# Managing Transactions

This guide walks you through the process of wrapping database operations with non-intrusive transactions.

## What You Will Build

You will build a simple JDBC application wherein you make database operations transactional without having to write specialized JDBC code.

## What You Need

- About 15 minutes
- A favorite text editor or IDE
- JDK 1.8 or later
- Gradle 4+ or Maven 3.2+
- You can also import the code straight into your IDE:
    - Spring Tool Suite (STS)
    - IntelliJ IDEA

## How to complete this guide

Like most Spring Getting Started guides, you can start from scratch and complete each step or you can bypass basic setup steps that are already familiar to you. Either way, you end up with working code.

To **start from scratch**, move on to Starting with Spring Initializr.

To **skip the basics**, do the following:

- Download and unzip the source repository for this guide, or clone it using Git: `git clone https://github.com/spring-guides/gs-managing-transactions.git`
- cd into `gs-managing-transactions/initial`
- Jump ahead to Create a Booking Service.

**When you finish**, you can check your results against the code in `gs-managing-transactions/complete`.

## Starting with Spring Initializr

If you use Maven, visit the Spring Initializr to generate a new project with the required dependencies (Spring Data JDBC and H2 Database).

The following listing shows the `pom.xml` file that is created when you choose Maven:

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0" xmlns:xsi="http://www.w3.org/2001/XMLSc
        xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 https://maven.apache.org/xsd
        <modelVersion>4.0.0</modelVersion>
        <parent>
                <groupId>org.springframework.boot</groupId>
                <artifactId>spring-boot-starter-parent</artifactId>
                <version>2.4.3</version>
                <relativePath/> <!-- lookup parent from repository -->
        </parent>
        <groupId>com.example</groupId>
        <artifactId>managing-transactions</artifactId>
        <version>0.0.1-SNAPSHOT</version>
        <name>managing-transactions</name>
        <description>Demo project for Spring Boot</description>
        <properties>
                <java.version>1.8</java.version>
        </properties>
        <dependencies>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-data-jdbc</artifactId>
                </dependency>

                <dependency>
```

```
                        <groupId>com.h2database</groupId>
                        <artifactId>h2</artifactId>
                        <scope>runtime</scope>
                </dependency>
                <dependency>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-test</artifactId>
                        <scope>test</scope>
                </dependency>
        </dependencies>

        <build>
                <plugins>
                        <plugin>
                                <groupId>org.springframework.boot</groupId>
                                <artifactId>spring-boot-maven-plugin</artifactId>
                        </plugin>
                </plugins>
        </build>

</project>
```

If you use Gradle, visit the Spring Initializr to generate a new project with the required dependencies (Spring Data JDBC and H2 Database).

The following listing shows the `build.gradle` file that is created when you choose Gradle:

```
plugins {
        id 'org.springframework.boot' version '2.4.3'
        id 'io.spring.dependency-management' version '1.0.11.RELEASE'
        id 'java'
}

group = 'com.example'
version = '0.0.1-SNAPSHOT'
sourceCompatibility = '1.8'

repositories {
        mavenCentral()
}

dependencies {
        implementation 'org.springframework.boot:spring-boot-starter-data-jdbc'
        runtimeOnly 'com.h2database:h2'
        testImplementation 'org.springframework.boot:spring-boot-starter-test'
}

test {
        useJUnitPlatform()
}
```

## Manual Initialization (optional)

If you want to initialize the project manually rather than use the links shown earlier, follow the steps given below:

1. Navigate to https://start.spring.io. This service pulls in all the dependencies you need for an application and does most of the setup for you.
2. Choose either Gradle or Maven and the language you want to use. This guide assumes that you chose Java.
3. Click **Dependencies** and select **Spring Data JDBC** and **H2 Database**.
4. Click **Generate**.
5. Download the resulting ZIP file, which is an archive of a web application that is configured with your choices.

> If your IDE has the Spring Initializr integration, you can complete this process from your IDE.

## Create a Booking Service

First, you need to use the `BookingService` class to create a JDBC-based service that books people into the system by name. The following listing (from `src/main/java/com/example/managingtransactions/BookingService.java`) shows how to do so:

COPY

```
package com.example.managingtransactions;

import java.util.List;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.stereotype.Component;
import org.springframework.transaction.annotation.Transactional;

@Component
public class BookingService {

  private final static Logger logger = LoggerFactory.getLogger(BookingService.class);
```

```
  private final JdbcTemplate jdbcTemplate;

  public BookingService(JdbcTemplate jdbcTemplate) {
    this.jdbcTemplate = jdbcTemplate;
  }

  @Transactional
  public void book(String... persons) {
    for (String person : persons) {
      logger.info("Booking " + person + " in a seat...");
      jdbcTemplate.update("insert into BOOKINGS(FIRST_NAME) values (?)", person);
    }
  }

  public List<String> findAllBookings() {
    return jdbcTemplate.query("select FIRST_NAME from BOOKINGS",
        (rs, rowNum) -> rs.getString("FIRST_NAME"));
  }

}
```

The code has an autowired `JdbcTemplate`, a handy template class that does all the database interactions needed by the remaining code.

You also have a `book` method that can book multiple people. It loops through the list of people and, for each person, inserts that person into the `BOOKINGS` table by using the `JdbcTemplate`. This method is tagged with `@Transactional`, meaning that any failure causes the entire operation to roll back to its previous state and to re-throw the original exception. This means that none of the people are added to `BOOKINGS` if one person fails to be added.

You also have a `findAllBookings` method to query the database. Each row fetched from the database is converted into a `String`, and all the rows are assembled into a `List`.

## Build an Application

The Spring Initializr provides an application class. In this case, you need not modify this application class. The following listing (from `src/main/java/com/example/managingtransactions/ManagingTransactionsApplication.java`) shows the application class

```java
package com.example.managingtransactions;

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

@SpringBootApplication
public class ManagingTransactionsApplication {

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

}
```

COPY

`@SpringBootApplication` is a convenience annotation that adds all of the following:

- `@Configuration`: Tags the class as a source of bean definitions for the application context.
- `@EnableAutoConfiguration`: Tells Spring Boot to start adding beans based on classpath settings, other beans, and various property settings. For example, if `spring-webmvc` is on the classpath, this annotation flags the application as a web application and activates key behaviors, such as setting up a `DispatcherServlet`.
- `@ComponentScan`: Tells Spring to look for other components, configurations, and services in the `com/example` package, letting it find the controllers.

The `main()` method uses Spring Boot's `SpringApplication.run()` method to launch an application. Did you notice that there was not a single line of XML? There is no `web.xml` file, either. This web application is 100% pure Java and you did not have to deal with configuring any plumbing or infrastructure.

Your application actually has zero configuration. Spring Boot detects `spring-jdbc` and `h2` on the classpath and automatically creates a `DataSource` and a `JdbcTemplate` for you. Because this infrastructure is now available and you have no dedicated configuration, a `DataSourceTransactionManager` is also created for you. This is the component that intercepts the method annotated with `@Transactional` (for example, the `book` method on `BookingService`). The `BookingService` is detected by classpath scanning.

Another Spring Boot feature demonstrated in this guide is the ability to initialize the schema on startup. The following file (from src/main/resources/schema.sql) defines the database schema:

```
drop table BOOKINGS if exists;
create table BOOKINGS(ID serial, FIRST_NAME varchar(5) NOT NULL);
```

COPY

There is also a `CommandLineRunner` that injects the `BookingService` and showcases various transactional use cases. The following listing (from `src/main/java/com/example/managingtransactions/AppRunner.java`) shows the command line runner:

COPY

```
package com.example.managingtransactions;

import org.slf4j.Logger;
import org.slf4j.LoggerFactory;
import org.springframework.boot.CommandLineRunner;
import org.springframework.stereotype.Component;
import org.springframework.util.Assert;

@Component
class AppRunner implements CommandLineRunner {

  private final static Logger logger = LoggerFactory.getLogger(AppRunner.class);

  private final BookingService bookingService;

  public AppRunner(BookingService bookingService) {
    this.bookingService = bookingService;
  }

  @Override
  public void run(String... args) throws Exception {
    bookingService.book("Alice", "Bob", "Carol");
    Assert.isTrue(bookingService.findAllBookings().size() == 3,
        "First booking should work with no problem");
    logger.info("Alice, Bob and Carol have been booked");
    try {
      bookingService.book("Chris", "Samuel");
    } catch (RuntimeException e) {
      logger.info("v--- The following exception is expect because 'Samuel' is too " +
          "big for the DB ---v");
      logger.error(e.getMessage());
    }

    for (String person : bookingService.findAllBookings()) {
      logger.info("So far, " + person + " is booked.");
    }
    logger.info("You shouldn't see Chris or Samuel. Samuel violated DB constraints, " +
        "and Chris was rolled back in the same TX");
    Assert.isTrue(bookingService.findAllBookings().size() == 3,
        "'Samuel' should have triggered a rollback");
```

```
    try {
      bookingService.book("Buddy", null);
    } catch (RuntimeException e) {
      logger.info("v--- The following exception is expect because null is not " +
          "valid for the DB ---v");
      logger.error(e.getMessage());
    }

    for (String person : bookingService.findAllBookings()) {
      logger.info("So far, " + person + " is booked.");
    }
    logger.info("You shouldn't see Buddy or null. null violated DB constraints, and " +
        "Buddy was rolled back in the same TX");
    Assert.isTrue(bookingService.findAllBookings().size() == 3,
        "'null' should have triggered a rollback");
  }

}
```

You can run the application from the command line with Gradle or Maven. You can also build a single executable JAR file that contains all the necessary dependencies, classes, and resources and run that. Building an executable jar makes it easy to ship, version, and deploy the service as an application throughout the development lifecycle, across different environments, and so forth.

If you use Gradle, you can run the application by using `./gradlew bootRun`. Alternatively, you can build the JAR file by using `./gradlew build` and then run the JAR file, as follows:

```
java -jar build/libs/gs-managing-transactions-0.1.0.jar
```

If you use Maven, you can run the application by using `./mvnw spring-boot:run`. Alternatively, you can build the JAR file with `./mvnw clean package` and then run the JAR file, as follows:

```
java -jar target/gs-managing-transactions-0.1.0.jar
```

> The steps described here create a runnable JAR. You can also build a classic WAR file.

You should see the following output:

```
2019-09-19 14:05:25.111  INFO 51911 --- [           main]
c.e.m.ManagingTransactionsApplication    : Starting ManagingTransactionsApplication on
Jays-MBP with PID 51911 (/Users/j/projects/guides/gs-managing-
transactions/complete/target/classes started by j in /Users/j/projects/guides/gs-
managing-transactions/complete)
2019-09-19 14:05:25.114  INFO 51911 --- [           main]
c.e.m.ManagingTransactionsApplication    : No active profile set, falling back to
default profiles: default
2019-09-19 14:05:25.421  INFO 51911 --- [           main]
.s.d.r.c.RepositoryConfigurationDelegate : Bootstrapping Spring Data repositories in
DEFAULT mode.
2019-09-19 14:05:25.438  INFO 51911 --- [           main]
.s.d.r.c.RepositoryConfigurationDelegate : Finished Spring Data repository scanning in
13ms. Found 0 repository interfaces.
2019-09-19 14:05:25.678  INFO 51911 --- [           main]
com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Starting...
2019-09-19 14:05:25.833  INFO 51911 --- [           main]
com.zaxxer.hikari.HikariDataSource       : HikariPool-1 - Start completed.
2019-09-19 14:05:26.158  INFO 51911 --- [           main]
c.e.m.ManagingTransactionsApplication    : Started ManagingTransactionsApplication in
1.303 seconds (JVM running for 3.544)
2019-09-19 14:05:26.170  INFO 51911 --- [           main]
c.e.managingtransactions.BookingService  : Booking Alice in a seat...
2019-09-19 14:05:26.181  INFO 51911 --- [           main]
c.e.managingtransactions.BookingService  : Booking Bob in a seat...
2019-09-19 14:05:26.181  INFO 51911 --- [           main]
c.e.managingtransactions.BookingService  : Booking Carol in a seat...
2019-09-19 14:05:26.195  INFO 51911 --- [           main]
c.e.managingtransactions.AppRunner       : Alice, Bob and Carol have been booked
2019-09-19 14:05:26.196  INFO 51911 --- [           main]
c.e.managingtransactions.BookingService  : Booking Chris in a seat...
2019-09-19 14:05:26.196  INFO 51911 --- [           main]
c.e.managingtransactions.BookingService  : Booking Samuel in a seat...
2019-09-19 14:05:26.271  INFO 51911 --- [           main]
c.e.managingtransactions.AppRunner       : v--- The following exception is expect
because 'Samuel' is too big for the DB ---v
2019-09-19 14:05:26.271 ERROR 51911 --- [           main]
c.e.managingtransactions.AppRunner       : PreparedStatementCallback; SQL [insert into
BOOKINGS(FIRST_NAME) values (?)]; Value too long for column """FIRST_NAME"" VARCHAR(5)
NOT NULL": "'Samuel' (6)"; SQL statement:
insert into BOOKINGS(FIRST_NAME) values (?) [22001-199]; nested exception is
org.h2.jdbc.JdbcSQLDataException: Value too long for column """FIRST_NAME"" VARCHAR(5)
NOT NULL": "'Samuel' (6)"; SQL statement:
insert into BOOKINGS(FIRST_NAME) values (?) [22001-199]
2019-09-19 14:05:26.271  INFO 51911 --- [           main]
c.e.managingtransactions.AppRunner       : So far, Alice is booked.
2019-09-19 14:05:26.271  INFO 51911 --- [           main]
c.e.managingtransactions.AppRunner       : So far, Bob is booked.
2019-09-19 14:05:26.271  INFO 51911 --- [           main]
c.e.managingtransactions.AppRunner       : So far, Carol is booked.
2019-09-19 14:05:26.271  INFO 51911 --- [           main]
c.e.managingtransactions.AppRunner       : You shouldn't see Chris or Samuel. Samuel
violated DB constraints, and Chris was rolled back in the same TX
2019-09-19 14:05:26.272  INFO 51911 --- [           main]
c.e.managingtransactions.BookingService  : Booking Buddy in a seat...
2019-09-19 14:05:26.272  INFO 51911 --- [           main]
c.e.managingtransactions.BookingService  : Booking null in a seat...
2019-09-19 14:05:26.273  INFO 51911 --- [           main]
```

```
c.e.managingtransactions.AppRunner       : v--- The following exception is expect
because null is not valid for the DB ---v
2019-09-19 14:05:26.273 ERROR 51911 --- [           main]
c.e.managingtransactions.AppRunner       : PreparedStatementCallback; SQL [insert into
BOOKINGS(FIRST_NAME) values (?)]; NULL not allowed for column "FIRST_NAME"; SQL
statement:
insert into BOOKINGS(FIRST_NAME) values (?) [23502-199]; nested exception is
org.h2.jdbc.JdbcSQLIntegrityConstraintViolationException: NULL not allowed for column
"FIRST_NAME"; SQL statement:
insert into BOOKINGS(FIRST_NAME) values (?) [23502-199]
2019-09-19 14:05:26.273  INFO 51911 --- [           main]
c.e.managingtransactions.AppRunner       : So far, Alice is booked.
2019-09-19 14:05:26.273  INFO 51911 --- [           main]
c.e.managingtransactions.AppRunner       : So far, Bob is booked.
2019-09-19 14:05:26.273  INFO 51911 --- [           main]
c.e.managingtransactions.AppRunner       : So far, Carol is booked.
2019-09-19 14:05:26.273  INFO 51911 --- [           main]
c.e.managingtransactions.AppRunner       : You shouldn't see Buddy or null. null
violated DB constraints, and Buddy was rolled back in the same TX
```

The `BOOKINGS` table has two constraints on the `first_name` column:

- Names cannot be longer than five characters.
- Names cannot be null.

The first three names inserted are `Alice`, `Bob`, and `Carol`. The application asserts that three people were added to that table. If that had not worked, the application would have exited early.

Next, another booking is done for `Chris` and `Samuel`. Samuel's name is deliberately too long, forcing an insert error. Transactional behavior stipulates that both `Chris` and `Samuel` (that is, all the values in this transaction) should be rolled back. Thus, there should still be only three people in that table, which the assertion demonstrates.

Finally, `Buddy` and `null` are booked. As the output shows, `null` causes a rollback as well, leaving the same three people booked.

## Summary

Congratulations! You have just used Spring to develop a simple JDBC application wrapped with non-intrusive transactions.

## See Also

The following guides may also be helpful:

- Building an Application with Spring Boot
- Accessing Data with JPA
- Accessing Data with MongoDB
- Accessing data with MySQL

Want to write a new guide or contribute to an existing one? Check out our contribution guidelines.