark on an exciting journey to master backend development efficiently and on schedule.

What We Offer:

- Comprehensive Course
- Expert Guidance for Efficient Learning
- Hands-on Experience with Real-world Projects
- Proven Track Record with 100,000+ Successful Geeks

How to Create Todo List API using Spring Boot and MySQL?

Spring Boot - Map Entity to DTO using ModelMapper

Share your thoughts in the comments

Add Your Comment

Similar Reads

[Spring Boot - Difference Between @Service Annotation and @Repository Annotation](#)

[Difference Between Spring Boot Starter Web and Spring Boot Starter Tomcat](#)

[Spring Boot - Spring JDBC vs Spring Data JDBC](#)

[Spring vs Spring Boot vs Spring MVC](#)

[Spring Boot - @LoadBalanced Annotation with Example](#)

[Spring Boot - @Async Annotation](#)

[Java Spring Boot Microservices - Develop API Gateway Using Spring Cloud Gateway](#)

[Spring Boot | How to access database using Spring Data JPA](#)

[How to Make a Project Using Spring Boot, MySQL, Spring Data JPA, and Maven?](#)

[Spring Boot - Reactive Programming Using Spring Webflux Framework](#)



ankur035

Article Tags : [Java-Spring-Boot](#), [Technical Scripter 2022](#), [Java](#), [Technical Scripter](#)

Practice Tags : [Java](#)





A-143, 9th Floor, Sovereign Corporate Tower, Sector-136, Noida, Uttar Pradesh - 201305



Company

- [About Us](#)
- [Legal](#)
- [Careers](#)
- [In Media](#)
- [Contact Us](#)
- [Advertise with us](#)
- [GFG Corporate Solution](#)
- [Placement Training Program](#)

Explore

- [Hack-A-Thons](#)
- [GfG Weekly Contest](#)
- [DSA in JAVA/C++](#)
- [Master System Design](#)
- [Master CP](#)
- [GeeksforGeeks Videos](#)
- [Geeks Community](#)

Languages

- [Python](#)
- [Java](#)
- [C++](#)
- [PHP](#)
- [GoLang](#)
- [SQL](#)
- [R Language](#)
- [Android Tutorial](#)
- [Tutorials Archive](#)

DSA

- [Data Structures](#)
- [Algorithms](#)
- [DSA for Beginners](#)
- [Basic DSA Problems](#)
- [DSA Roadmap](#)
- [Top 100 DSA Interview Problems](#)
- [DSA Roadmap by Sandeep Jain](#)
- [All Cheat Sheets](#)

Data Science & ML

- [Data Science With Python](#)
- [Data Science For Beginner](#)
- [Machine Learning Tutorial](#)
- [ML Maths](#)
- [Data Visualisation Tutorial](#)
- [Pandas Tutorial](#)

HTML & CSS

- [HTML](#)
- [CSS](#)
- [Web Templates](#)
- [CSS Frameworks](#)
- [Bootstrap](#)
- [Tailwind CSS](#)

[NumPy Tutorial](#)[SASS](#)[NLP Tutorial](#)[LESS](#)[Deep Learning Tutorial](#)[Web Design](#)[Django Tutorial](#)

Python Tutorial

[Python Programming Examples](#)[Python Projects](#)

Computer Science

[Operating Systems](#)[Python Tkinter](#)[Computer Network](#)[Web Scraping](#)[Database Management System](#)[OpenCV Tutorial](#)[Software Engineering](#)[Python Interview Question](#)[Digital Logic Design](#)[Engineering Maths](#)

DevOps

[Git](#)

Competitive Programming

[AWS](#)[Top DS or Algo for CP](#)[Docker](#)[Top 50 Tree](#)[Kubernetes](#)[Top 50 Graph](#)[Azure](#)[Top 50 Array](#)[GCP](#)[Top 50 String](#)[DevOps Roadmap](#)[Top 50 DP](#)[Top 15 Websites for CP](#)

System Design

[High Level Design](#)

JavaScript

[Low Level Design](#)[JavaScript Examples](#)[UML Diagrams](#)[TypeScript](#)[Interview Guide](#)[ReactJS](#)[Design Patterns](#)[NextJS](#)[OOAD](#)[AngularJS](#)[System Design Bootcamp](#)[NodeJS](#)[Interview Questions](#)[Lodash](#)[Web Browser](#)

Preparation Corner

Company-Wise Recruitment Process
 Resume Templates
 Aptitude Preparation
 Puzzles

Company-Wise Preparation

School Subjects

Mathematics
 Physics
 Chemistry
 Biology
 Social Science
 English Grammar
 World GK

Management & Finance

Management
 HR Management
 Finance
 Income Tax
 Organisational Behaviour
 Marketing

Free Online Tools

Typing Test
 Image Editor
 Code Formatters
 Code Converters
 Currency Converter
 Random Number Generator
 Random Password Generator

More Tutorials

Software Development
 Software Testing
 Product Management
 SAP
 SEO - Search Engine Optimization
 Linux
 Excel

GeeksforGeeks Videos

DSA
 Python
 Java
 C++
 Data Science
 CS Subjects

@GeeksforGeeks, Sanchhaya Education Private Limited, All rights reserved