RDBMS

jim stafford

Fall 2022 v2021-02-01: Built: 2022-12-07 06:13 EST

Table of Contents

Chapter 1. Introduction

This lecture will introduce working with relational databases with Spring Boot. It includes the creation and migration of schema, SQL commands, and low-level application interaction with JDBC.

1.1. Goals

The student will learn:

- to identify key parts of a RDBMS schema
- to instantiate and migrate a database schema
- to automate database schema migration
- to interact with database tables and rows using SQL
- to identify key aspects of Java Database Connectivity (JDBC) API

1.2. Objectives

At the conclusion of this lecture and related exercises, the student will be able to:

1. define a database schema that maps a single class to a single table
2. implement a primary key for each row of a table
3. define constraints for rows in a table
4. define an index for a table
5. automate database schema migration with the Flyway tool
6. manipulate table rows using SQL commands
7. identify key aspects of a JDBC call

Chapter 2. Schema Concepts

Relational databases are based on a set of explicitly defined tables, columns, constraints, sequences, and indexes. The overall structure of these definitions is called `schema`. Our first example will be a single table with a few columns.

2.1. RDBMS Tables/Columns

A table is identified by a name and contains a flat set of fields called `columns`. It is common for the table name to have an optional scoping prefix in the event that the database is shared (e.g., during testing or a minimal deployment).

In this example, the `song` table is prefixed by a `reposongs_` name that identifies which course example this table belongs to.

Example Table and Columns

```
                    Table "public.reposongs_song" ①
   Column   |
 ----------+
  id        | ②
  title     | ③
  artist    |
  released  |
```

① table named `reposongs_song`, part of the `reposongs` schema

② column named `id`

③ column named `title`

2.2. Column Data

Individual tables represent a specific type of object and their columns hold the data. Each row of the `song` table will always have an `id`, `title`, `artist`, and `released` column.

Example Table/Column Data

```
  id |            title             |              artist    |  released
 ----+-----------------------------+------------------------+------------
   1 | Noli Me Tangere              | Orbital                | 2002-07-06
   2 | Moab Is My Washpot           | Led Zeppelin           | 2005-03-26
   3 | Arms and the Man             | Parliament Funkadelic  | 2019-03-11
```

2.3. Column Types

Each column is assigned a type that constrains the type and size of value they can hold.

Song Column Types

```
                   Table "public.reposongs_song"
  Column  |          Type          |
----------+------------------------+
 id       | integer                | ①
 title    | character varying(255) | ②
 artist   | character varying(255) |
 released | date                   | ③
```

① `id` column has type integer

② `title` column has type varchar that is less than or equal to 255 characters

③ `released` column has type `date`

2.4. Example Column Types

The following lists several common example column data types. A more complete list of column types can be found on the w3schools web site. Some column types can be vendor-specific.

Table 1. Example Column Types

Category	Example Type
Character Data	• char(size) - a fixed length set of characters • varchar(size) - a variable length of characters • blob(size), clob(size) - a large field of binary or textual data
Boolean/ Numeric data	• boolean - true/false value • int(size), bigint(size) - numeric value • numeric(size, digits) and/or decimal(size, digits) - fixed-point number. e.g. money. numeric definition is more strict in size. decimal definition is "at least" in size. In practice — they tend to be the same.
Temporal data	• date - date without time • time - time without date • datetime - a specific time on a specific date. Timezone is commonly UTC

Character field maximum size is vendor-specific

The maximum size of a char/varchar column is vendor-specific, ranging from 4000 characters to much larger values.

2.5. Constraints

Column values are constrained by their defined type and can be additionally constrained to be required (`not null`), unique (e.g., primary key), a valid reference to an existing row (foreign key), and various other constraints that will be part of the total schema definition.

The following example shows a required column and a unique primary key constraint.

Example Column Types

```
postgres=# \d reposongs_song
                   Table "public.reposongs_song"
  Column  |          Type          | Nullable |
----------+------------------------+----------+
 id       | integer                | not null |①
 title    | character varying(255) |          |
 artist   | character varying(255) |          |
 released | date                   |          |
Indexes:
    "song_pk" PRIMARY KEY, btree (id) ②
```

① column `id` is required

② column `id` constrained to hold a unique (primary) key for each row

2.6. Primary Key

A primary key is used to uniquely identify a specific row within a table and can also be the target of incoming references (foreign keys). There are two origins of a primary key: natural and surrogate. Natural primary keys are derived directly from the business properties of the object. Surrogate primary keys are externally generated and added to the business properties.

The following identifies the two primary key origins and lists a few advantages and disadvantages.

Table 2. Primary Key Origins

Primary Key Origins	Natural PKs	Surrogate PKs
Description	derived directly from business properties of object	externally generated and added to object
Example	• employee ID (e123) • e-mail address (me@gmail.com)	• centrally generated sequence number (1,2,3) • distributed generated UUID (594075a4-5578-459f-9091-e7734d4f58ce)
Advantages	• no new fields are necessary • ID can be determined before DB insert	• guaranteed to be unique • unique business properties permitted to change (e.g. switch email address)

Primary Key Origins	Natural PKs	Surrogate PKs
Disadvantages	• business properties for ID are each required • business properties for ID cannot change • sometimes requires combining multiple properties (i.e., "compound primary key") — which complicates foreign keys	• a new field must be added • visible sequences can be guessed and deterministic increments can be used for size and rate measurement

For this example, I am using a surrogate primary key that could have been based on either a UUID or sequence number.

2.7. UUID

A UUID is a globally unique 128 bit value written in hexadecimal, broken up into five groups using dashes, resulting in a 36 character string.

Example UUID

```
$ uuidgen | awk '{print tolower($0)}'
594075a4-5578-459f-9091-e7734d4f58ce
```

There are different versions of the algorithm, but each target the same structure and the negligible chance of duplication. [1] This provides not only a unique value for the table row, but also a unique value across all tables, services, and domains.

The following lists a few advantages and disadvantages for using UUIDs as a primary key.

Table 3. UUID as Primary Key

UUID Advantages	UUID Disadvantages
• globally unique ◦ easier to search through logs containing information from many tables • can be generated anytime and anywhere ◦ object does not have to wait to be inserted into DB before having an ID — feature similar to natural keys	• larger than needed to be unique for only a table ◦ requires more storage space • slower to compare relative to a smaller integer value ◦ requires additional comparison time

2.8. Database Sequence

A database sequence is a numeric value guaranteed to be unique by the database. Support for sequences and the syntax used to work with them varies per database. The following shows an example of creating, incrementing, and dropping a sequence in postgres.

postgres sequence value

```
postgres=# create sequence seq_a start 1 increment 1; ①
CREATE SEQUENCE

postgres=# select nextval('seq_a'); ②
 nextval
---------
       1
(1 row)

postgres=# select nextval('seq_a');
 nextval
---------
       2
(1 row)

postgres=# drop sequence seq_a;
DROP SEQUENCE
```

① can define starting point and increment for sequence

② obtain next value of sequence using a database query

Database Sequences do not dictate how unique value is used

Database Sequences do not dictate how the unique value is used. The caller can use that directly as the primary key for one or more tables or anything at all. The caller may also use the returned value to self-generate IDs on its own (e.g., a page offset of IDs). That is where the `increment` option can be of use.

2.8.1. Database Sequence with Increment

We can use the `increment` option to help maintain a 1:1 ratio between sequence and primary key values — while giving the caller the ability to self-generate values within a increment window.

Database Sequence with Increment

```
postgres=# create sequence seq_b start 1 increment 100; ①
CREATE SEQUENCE
postgres=# select nextval('seq_b');
 nextval
---------
       1 ①
(1 row)

postgres=# select nextval('seq_b');
 nextval
---------
```

```
     101 ①
(1 row)
```

① increment leaves a window of values that can be self-generated by caller

The database client calls `nextval` whenever it starts or runs out of a window of IDs. This can cause gaps in the sequence of IDs.

[1] "Universally unique identifier", Wikipedia

Chapter 3. Example POJO

We will be using an example `Song` class to demonstrate some database schema and interaction concepts. Initially, I will only show the POJO portions of the class required to implement a business object and manually map this to the database. Later, I will add some JPA mapping constructs to automate the database mapping.

The class is a read-only value class with only constructors and getters. We cannot use the lombok @Value annotation because JPA (part of a follow-on example) will require us to define a no argument constructor and attributes cannot be final.

Song POJO being mapped to database

```
package info.ejava.examples.db.repo.jpa.songs.bo;
...
@Getter ①
@ToString
@Builder
@AllArgsConstructor
@NoArgsConstructor
public class Song {
    private int id; ②
    private String title;
    private String artist;
    private LocalDate released;
}
```

① each property will have a getter method() but the only way to set values is through the constructor/builder

② surrogate primary key will be used as a primary key

POJOs can be read/write

There is no underlying requirement to use a read-only POJO with JPA or any other mapping. However, doing so does make it more consistent with DDD read-only entity concepts where changes are through explicit save/update calls to the repository versus subtle side-effects of calling an entity `setter()`.

Chapter 4. Schema

To map this class to the database, we will need the following constructs:

- a table
- a sequence to generate unique values for primary keys
- an integer column to hold `id`
- 2 varchar columns to hold `title` and `artist`
- a date column to hold `released`

The constructs are defined by `schema`. Schema is instantiated using specific commands. Most core schema creation commands are vendor neutral. Some schema creation commands (e.g., `IF EXISTS`) and options are vendor-specific.

4.1. Schema Creation

Schema can be

- authored by hand,
- auto-generated, or
- a mixture of the two.

We will have the tooling necessary to implement auto-generation once we get to JPA, but we are not there yet. For now, we will start by creating a complete schema definition by hand.

4.2. Example Schema

The following example defines a sequence and a table in our database ready for use with postgres.

Schema Creation Example (V1.0.0__initial_schema.sql)

```
drop sequence IF EXISTS hibernate_sequence; ①
drop table IF EXISTS reposongs_song;

create sequence hibernate_sequence start 1 increment 1; ②
create table reposongs_song (
    id int not null,
    title varchar(255),
    artist varchar(255),
    released date,
    constraint song_pk primary key (id)
);

comment on table reposongs_song is 'song database'; ③
comment on column reposongs_song.id is 'song primary key';
comment on column reposongs_song.title is 'official song name';
comment on column reposongs_song.artist is 'who recorded song';
comment on column reposongs_song.released is 'date song released';

create index idx_song_title on reposongs_song(title);
```

① remove any existing residue

② create new DB table(s) and sequence

③ add descriptive comments

Chapter 5. Schema Command Line Population

To instantiate the schema, we have the option to use the command line interface (CLI). The following example connects to a database running within docker-compose. The `psql` CLI is executed on the same machine as the database, thus saving us the requirement of supplying the password. The contents of the schema file is supplied as stdin.

Schema Command Line Population

```
$ docker-compose up -d postgres
Creating ejava_postgres_1 ... done

$ docker-compose exec -T postgres psql -U postgres \ ① ②
< .../src/main/resources/db/migration/V1.0.0_0__initial_schema.sql ③
DROP SEQUENCE
DROP TABLE
NOTICE:  sequence "hibernate_sequence" does not exist, skipping
NOTICE:  table "reposongs_song" does not exist, skipping
CREATE SEQUENCE
CREATE TABLE
COMMENT
COMMENT
COMMENT
COMMENT
COMMENT
```

① running `psql` CLI command on `postgres` image

② `-T` disables docker-compose pseudo-tty allocation

③ reference to schema file on host

Pass file using stdin

The file is passed in through stdin using the "<" character. Do not miss adding the "<" character.

The following schema commands add an index to the `title` field.

Additional Schema Index

```
$ docker-compose exec -T postgres psql -U postgres \
< .../src/main/resources/db/migration/V1.0.0_1__initial_indexes.sql
CREATE INDEX
```

5.1. Schema Result

We can log back into the database to take a look at the resulting schema. The following executes the `psql` CLI interface in the postgres image.

Interactive Login to postgres

```
$ docker-compose exec postgres psql -U postgres
psql (12.3)
Type "help" for help.
#
```

5.2. List Tables

The following lists the tables created in the postgres database.

List Tables

```
postgres=# \d+
                               List of relations
 Schema |        Name         |   Type    |  Owner   |    Size    |  Description
--------+---------------------+-----------+----------+------------+---------------
 public | hibernate_sequence | sequence | postgres | 8192 bytes |
 public | reposongs_song     | table    | postgres | 8192 bytes | song database
(2 rows)
```

5.3. Describe Song Table

Describe Song Table

```
postgres=# \d reposongs_song
                   Table "public.reposongs_song"
  Column  |          Type          | Collation | Nullable | Default
----------+------------------------+-----------+----------+---------
 id       | integer                |           | not null |
 title    | character varying(255) |           |          |
 artist   | character varying(255) |           |          |
 released | date                   |           |          |
Indexes:
    "song_pk" PRIMARY KEY, btree (id)
    "idx_song_title" btree (title)
```

Chapter 6. RDBMS Project

Although it is common to execute schema commands interactively during initial development, sooner or later they should end up documented in source file(s) that can help document the baseline schema and automate getting to a baseline schema state. Spring Boot provides direct support for automating schema migration — whether it be for test environments or actual production migration. This automation is critical to modern dynamic deployment environments. Lets begin filling in some project-level details of our example.

6.1. RDBMS Project Dependencies

To get our project prepared to communicate with the database, we are going to need a RDBMS-based spring-data starter and at least one database dependency.

The following dependency example readies our project for JPA (a layer well above RDBMS) and to be able to use either the `postgres` or `h2` database.

- h2 is an easy and efficient in-memory database choice to base unit testing. Other in-memory choices include HSQLDB and Derby databases.
- `postgres` is one of many choices we could use for a production-ready database

RDBMS Project Dependencies

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-data-jpa</artifactId> ①
</dependency>
②
<dependency>
    <groupId>org.postgresql</groupId>
    <artifactId>postgresql</artifactId>
    <scope>runtime</scope>
</dependency>
<dependency>
    <groupId>com.h2database</groupId>
    <artifactId>h2</artifactId>
    <scope>runtime</scope>
</dependency>
<!-- schema management --> ③
<dependency>
    <groupId>org.flywaydb</groupId>
    <artifactId>flyway-core</artifactId>
    <scope>runtime</scope>
</dependency>
```

① brings in all dependencies required to access database using JPA (including APIs and Hibernate implementation)

② defines two database clients we have the option of using — `h2` offers an in-memory server

③ brings in a schema management tool

6.2. RDBMS Access Objects

The JPA starter takes care of declaring a few key `@Bean` instances that can be injected into components.

- `javax.sql.DataSource` is part of the standard JDBC API — which is a very mature and well-supported standard
- `javax.persistence.EntityManager` is part of the standard JPA API — which is a layer above JDBC and also a well-supported standard.

Key RDBMS Objects

```
@Autowired
private javax.sql.DataSource dataSource; ①

@Autowired
private javax.persistence.EntityManager entityManager; ②
```

① `DataSource` defines a starting point to interface to database using JDBC

② `EntityManager` defines a starting point for JPA interaction with the database

6.3. RDBMS Connection Properties

Spring Boot will make some choices automatically, but since we have defined two database dependencies, we should be explicit. The default datasource is defined with the `spring.datasource` prefix. The URL defines which client to use. The driver-class-name and dialect can be explicitly defined, but can also be determined internally based on the URL and details reported by the live database.

The following example properties define an in-memory h2 database.

h2 in-memory database properties

```
spring.datasource.url=jdbc:h2:mem:songs
#spring.datasource.driver-class-name=org.h2.Driver ①
```

① Spring Boot can automatically determine driver-class-name from provided URL

The following example properties define a postgres client. Since this is a server, we have other properties — like username and password — that have to be supplied.

postgres server database client properties

```
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
spring.datasource.username=postgres
spring.datasource.password=secret
#spring.datasource.driver-class-name=org.postgresql.Driver
```

```
#spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect
```

Driver can be derived from JDBC URL

In a normal Java application, JDBC drivers automatically register with the JDBC DriverManager at startup. When a client requests a connection to a specific JDBC URL, the JDBC DriverManager interrogates each driver, looking for support for the provided JDBC URL.

Chapter 7. Schema Migration

The schema of a project rarely stays constant and commonly has to migrate from version to version. No matter what can be automated during development, we need to preserve existing data in production and formal integration environments. Spring Boot has a default integration with Flyway in order to provide ordered migration from version to version. Some of its features (e.g., undo) require a commercial license, but its open-source offering implements forward migrations for free.

7.1. Flyway Automated Schema Migration

"Flyway is an open-source database migration tool". [1] It comes pre-integrated with Spring Boot once we add the Maven module dependency. Flyway executes provided SQL migration scripts against the database and maintains the state of the migration for future sessions.

7.2. Flyway Schema Source

By default, schema files [2]

- are searched for in the `classpath:db/migration` directory
  - overridden using `spring.flyway.locations` property
  - locations can be from the classpath and filesystem
  - location expressions support `{vendor}` placeholder expansion

```
spring.flyway.locations=classpath:db/migration/common,classpath:db/migration/{vendor}
```

- following a naming pattern of V<version>__<name/comment>.sql (double underscore between version and name/comment) with version being a period (".") or single underscore ("_") separated set of version digits (e.g., V1.0.0_0, V1_0_0_0)

The following example shows a set of schema migration files located in the default, vendor neutral location.

Schema Migration Target Folder

```
target/classes/
|-- application-postgres.properties
|-- application.properties
`-- db
    `-- migration
        |-- V1.0.0_0__initial_schema.sql
        |-- V1.0.0_1__initial_indexes.sql
        `-- V1.1.0_0__add_artist.sql
```

7.3. Flyway Automatic Schema Population

Spring Boot will automatically trigger a migration of the files when the application starts.

The following example is launching the application and activating the `postgres` profile with the client setup to communicate with the remote postgres database. The `--db.populate` is turning off application level population of the database. That is part of a later example.

Active Database Server Profile

```
java -jar target/jpa-song-example-6.0.1-SNAPSHOT-bootexec.jar
--spring.profiles.active=postgres --db.populate=false
```

7.4. Database Server Profiles

By default, the example application will use an in-memory database.

application.properties

```
#h2
spring.datasource.url=jdbc:h2:mem:users
```

To use the postgres database, we need to fill in the properties within the selected profile.

application-postgres.properties

```
#postgres
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
spring.datasource.username=postgres
spring.datasource.password=secret
```

7.5. Dirty Database Detection

If flyway detects a non-empty schema and no flyway table(s), it will immediately throw an exception and the program terminates.

Flyway Detects an Error

```
FlywayException: Found non-empty schema(s) "public" but no schema history table.
Use baseline() or set baselineOnMigrate to true to initialize the schema history
table.
```

Keeping this simple, we can simply drop the existing schema.

Drop Existing

```
postgres=# drop table reposongs_song;
```

```
DROP TABLE
postgres=# drop sequence hibernate_sequence;
DROP SEQUENCE
```

7.6. Flyway Migration

With everything correctly in place, flyway will execute the migration.

The following output is from the console log showing the activity of Flyway migrating the schema of the database.

Flyway Migration Debug Log Statements

```
VersionPrinter : Flyway Community Edition 7.1.1 by Redgate
DatabaseType   : Database: jdbc:postgresql://localhost:5432/postgres (PostgreSQL 12.3)
DbValidate     : Successfully validated 3 migrations (execution time 00:00.026s)
JdbcTableSchemaHistory : Creating Schema History table
"public"."flyway_schema_history" ...
DbMigrate : Current version of schema "public": << Empty Schema >>
DbMigrate : Migrating schema "public" to version "1.0.0.0 - initial schema"
DefaultSqlScriptExecutor  : DB: sequence "hibernate_sequence" does not exist, skipping
DefaultSqlScriptExecutor  : DB: table "reposongs_song" does not exist, skipping
DbMigrate : Migrating schema "public" to version "1.0.0.1 - initial indexes"
DbMigrate : Migrating schema "public" to version "1.1.0.0 - add artist"
DbMigrate : Successfully applied 3 migrations to schema "public" (execution time
00:00.190s)
```

[1] "Flyway Documentation",Flyway Web Page

[2] "Execute Flyway Database Migrations on Startup",docs.spring.io Web Site

Chapter 8. SQL CRUD Commands

All RDBMS-based interactions are based on Structured Query Language (SQL) and its set of Data Manipulation Language (DML) commands. It will help our understanding of what the higher-level frameworks are providing if we take a look at a few raw examples.

SQL Commands are case-insensitive

All SQL commands are case-insensitive. Using upper or lower case in these examples is a matter of personal/project choice.

8.1. H2 Console Access

When H2 is activated — we can activate the H2 user interface using the following property.

Activating H2 Console

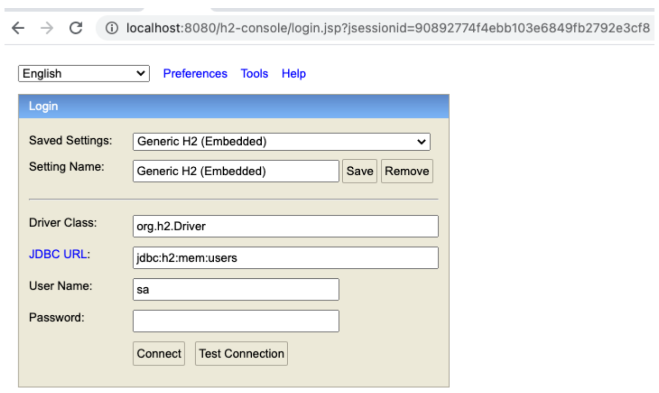
```
spring.h2.console.enabled=true
```

Once the application is up and running, the following URL provides access to the H2 console.

Accessing H2 Console

```
http://localhost:8080/h2-console
```

Table 4. H2 Console Windows



localhost:8080/h2-console/login.jsp?jsessionid=90892774f4ebb103e6849fb2792e3cf8

English

Login

Saved Settings: Generic H2 (Embedded)

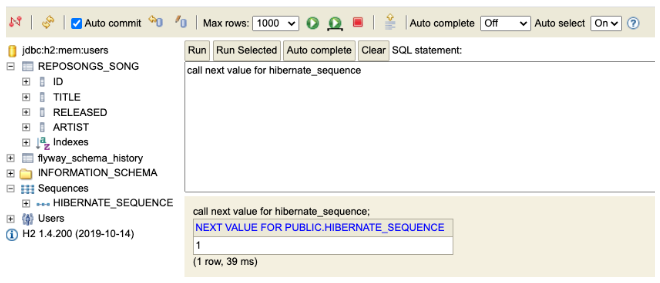
Setting Name: Generic H2 (Embedded)

Driver Class: org.h2.Driver

JDBC URL: jdbc:h2:mem:users

User Name: sa

Password:



Auto commit ✓ Max rows: 1000 Auto complete Off Auto select On

jdbc:h2:mem:users
REPOSONGS_SONG
ID
TITLE
RELEASED
ARTIST
Indexes
flyway_schema_history
INFORMATION_SCHEMA
Sequences
HIBERNATE_SEQUENCE
Users
H2 1.4.200 (2019-10-14)

SQL statement:

```
call next value for hibernate_sequence
```

```
call next value for hibernate_sequence;
NEXT VALUE FOR PUBLIC.HIBERNATE_SEQUENCE
1
(1 row, 39 ms)
```

8.2. Postgres CLI Access

With postgres activated, we can access the postgres server using the `psql` CLI.

Postgres Command Line Interface Access

```
$ docker-compose exec postgres psql -U postgres
psql (12.3)
Type "help" for help.
```

```
postgres=#
```

8.3. Next Value for Sequence

We created a sequence in our schema to managed unique IDs. We can obtain the next value for that sequence using a SQL command. Unfortunately, obtaining the next value for a sequence is vendor-specific. The following two examples show examples for postgres and h2.

postgres sequence next value example

```
select nextval('hibernate_sequence');
 nextval
---------
       6
```

h2 sequence next value example

```
call next value for hibernate_sequence;
---
1
```

8.4. SQL ROW INSERT

We add data to a table using the INSERT command.

SQL INSERT Example

```
insert into reposongs_song(id, title, artist, released)
values (6,'Don''t Worry Be Happy','Bobby McFerrin', '1988-08-05');
```

Use two single-quote characters to embed single-quote

The single-quote character is used to delineate a string in SQL commands. Use two single-quote characters to express a single quote character within a command (e.g., `Don''t`).

8.5. SQL SELECT

We output row data from the table using the SELECT command;

SQL SELECT Wildcard Example

```
# select * from reposongs_song;

 id |        title         |     artist     |  released
----+----------------------+----------------+------------
  6 | Don't Worry Be Happy | Bobby McFerrin | 1988-08-05
  7 | Sledgehammer         | Peter Gabriel  | 1986-05-18
```

The previous example output all columns and rows for the table in a non-deterministic order. We can control the columns output, the column order, and the row order for better management. The next example outputs specific columns and orders rows in ascending order by the released date.

SQL SELECT Columns and Order Example

```
# select released, title, artist from reposongs_song order by released ASC;
  released  |        title         |     artist
------------+----------------------+----------------
 1986-05-18 | Sledgehammer         | Peter Gabriel
 1988-08-05 | Don't Worry Be Happy | Bobby McFerrin
```

8.6. SQL ROW UPDATE

We can change column data of one or more rows using the UPDATE command.

The following example shows a row with a value that needs to be changed.

Incorrect Row

```
# insert into reposongs_song(id, title, artist, released)
values (8,'October','Earth Wind and Fire', '1978-11-18');
```

The following snippet shows updating the title column for the specific row.

SQL UDPATE Example

```
# update reposongs_song set title='September' where id=8;
```

The following snippet uses the SELECT command to show the results of our change.

SQL UPDATE Result

```
# select * from reposongs_song where id=8;

 id |   title   |       artist        |  released
----+-----------+---------------------+------------
  8 | September | Earth Wind and Fire | 1978-11-18
```

8.7. SQL ROW DELETE

We can remove one or more rows with the DELETE command. The following example removes a specific row matching the provided ID.

SQL DELETE Example

```
# delete from reposongs_song where id=8;
DELETE 1
```

```
# select * from reposongs_song;
 id |        title         |     artist      |  released
----+----------------------+-----------------+------------
  6 | Don't Worry Be Happy | Bobby McFerrin | 1988-08-05
  7 | Sledgehammer         | Peter Gabriel  | 1986-05-18
```

8.8. RDBMS Transaction

Transactions are an important and integral part of relational databases. The transactionality of a database are expressed in "ACID" properties [1]:

- Atomic - all or nothing. Everything in the unit acts as a single unit
- Consistent - moves from one valid state to another
- Isolation - the degree of visibility/independence between concurrent transactions
- Durability - a committed transaction exists

By default, most interactions with the database are considered individual transactions with an auto-commit after each one. Auto-commit can be disabled so that multiple commands can be part of the same, single transaction.

8.8.1. BEGIN Transaction Example

The following shows an example of a disabling auto-commit in postgres by issuing the BEGIN command. Every change from this point until the COMMIT or ROLLBACK is temporary and is isolated from other concurrent transactions (to the level of isolation supported by the database and configured by the connection).

BEGIN Transaction Example

```
# BEGIN; ①
BEGIN

# insert into reposongs_song(id, title, artist, released)
values (7,'Sledgehammer','Peter Gabriel', '1986-05-18');
INSERT 0 1

# select * from reposongs_song;
id |        title         |     artist      |  released  | foo
----+----------------------+-----------------+------------+-----
6 | Don't Worry Be Happy | Bobby McFerrin | 1988-08-05 |
7 | Sledgehammer         | Peter Gabriel  | 1986-05-18 | ②
```

```
(3 rows)
```

① new transaction started when BEGIN command issued

② commands within a transaction will be able to see uncommitted changes from the same transaction

8.8.2. ROLLBACK Transaction Example

The following shows how the previous command(s) in the current transaction can be rolled back — as if they never executed. The transaction ends once we issue COMMIT or ROLLBACK.

ROLLBACK Example

```
# ROLLBACK; ①
ROLLBACK

# select * from reposongs_song; ②
 id |        title         |     artist     |  released
----+----------------------+----------------+------------
  6 | Don't Worry Be Happy | Bobby McFerrin | 1988-08-05
```

① transaction ends once COMMIT or ROLLBACK command issued

② commands outside of a transaction will not be able to see uncommitted and rolled back changes of another transaction

[1] "ACID Wikipedia Page", Wikipedia

Chapter 9. JDBC

With database schema in place and a key amount of SQL under our belt, it is time to move on to programmatically interacting with the database. Our next stop is a foundational aspect of any Java database interaction, the Java Database Connectivity (JDBC) API. JDBC is a standard Java API for communicating with tabular databases. [1] We hopefully will never need to write this code in our applications, but it eventually gets called by any database mapping layers we may use — therefore it is good to know some of the foundation.

9.1. JDBC DataSource

The `javax.sql.DataSource` is the starting point for interacting with the database. Assuming we have Flyway schema migrations working at startup, we already know we have our database properties setup properly. It is now our chance to inject a `DataSource` and do some work.

The following snippet shows an example of an injected `DataSource`. That `DataSource` is being used to obtain the URL used to connect to the database. Most JDBC commands declare a checked exception (`SQLException`) that must be caught or also declared thrown.

Injecting a DataSource

```
@Component
@RequiredArgsConstructor
public class JdbcSongDAO {
    private final javax.sql.DataSource dataSource; ①

    @PostConstruct
    public void init() {
        try {
            String url = dataSource.getConnection().getMetaData().getURL();
            ... ②
        } catch (SQLException ex) { ③
            throw new IllegalStateException(ex);
        }
    }
```

① `DataSource` injected using constructor injection

② `DataSource` used to obtain a connection and metadata for the URL

③ All/most JDBC commands declare throwing a SQLException that must be explicitly handled

9.2. Obtain Connection and Statement

We obtain a `java.sql.Connection` from the `DataSource` and a `Statement` from the connection. Connections and statements must be closed when complete and we can automated that with a Java try-with-resources statement. `PreparedStatement` can be used to assemble the statement up front and reused in a loop if appropriate.

```
public void create(Song song) throws SQLException {
    String sql = //insert/select/delete/update ... ①

    try(Connection conn = dataSource.getConnection(); ②
        PreparedStatement statement = conn.prepareStatement(sql)) {

        //statement.executeUpdate(); ③
        //statement.executeQuery();
    }
}
```

① action-specific SQL will be supplied to the `PreparedStatement`

② try-with-resources construct automatically closes objects declared at this scope

③ `Statement` used to query and modify database

9.3. JDBC Create Example

JDBC Create Example

```
public void create(Song song) throws SQLException {
    String sql = "insert into REPOSONGS_SONG(id, title, artist, released)
values(?,?,?,?)";①

    try(Connection conn = dataSource.getConnection();
        PreparedStatement statement = conn.prepareStatement(sql)) {
        int id = nextId(conn); //get next ID from database ②
        log.info("{}, params={}", sql, List.of(id, song.getTitle(), song.getArtist(),
song.getReleased()));

        statement.setInt(1, id); ③
        statement.setString(2, song.getTitle());
        statement.setString(3, song.getArtist());
        statement.setDate(4, Date.valueOf(song.getReleased()));
        statement.executeUpdate();

        setId(song, id); //inject ID into supplied instance ④
    }
}
```

① SQL commands have `?` placeholders for parameters

② leveraging a helper method (based on a query statement) to obtain next sequence value

③ filling in the individual variables of the SQL template

④ leveraging a helper method (based on Java reflection) to set the generated ID of the instance before returning

Use Variables over String Literal Values

> Repeated SQL commands should always use parameters over literal values. Identical SQL templates allow database parsers to recognize a repeated command and leverage earlier query plans. Unique SQL strings require database to always parse the command and come up with new plans.

9.4. Set ID Example

The following snippet shows the helper method used earlier to set the ID of an existing instance. We need the helper because `id` is declared private. `id` is declared private and without a setter because it should never change. Persistence is one of the exceptions to "should never change".

Example Helper Method to Set Private ID of instance

```
private void setId(Song song, int id) {
    try {
        Field f = Song.class.getDeclaredField("id"); ①
        f.setAccessible(true); ②
        f.set(song, id); ③
    } catch (NoSuchFieldException | IllegalAccessException ex) {
        throw new IllegalStateException("unable to set Song.id", ex);
    }
}
```

① using Java reflection to locate the `id` field of the `Song` class

② must set to accessible since `id` is private — otherwise an `IllegalAccessException`

③ setting the value of the `id` field

9.5. JDBC Select Example

The following snippet shows an examle of using a JDBC select. In this case we are querying the database and representing the returned rows as instances of `Song` POJOs.

JDBC Select Example

```
public Song findById(int id) throws SQLException {
    String sql = "select title, artist, released from REPOSONGS_SONG where id=?"; ①

    try(Connection conn = dataSource.getConnection();
        PreparedStatement statement = conn.prepareStatement(sql)) {
        statement.setInt(1, id); ②
        try (ResultSet rs = statement.executeQuery()) { ③
            if (rs.next()) { ④
                Date releaseDate = rs.getDate(3); ⑤
                return Song.builder()
                    .id(id)
                    .title(rs.getString(1))
                    .artist(rs.getString(2))
                    .released(releaseDate == null ? null : releaseDate.toLocalDate())
                    .build();
            } else {
                throw new NoSuchElementException(String.format("song[%d] not found",
id));
            }
        }
    }
}
```

① provide a SQL template with `?` placeholders for runtime variables

② fill in variable placeholders

③ execute query and process results in one or more `ResultSet` — which must be closed when complete

④ must test `ResultSet` before obtaining first and each subsequent row

⑤ obtain values from the `ResultSet` — numerical order is based on SELECT clause

9.6. nextId

The `nextId()` call from `createSong()` is another query on the surface, but it is incrementing a sequence at the database level to supply the value.

nextId

```
private int nextId(Connection conn) throws SQLException {
    String sql = dialect.getNextvalSql();
    try(PreparedStatement call = conn.prepareStatement(sql)) {
        try (ResultSet rs = call.executeQuery()) {
            if (rs.next()) {
                Long id = rs.getLong(1);
                return id.intValue();
            } else {
                throw new IllegalStateException("no sequence result returned from
call");
            }
        }
    }
}
```

9.7. Dialect

Sequences syntax (and support for Sequences) is often DB-specific. Therefore, if we are working at the SQL or JDBC level, we need to use the proper dialect for our target database. The following snippet shows two choices for dialect for getting the next value for a sequence.

Dialect

```
    private Dialect dialect;
    enum Dialect {
        H2("call next value for hibernate_sequence"),
        POSTGRES("select nextval('hibernate_sequence')");
        private String nextvalSql;
        private Dialect(String nextvalSql) {
            this.nextvalSql = nextvalSql;
        }
        String getNextvalSql() { return nextvalSql; }
    }
```

[1] "JDBC Tutorial", tutorialspoint.com

Chapter 10. Summary

In this module we learned:

- to define a relational database schema for a table, columns, sequence, and index
- to define a primary key, table constraints, and an index
- to automate the creation and migration of the database schema
- to interact with database tables and columns with SQL
- underlying JDBC API interactions