Database Connectivity With JDBC

Today's Topic

A common requirement for most of the applications that we use is that they need to maintain **persistent state**. This means that certain interactions with the application have lasting effects that can be recalled hours, days, or weeks later.

Examples:

- Order history at Amazon.com
- LinkedIn profile information
- Email messages in GMail

Today's Topic

This data often needs to be searched and updated in order for the application to fulfill its purpose. One of the most common ways an application stores persistent data is by using a database.

We've already seen how we can interact with a database directly by typing SQL commands into a GUI client (i.e. PgAdmin). Today we'll learn how to write application code that can interact with a database in order to read and write persistent data.

What We'll Cover Today

- What JDBC is and how Java abstracts database functionality
- Managing database connections
- Using SpingJdbc to simplify cumbersome aspects of JDBC
- How to create and use SpringJdbc's JdbcTemplate to query and update data
- How to use query parameters and why we should do so
- What the DAO pattern is and why it is useful

So Many Database Flavors... What To Do?

- Application code that we write to interact with a database is a **client** of the database in the same way PgAdmin is.

- There are many different database vendors (e.g. PostgreSQL, SQL Server, Oracle, etc) that a Java application may want to integrate with.

- Each vendor's database implementation is likely to be quite different which would make it a huge task to switch to a different database vendor (which actually happens fairly frequently).

JDBC To The Rescue!

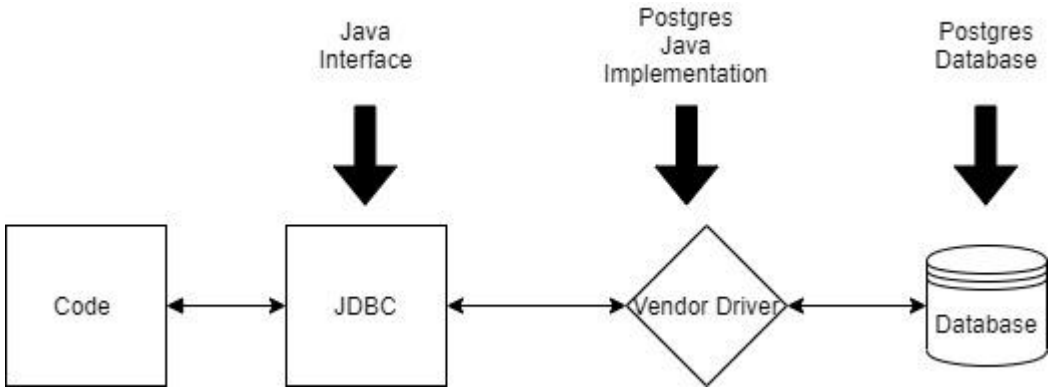
- Java uses a database interface called **JDBC** (Java Database Connectivity) to abstract database operations away from the actual database implementation.

- Each vendor provides an implementation of the interface specific to its code.

- The vendor's implementation is known as a **driver**.
  .

JDBC To The Rescue!

- We use the JDBC interface to communicate with the vendor's driver.
- This makes it fairly easy to swap out one vendor's database for another by swapping out the JDBC vendor driver.



Managing Connections

When we interact with a database, we need to create a connection.

- Connections remain open until they are closed or time out.
- Connections have overhead when created and opened, thus there is often a finite number of connections.
- A **connection pool** can be used to reuse a few connections to conserve resources within an application by allowing the application to acquire a connection and release it when it is no longer needed so it can be reused.

Managing Connections

We use a **connection string**, which specifies the name of the driver to use, the host and any port, and the database name, along with the username and password to create a datasource in Java.

```
// Create BasicDataSource
BasicDataSource dataSource = new BasicDataSource();
dataSource.setUrl("jdbc:postgresql://localhost:5432/MovieDB");
dataSource.setUsername("postgres");
dataSource.setPassword("postgres1");
```

Connection String

Managing Connections

We use a **connection string**, which specifies the name of the driver to use, the host and any port, and the database name, along with the username and password to create a datasource in Java.

```
// Create BasicDataSource
BasicDataSource dataSource = new BasicDataSource();
dataSource.setUrl("jdbc:postgresql://localhost:5432/MovieDB");
dataSource.setUsername("postgres");
dataSource.setPassword("postgres1");
```

Connection String

Username

Managing Connections

We use a **connection string**, which specifies the name of the driver to use, the host and any port, and the database name, along with the username and password to create a datasource in Java.

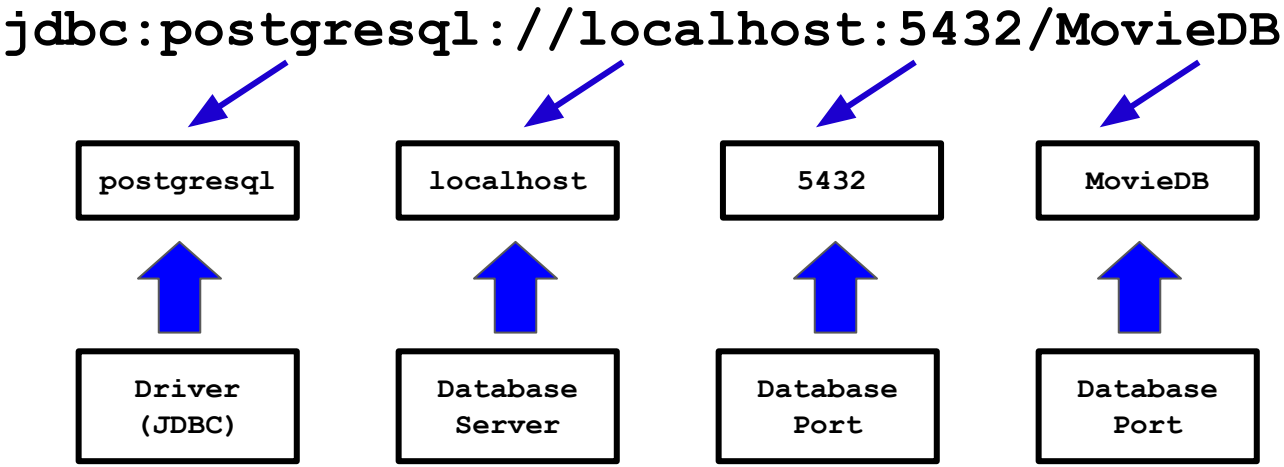
```
// Create BasicDataSource
BasicDataSource dataSource = new BasicDataSource();
dataSource.setUrl("jdbc:postgresql://localhost:5432/MovieDB");
dataSource.setUsername("postgres");
dataSource.setPassword("postgres1");
```

Connection String

Password

Username

Breaking Down the Connection String



Managing Connections

- Connection strings should not be written directly in our code. Why?

- Connections are valuable resources. It may not seem like a big deal if we leave it running in our single application, but what about a larger-scale application?

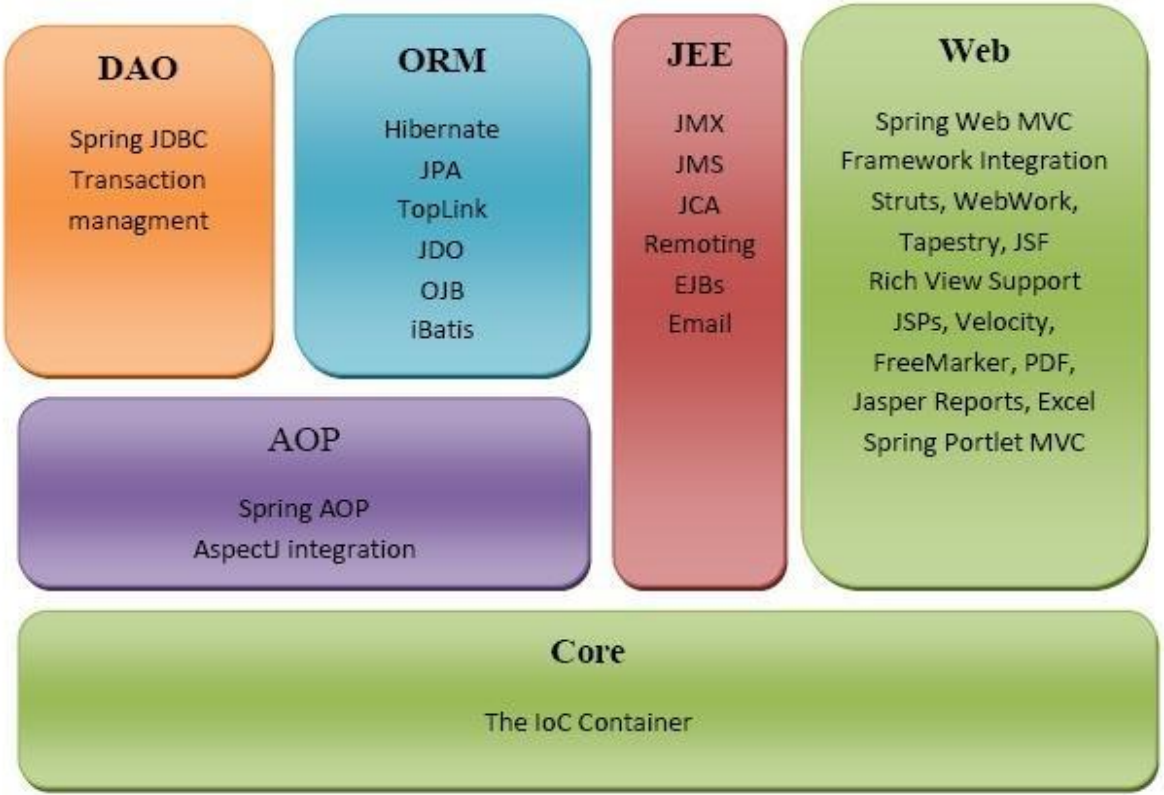
Working With JDBC

The JDBC interface provides a standard way of working with databases in Java, but using it can be very cumbersome:

- In order to accomplish things you need to rewrite the same boilerplate code over and over.
- You must pay attention to cleaning up properly (closing database connections for instance).
- There are many exceptions that can occur when working with JDBC and handling them properly can result in a lot of extra code.

Spring JDBC to the Rescue!

Spring is a popular Java framework. It is made up of many modules including **Spring JDBC**, which is is intended to solve some of the problems with using JDBC we mentioned before.

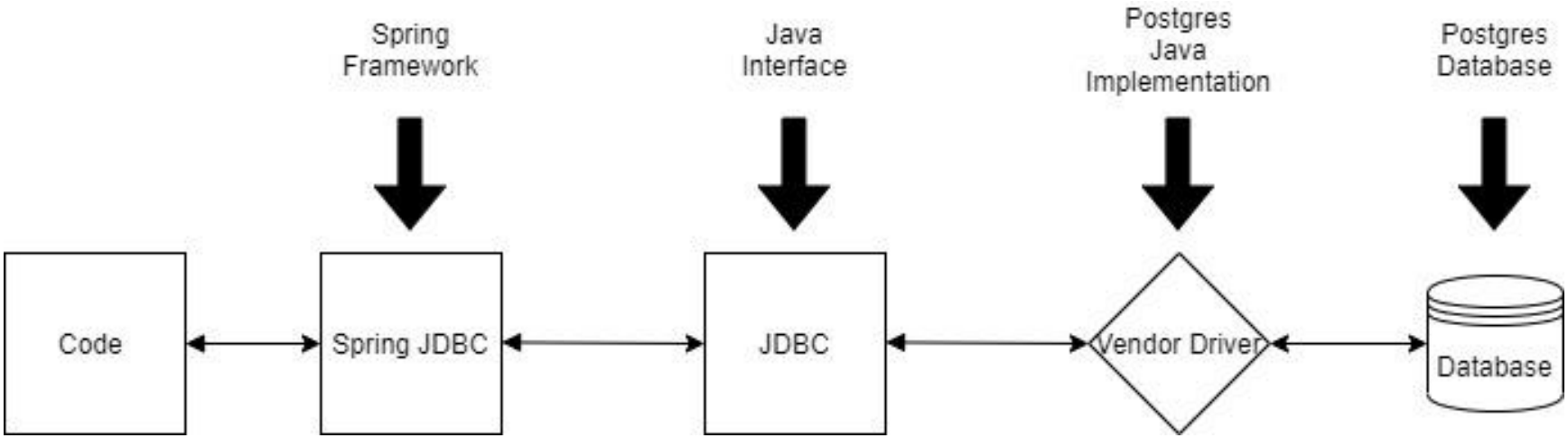


Spring JDBC

Spring JDBC makes things easier:

- Abstracts away boilerplate code so you can focus on YOUR code rather than all the nuts and bolts of interfacing with JDBC.
- Does most of the clean-up “automagically” for you.
- Simplifies SQL Exceptions in a way that makes them much easier to handle without a lot of extra code.
- Greatly simplifies working with Transactions.

Spring JDBC


Spring
Framework
Java
Interface
Postgres
Java
Implementation
Postgres
Database
Code
Spring JDBC
JDBC
Vendor Driver
Database

Diving Into Code

Introducing... JdbcTemplate

The Spring JDBC **`JdbcTemplate`** class provides methods for working with many aspects of JDBC all in one class.

- **`queryForRowSet`** is used to **`SELECT`** data sets from the database
- **`queryForObject`** is used to **`SELECT`** a single value from the database
- **`update`** is used when we don't need to return **`SELECT`**ed data.
    - **`INSERT`**
    - **`UPDATE`**
    - **`DELETE`**

Creating a JdbcTemplate

```
// Create BasicDataSource (remember that diagram?)
BasicDataSource dataSource = new BasicDataSource();
dataSource.setUrl("jdbc:postgresql://localhost:5432/MovieDB");
dataSource.setUsername("postgres");
dataSource.setPassword("postgres1");

// Create a JdbcTemplate using the Datasource
JdbcTemplate jdbcTemplate =
    new JdbcTemplate(dataSource);
```

Creating a JdbcTemplate

Datasource creation code we saw before

```
// Create BasicDataSource (remember that diagram?)
BasicDataSource dataSource = new BasicDataSource();
dataSource.setUrl("jdbc:postgresql://localhost:5432/MovieDB");
dataSource.setUsername("postgres");
dataSource.setPassword("postgres1");

// Create a JdbcTemplate using the Datasource
JdbcTemplate jdbcTemplate =
    new JdbcTemplate(dataSource);
```

Creating a JdbcTemplate

```
// Create BasicDataSource (remember that diagram?)
BasicDataSource dataSource = new BasicDataSource();
dataSource.setUrl("jdbc:postgresql://localhost:5432/MovieDB");
dataSource.setUsername("postgres");
dataSource.setPassword("postgres1");

// Create a JdbcTemplate using the Datasource
JdbcTemplate jdbcTemplate =
    new JdbcTemplate(dataSource);
```

`Datasource` creation code we saw before

Create **`JdbcTemplate`** with the **`datasource`** as the parameter

Selecting Data with JdbcTemplate

The JdbcTemplate queryForRowSet method allows you to pass a string that contains the SQL of the query and get back a SqlRowSet, which represents the rows returned by the queries. We'll look at how to work with SQLRowSets shortly…

```
SqlRowSet movies = jdbcTemplate.queryForRowSet("SELECT
movie_id, title FROM movie");
```

Selecting Data with JdbcTemplate

The JdbcTemplate queryForRowSet method allows you to pass a string that contains the SQL of the query and get back a SqlRowSet, which represents the rows returned by the queries. We'll look at how to work with SQLRowSets shortly…

The result will be returned in a `SqlRowSet` object.

```
SqlRowSet movies = jdbcTemplate.queryForRowSet("SELECT
movie_id, title FROM movie");
```

Selecting Data with JdbcTemplate

The JdbcTemplate queryForRowSet method allows you to pass a string that contains the SQL of the query and get back a SqlRowSet, which represents the rows returned by the queries. We'll look at how to work with SQLRowSets shortly…

The result will be returned in a `SqlRowSet` object.

`queryForRowSet` is used to get back a data set.

```
SqlRowSet movies = jdbcTemplate.queryForRowSet("SELECT
movie_id, title FROM movie");
```

Selecting Data with JdbcTemplate

The JdbcTemplate queryForRowSet method allows you to pass a string that contains the SQL of the query and get back a SqlRowSet, which represents the rows returned by the queries. We'll look at how to work with SQLRowSets shortly…

The result will be returned in a **`SqlRowSet`** object.

`queryForRowSet` is used to get back a data set.

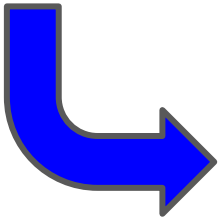
```
SqlRowSet movies = jdbcTemplate.queryForRowSet("SELECT
movie_id, title FROM movie");
```

The parameter is a **`String`** with the query SQL.

Working with SqlRowSets

A SQLRowSet contains a data set representing the results of our query.

```
SqlRowSet movies =
    jdbcTemplate.queryForRowSet("SELECT movie_id, title FROM movie");
```



movie_id [PK] integer	title character varying (200)
155	The Dark Knight
122	The Lord of the Rings: The Return of the King

Working with SqlRowSets

```
SqlRowSet movies =
    jdbcTemplate.queryForRowSet("SELECT
    movie_id, title FROM movie");
```

movie_id [PK] integer	title character varying (200)
155	The Dark Knight
122	The Lord of the Rings: The Return of the King
324857	Spider-Man: Into the Spider-Verse
1891	The Empire Strikes Back
120	The Lord of the Rings: The Fellowship of the Ring
27205	Inception
121	The Lord of the Rings: The Two Towers
101	Léon: The Professional

`results` will return a set of rows that uses a **cursor** to move through the data. Each time we call the `SqlRowSet` method `next()`, the `SqlRowSet` checks if there is more data and, if so, **moves the cursor** to next `SqlRow` in the set **and returns** `true`. Otherwise, it **returns** `false`.

Working with SqlRowSets

```
SqlRowSet movies =
    jdbcTemplate.queryForRowSet("SELECT
    movie_id, title FROM movie");
```

movie_id [PK] integer	title character varying (200)
155	The Dark Knight
122	The Lord of the Rings: The Return of the King
324857	Spider-Man: Into the Spider-Verse
1891	The Empire Strikes Back
120	The Lord of the Rings: The Fellowship of the Ring
27205	Inception
121	The Lord of the Rings: The Two Towers
101	Léon: The Professional

Cursor ***does NOT start at beginning of data***. You must call the **next()** method to get to first row.

`results` will return a set of rows that uses a **cursor** to move through the data. Each time we call the `SqlRowSet` method `next()`, the `SqlRowSet` checks if there is more data and, if so, **moves the cursor** to next `SqlRow` in the set **and returns** `true`. Otherwise, it **returns** `false`.

Getting Data From SqlRowSet

Remember that we must call **`next()`** method in order to advance to the **FIRST** row as well.

As the next() method is called, the cursor moves forward and represents the next row in the set.

We can query the current row by using get methods like getLong, getInt, and getString that take the column name as a parameter.

```
while(movies.next()) {
    String title = movies.getString("title");
    Long id = movies.getLong("movie_id");
    System.out.println(id + " " + title);
}
```

Getting Data From SqlRowSet

Remember that we must call **`next()`** method in order to advance to the **FIRST** row as well.

As the next() method is called, the cursor moves forward and represents the next row in the set.

We can query the current row by using get methods like getLong, getInt, and getString that take the column name as a parameter.

The **`next()`** method advances the cursor to the next row. If a row exists, it returns **`true`**, otherwise **`false`**

```
while(movies.next()) {
    String title = movies.getString("title");
    Long id = movies.getLong("movie_id");
    System.out.println(id + " " + title);
}
```

Getting Data From SqlRowSet

Remember that we must call **`next()`** method in order to advance to the **FIRST** row as well.

As the next() method is called, the cursor moves forward and represents the next row in the set.

We can query the current row by using get methods like getLong, getInt, and getString that take the column name as a parameter.

The **`next()`** method advances the cursor to the next row. If a row exists, it returns **`true`**, otherwise **`false`**

Get the current row's **`title`** column as a **`String`**

```
while(movies.next()) {
    String title = movies.getString("title");
    Long id = movies.getLong("movie_id");
    System.out.println(id + " " + title);
}
```

Getting Data From SqlRowSet

Remember that we must call **`next()`** method in order to advance to the **FIRST** row as well.

As the next() method is called, the cursor moves forward and represents the next row in the set.

We can query the current row by using get methods like getLong, getInt, and getString that take the column name as a parameter.

The **`next()`** method advances the cursor to the next row. If a row exists, it returns **`true`**, otherwise **`false`**

Get the current row's **`title`** column as a **`String`**

```
while(movies.next()) {
    String title = movies.getString("title");
    Long id = movies.getLong("movie_id");
    System.out.println(id + " " + title);
}
```

Get the current row's **`movie_id`** column as a **`Long`**

Using Query Params

The **`queryForRowSet`** method can take parameterized queries. We replace each value in the query with a ? and then provide a list of parameters to populate the data. This makes our queries reusable but also helps prevent SQL injection, which can be a huge security risk (more on this shortly…)

Here is how we use parameters with queryForRowSet:

```
// with queryForRowSet
SqlRowSet movies = jdbcTemplate.queryForRowSet("SELECT title
    FROM movie WHERE movie_id = ?", movieId);
```

Using Query Params

The **queryForRowSet** method can take parameterized queries. We replace each value in the query with a ? and then provide a list of parameters to populate the data. This makes our queries reusable but also helps prevent SQL injection, which can be a huge security risk (more on this shortly…)

Here is how we use parameters with queryForRowSet:

```
// with queryForRowSet
SqlRowSet movies = jdbcTemplate.queryForRowSet("SELECT title
    FROM movie WHERE movie_id = ?", movieId);
```

The ? represents a query parameter that will be provided.

Using Query Params

The **queryForRowSet** method can take parameterized queries. We replace each value in the query with a ? and then provide a list of parameters to populate the data. This makes our queries reusable but also helps prevent SQL injection, which can be a huge security risk (more on this shortly…)

Here is how we use parameters with queryForRowSet:

```
// with queryForRowSet
SqlRowSet movies = jdbcTemplate.queryForRowSet("SELECT title
    FROM movie WHERE movie_id = ?", movieId);
```

The ? represents a query parameter that will be provided.

The **parameters following the SQL replace the ?'s in the query**

Using Query Params vs Concatenation

It may be tempting to use String concatenation rather than using query params. For instance:

```
// with queryForRowSet
SqlRowSet results =
    jdbcTemplate.queryForRowSet("SELECT title FROM movie
    WHERE movie_id = " + id);
```

However, using **concatenation** can open us up to SQL Injection which is a major security issue and should be avoided. If we concatenate we have no way of ensuring that the param passed in is actually data and not malicious SQL code that will be executed with the query. Hackers can exploit injecting code instead of data. By using params, we allow Java to make sure all params are treated as data and not as code.

Using Query Params vs Concatenation

It may be tempting to use String concatenation rather than using query params. For instance:

Concatenation should be avoided. Use query parameters instead.

```
// with queryForRowSet
SqlRowSet results =
    jdbcTemplate.queryForRowSet("SELECT title FROM movie
    WHERE movie_id = " + id);
```

However, using **concatenation** can open us up to SQL Injection which is a major security issue and should be avoided. If we concatenate we have no way of ensuring that the param passed in is actually data and not malicious SQL code that will be executed with the query. Hackers can exploit injecting code instead of data. By using params, we allow Java to make sure all params are treated as data and not as code.

Selecting A Single Value with JdbcTemplate

The JdbcTemplate queryForObject method allows you to pass a string that contains the SQL of the query and get back a single object, which will be an object that can hold the type of data you are querying for.

```
Integer count = jdbcTemplate.queryForObject("SELECT
    COUNT(*) FROM movie", Integer.class);
```

Selecting A Single Value with JdbcTemplate

The JdbcTemplate queryForObject method allows you to pass a string that contains the SQL of the query and get back a single object, which will be an object that can hold the type of data you are querying for.

The result will be returned in an `Integer` object.

```
Integer count = jdbcTemplate.queryForObject("SELECT
    COUNT(*) FROM movie", Integer.class);
```

Selecting A Single Value with JdbcTemplate

The JdbcTemplate queryForObject method allows you to pass a string that contains the SQL of the query and get back a single object, which will be an object that can hold the type of data you are querying for.

The result will be returned in an `Integer` object.

`queryForObject` is used to get back a single object.

```
Integer count = jdbcTemplate.queryForObject("SELECT
    COUNT(*) FROM movie", Integer.class);
```

Selecting A Single Value with JdbcTemplate

The JdbcTemplate queryForObject method allows you to pass a string that contains the SQL of the query and get back a single object, which will be an object that can hold the type of data you are querying for.

The result will be returned in an `Integer` object.

`queryForObject` is used to get back a single object.

```
Integer count = jdbcTemplate.queryForObject("SELECT
    COUNT(*) FROM movie", Integer.class);
```

The first parameter is a `String` with the query SQL.

Selecting A Single Value with JdbcTemplate

The JdbcTemplate queryForObject method allows you to pass a string that contains the SQL of the query and get back a single object, which will be an object that can hold the type of data you are querying for.

The result will be returned in an `Integer` object.

`queryForObject` is used to get back a single object.

```
Integer count = jdbcTemplate.queryForObject("SELECT
    COUNT(*) FROM movie", Integer.class);
```

The first parameter is a `String` with the query SQL.

The second parameter is the type of class to return as result. The `.class` extension indicates a class type.

Using Query Params with queryForObject

The **`queryForObject`** method of JdbcTemplate can also use parameterized queries. The only difference from the other methods is that the parameter indicating the class type to return comes before the query parameters,

```
String title = jdbcTemplate.queryForObject("SELECT title
FROM movie WHERE movie_id = ?", String.class, movieId);
```

Using Query Params with queryForObject

The **`queryForObject`** method of JdbcTemplate can also use parameterized queries. The only difference from the other methods is that the parameter indicating the class type to return comes before the query parameters,

```
String title = jdbcTemplate.queryForObject("SELECT title
FROM movie WHERE movie_id = ?", String.class, movieId);
```

The `?` represents a query parameter that will be provided.

Using Query Params with queryForObject

The **`queryForObject`** method of JdbcTemplate can also use parameterized queries. The only difference from the other methods is that the parameter indicating the class type to return comes before the query parameters,

```
String title = jdbcTemplate.queryForObject("SELECT title
FROM movie WHERE movie_id = ?", String.class, movieId);
```

The `?` represents a query parameter that will be provided.

The second parameter is the type of class to return as result. The `.class` extension indicates a class type.

Using Query Params with queryForObject

The **`queryForObject`** method of JdbcTemplate can also use parameterized queries. The only difference from the other methods is that the parameter indicating the class type to return comes before the query parameters,

```
String title = jdbcTemplate.queryForObject("SELECT title
FROM movie WHERE movie_id = ?", String.class, movieId);
```

The `?` represents a query parameter that will be provided.

The second parameter is the type of class to return as result. The `.class` extension indicates a class type.

The parameters to replace the ?'s in the query follow the class type parameter

Working with Like/ILike

When using **`LIKE`** (or **`ILIKE`** which is a postgreSQL extension that ignores case), you must specify any wildcards in the params rather than in the query.

```
SqlRowSet title = jdbcTemplate.queryForRowSet("SELECT title
    FROM movie WHERE title ILIKE ?", startsWith + "%");
```

The `%` is concatenated onto the end of the **`startsWith`** param. Using `%` in the SQL **`String`** (as in "**`WHERE title ILIKE '?%'`**" or "**`WHERE title ILIKE ? + '%'`**") won't work because of the special significance of th **?** character. Concatenation in a parameter is ok - just not in the SQL String itself.

Updating Data with JdbcTemplate

The JdbcTemplate **update** method allows you to pass a string that contains the SQL of an update statement which will be executed in the database.

```
jdbcTemplate.update("UPDATE movie SET length_minutes =
    300 WHERE movie_id = 11");
```

Updating Data with JdbcTemplate

The JdbcTemplate **update** method allows you to pass a string that contains the SQL of an update statement which will be executed in the database.

```
jdbcTemplate.update("UPDATE movie SET length_minutes =
    300 WHERE movie_id = 11");
```

The `update` method takes the `String` with the SQL as a parameter

Using Query Parameters with Update

We can use query parameters with update the same way we do with queryForRowSet:

```
jdbcTemplate.update("UPDATE movie SET length_minutes =
    300 WHERE movie_id = ?", id);
```

Using Query Parameters with Update

We can use query parameters with update the same way we do with queryForRowSet:

```
jdbcTemplate.update("UPDATE movie SET length_minutes =
    300 WHERE movie_id = ?", id);
```

The `update` method takes the `String` with the SQL as a parameter

Using Query Parameters with Update

We can use query parameters with update the same way we do with queryForRowSet:

```
jdbcTemplate.update("UPDATE movie SET length_minutes =
    300 WHERE movie_id = ?", id);
```

The `?` represents a query parameter that will be provided.

The `update` method takes the `String` with the SQL as a parameter

Using Query Parameters with Update

We can use query parameters with update the same way we do with queryForRowSet:

```
jdbcTemplate.update("UPDATE movie SET length_minutes =
    300 WHERE movie_id = ?", id);
```

The `?` represents a query parameter that will be provided.

The parameters to replace the ?'s in the query follow the class type parameter

The `update` method takes the `String` with the SQL as a parameter

Introducing DTOs

Introducing… DTOs

Data Transfer Objects are also known as **DTO**s (as well as many other names… ever heard of a POJO?)

These contain the data that represents some business concept in our code (such as City in the UnitedStates database).

Usually, these will only have data members and getters/setters.

Creating a DTO from SqlRowSet Row

We can use the SqlRowSet get methods to read all the columns of a row and use them to populate a DTO that has meaning in our code.

```
City city = new City();
city.setCityId(rowSet.getLong("city_id"));
city.setCityName(rowSet.getString("city_name"));
city.setStateAbbreviation(rowSet.getString(
    "state_abbreviation"));
```

Creating a DTO from SqlRowSet Row

We can use the SqlRowSet get methods to read all the columns of a row and use them to populate a DTO that has meaning in our code.

Create a `City` object

```
City city = new City();
city.setCityId(rowSet.getLong("city_id"));
city.setCityName(rowSet.getString("city_name"));
city.setStateAbbreviation(rowSet.getString(
    "state_abbreviation"));
```

Creating a DTO from SqlRowSet Row

We can use the SqlRowSet get methods to read all the columns of a row and use them to populate a DTO that has meaning in our code.

Create a `City` object

Set the `City` object's `id` data member by calling the appropriate `rowSet` get method (`getLong` in this case because `id` is a `long`) with the column name as a parameter

```
City city = new City();
city.setCityId(rowSet.getLong("city_id"));
city.setCityName(rowSet.getString("city_name"));
city.setStateAbbreviation(rowSet.getString(
    "state_abbreviation"));
```

Creating a DTO from SqlRowSet Row

We can use the SqlRowSet get methods to read all the columns of a row and use them to populate a DTO that has meaning in our code.

Create a `City` object

Set the `City` object's `id` data member by calling the appropriate `rowSet` get method (`getLong` in this case because `id` is a `long`) with the column name as a parameter

```
City city = new City();
city.setCityId(rowSet.getLong("city_id"));
city.setCityName(rowSet.getString("city_name"));
city.setStateAbbreviation(rowSet.getString(
    "state_abbreviation"));
```

Set the `City` object's `name` data member using `rowSet.getString` in the same manner

Creating a Mapping Method

We can create a method that can be called to map the current row in the SqlRowSet to a DTO.

```
private City mapRowToCity(SqlRowSet rowSet) {
    City city = new City();
    city.setCityId(rowSet.getLong("city_id"));
    city.setCityName(rowSet.getString("city_name"));
    city.setStateAbbreviation(rowSet.getString("state_abbreviation"));
    city.setPopulation(rowSet.getLong("population"));
    city.setArea(rowSet.getDouble("area"));
    return city;
}
```

This is the previous code in the context of a method that takes a `SqlRowSet` and returns a `City`.

Working with SqlRowSet and DTOs

```
public List<City> getUsaCities() {

     // query code that returns a SqlRowSet

     // need to ADVANCE cursor to get FIRST row
     while(resultSet.next()) {
          // use our method mapRowToCity to map a
          // SqlRow to a City object - we'll see
          // how
          City city =
          mapRowToCity(resultSet);
          if (city  != null) {
               result.add(city );
          }
     }
     return result;
}
```

Cursor ***does NOT start at beginning of data***. You must call the **next()** method to get to first row.

id	name	countrycode	district	population
3793	New York	USA	New York	8008278
3794	Los Angeles	USA	California	3694820
3795	Chicago	USA	Illinois	2896016
3796	Houston	USA	Texas	1953631
3797	Philadelphia	USA	Pennsylvania	1517550
3798	Phoenix	USA	Arizona	1321045
3799	San Diego	USA	California	1223400
3800	Dallas	USA	Texas	1188580
3801	San Antonio	USA	Texas	1144646
3802	Detroit	USA	Michigan	951270
3803	San Jose	USA	California	894943
3804	Indianapolis	USA	Indiana	791926
3805	San Francisco	USA	California	776733

Working with SqlRowSet and DTOs

Cursor ***does NOT start at beginning of data***. You must call the **next()** method to get to first row.

Returns a `List` of `City` objects

```
public List<City> getUsaCities() {

    // query code that returns a SqlRowSet

    // need to ADVANCE cursor to get FIRST row
    while(resultSet.next()) {
        // use our method mapRowToCity to map a
        // SqlRow to a City object - we'll see
        // how
        City city =
        mapRowToCity(resultSet);
        if (city  != null) {
            result.add(city );
        }
    }
    return result;
}
```

id	name	countrycode	district	population
3793	New York	USA	New York	8008278
3794	Los Angeles	USA	California	3694820
3795	Chicago	USA	Illinois	2896016
3796	Houston	USA	Texas	1953631
3797	Philadelphia	USA	Pennsylvania	1517550
3798	Phoenix	USA	Arizona	1321045
3799	San Diego	USA	California	1223400
3800	Dallas	USA	Texas	1188580
3801	San Antonio	USA	Texas	1144646
3802	Detroit	USA	Michigan	951270
3803	San Jose	USA	California	894943
3804	Indianapolis	USA	Indiana	791926
3805	San Francisco	USA	California	776733

Working with SqlRowSet and DTOs

```
public List<City> getUsaCities() {

    // query code that returns a SqlRowSet

    // need to ADVANCE cursor to get FIRST row
    while(resultSet.next()) {
        // use our method mapRowToCity to map a
        // SqlRow to a City object - we'll see
        // how
        City city =
        mapRowToCity(resultSet);
        if (city  != null) {
            result.add(city );
        }
    }
    return result;
}
```

Returns a `List` of `City` objects

`resultSet` contains our `SqlRowSet`

Cursor ***does NOT start at beginning of data***. You must call the **next()** method to get to first row.

id	name	countrycode	district	population
3793	New York	USA	New York	8008278
3794	Los Angeles	USA	California	3694820
3795	Chicago	USA	Illinois	2896016
3796	Houston	USA	Texas	1953631
3797	Philadelphia	USA	Pennsylvania	1517550
3798	Phoenix	USA	Arizona	1321045
3799	San Diego	USA	California	1223400
3800	Dallas	USA	Texas	1188580
3801	San Antonio	USA	Texas	1144646
3802	Detroit	USA	Michigan	951270
3803	San Jose	USA	California	894943
3804	Indianapolis	USA	Indiana	791926
3805	San Francisco	USA	California	776733

Working with SqlRowSet and DTOs

Cursor ***does NOT start at beginning of data***. You must call the **next()** method to get to first row.

```
public List<City> getUsaCities() {

    // query code that returns a SqlRowSet

    // need to ADVANCE cursor to get FIRST row
    while(resultSet.next()) {
        // use our method mapRowToCity to map a
        // SqlRow to a City object - we'll see
        // how
        City city =
        mapRowToCity(resultSet);
        if (city  != null) {
            result.add(city );
        }
    }
    return result;
}
```

Returns a `List` of `City` objects

`resultSet` contains our `SqlRowSet`

Call `next()` to get to first/next record

id	name	countrycode	district	population
3793	New York	USA	New York	8008278
3794	Los Angeles	USA	California	3694820
3795	Chicago	USA	Illinois	2896016
3796	Houston	USA	Texas	1953631
3797	Philadelphia	USA	Pennsylvania	1517550
3798	Phoenix	USA	Arizona	1321045
3799	San Diego	USA	California	1223400
3800	Dallas	USA	Texas	1188580
3801	San Antonio	USA	Texas	1144646
3802	Detroit	USA	Michigan	951270
3803	San Jose	USA	California	894943
3804	Indianapolis	USA	Indiana	791926
3805	San Francisco	USA	California	776733

Working with SqlRowSet and DTOs

Cursor ***does NOT start at beginning of data***. You must call the **next()** method to get to first row.

```
public List<City> getUsaCities() {

    // query code that returns a SqlRowSet

    // need to ADVANCE cursor to get FIRST row
    while(resultSet.next()) {
        // use our method mapRowToCity to map a
        // SqlRow to a City object - we'll see
        // how
        City city =
        mapRowToCity(resultSet);
        if (city  != null) {
            result.add(city );
        }
    }
    return result;
}
```

Returns a `List` of `City` objects

Call `next()` to get to first/next record

`resultSet` contains our `SqlRowSet`

Create `City` object using the mapping method we created..

id	name	countrycode	district	population
3793	New York	USA	New York	8008278
3794	Los Angeles	USA	California	3694820
3795	Chicago	USA	Illinois	2896016
3796	Houston	USA	Texas	1953631
3797	Philadelphia	USA	Pennsylvania	1517550
3798	Phoenix	USA	Arizona	1321045
3799	San Diego	USA	California	1223400
3800	Dallas	USA	Texas	1188580
3801	San Antonio	USA	Texas	1144646
3802	Detroit	USA	Michigan	951270
3803	San Jose	USA	California	894943
3804	Indianapolis	USA	Indiana	791926
3805	San Francisco	USA	California	776733

Working with SqlRowSet and DTOs

Cursor ***does NOT start at beginning of data***. You must call the **next()** method to get to first row.

```
public List<City> getUsaCities() {

    // query code that returns a SqlRowSet

    // need to ADVANCE cursor to get FIRST row
    while(resultSet.next()) {
        // use our method mapRowToCity to map a
        // SqlRow to a City object - we'll see
        // how
        City city =
        mapRowToCity(resultSet);
        if (city  != null) {
            result.add(city);
        }
    }
    return result;
}
```

Returns a `List` of `City` objects

Call `next()` to get to first/next record

`resultSet` contains our `SqlRowSet`

Create `City` object using the mapping method we created..

Add `City` object to result `List`

id	name	countrycode	district	population
3793	New York	USA	New York	8008278
3794	Los Angeles	USA	California	3694820
3795	Chicago	USA	Illinois	2896016
3796	Houston	USA	Texas	1953631
3797	Philadelphia	USA	Pennsylvania	1517550
3798	Phoenix	USA	Arizona	1321045
3799	San Diego	USA	California	1223400
3800	Dallas	USA	Texas	1188580
3801	San Antonio	USA	Texas	1144646
3802	Detroit	USA	Michigan	951270
3803	San Jose	USA	California	894943
3804	Indianapolis	USA	Indiana	791926
3805	San Francisco	USA	California	776733

Working with SqlRowSet and DTOs

Cursor ***does NOT start at beginning of data***. You must call the **next()** method to get to first row.

```
public List<City> getUsaCities() {

    // query code that returns a SqlRowSet

    // need to ADVANCE cursor to get FIRST row
    while(resultSet.next()) {
        // use our method mapRowToCity to map a
        // SqlRow to a City object - we'll see
        // how
        City city =
        mapRowToCity(resultSet);
        if (city  != null) {
            result.add(city);
        }
    }
    return result;
}
```

Returns a `List` of `City` objects

Call `next()` to get to first/next record

`resultSet` contains our `SqlRowSet`

Create `City` object using the mapping method we created..

Add `City` object to result `List`

Return the result `List`

id	name	countrycode	district	population
3793	New York	USA	New York	8008278
3794	Los Angeles	USA	California	3694820
3795	Chicago	USA	Illinois	2896016
3796	Houston	USA	Texas	1953631
3797	Philadelphia	USA	Pennsylvania	1517550
3798	Phoenix	USA	Arizona	1321045
3799	San Diego	USA	California	1223400
3800	Dallas	USA	Texas	1188580
3801	San Antonio	USA	Texas	1144646
3802	Detroit	USA	Michigan	951270
3803	San Jose	USA	California	894943
3804	Indianapolis	USA	Indiana	791926
3805	San Francisco	USA	California	776733

Getting the ID of an Object Being Inserted

When inserting a record, it is often important to get the id that was generated by PostgreSQL. We can do this using the INSERT … RETURNING syntax.

```
public City createCity(City city) {
      String sql = "INSERT INTO city (city_name, state_abbreviation,
            population, area) " +
            "VALUES (?, ?, ?, ?) RETURNING city_id;";

      Long newId = jdbcTemplate.queryForObject(sql, Long.class,
            city.getCityName(), city.getStateAbbreviation(),
            city.getPopulation(), city.getArea());

      city.setCityId(newId);
      return city;
}
```

Getting the ID of an Object Being Inserted

When inserting a record, it is often important to get the id that was generated by PostgreSQL. We can do this using the INSERT … RETURNING syntax.

The **INSERT** ends with **RETURNING** and the field being returned (**city_id**)..

```
public City createCity(City city) {
      String sql = "INSERT INTO city (city_name, state_abbreviation,
            population, area) " +
            "VALUES (?, ?, ?, ?) RETURNING city_id;";

      Long newId = jdbcTemplate.queryForObject(sql, Long.class,
            city.getCityName(), city.getStateAbbreviation(),
            city.getPopulation(), city.getArea());

      city.setCityId(newId);
      return city;
}
```

Getting the ID of an Object Being Inserted

When inserting a record, it is often important to get the id that was generated by PostgreSQL. We can do this using the INSERT … RETURNING syntax.

The **INSERT** ends with **RETURNING** and the field being returned (**city_id**)..

```
public City createCity(City city) {
      String sql = "INSERT INTO city (city_name, state_abbreviation,
            population, area) " +
            "VALUES (?, ?, ?, ?) RETURNING city_id;";

      Long newId = jdbcTemplate.queryForObject(sql, Long.class,
            city.getCityName(), city.getStateAbbreviation(),
            city.getPopulation(), city.getArea());

      city.setCityId(newId);
      return city;
}
```

We use **queryForObject** rather than **update** so we can get a result back.

Getting the ID of an Object Being Inserted

When inserting a record, it is often important to get the id that was generated by PostgreSQL. We can do this using the INSERT … RETURNING syntax.

The **INSERT** ends with **RETURNING** and the field being returned (**city_id**)..

```
public City createCity(City city) {
      String sql = "INSERT INTO city (city_name, state_abbreviation,
            population, area) " +
            "VALUES (?, ?, ?, ?) RETURNING city_id;";

      Long newId = jdbcTemplate.queryForObject(sql, Long.class,
            city.getCityName(), city.getStateAbbreviation(),
            city.getPopulation(), city.getArea());

      city.setCityId(newId);
      return city;
}
```

We use **queryForObject** rather than **update** so we can get a result back.

We update the id of the City object passed in and return it.

Introducing the DAO Pattern

Following the DAO Pattern

Although JDBC makes it easier for us to swap one database implementation for another, it often involves some internal changes to work with the features that aren't part of the ANSI SQL Standard. Sometimes we may even want to use different database implementations for different tasks (i.e. one database is Postgres and one is Oracle).

The **Data Access Object (DAO) pattern** allows us to add abstraction our own data objects so that we can write code that is not database implementation dependant.

Following the DAO Pattern

The DAO pattern uses a **data access interface** to add an abstraction layer to data objects. The pattern consists of:

- Data Access Object Interface
- Data Access Implementation Class
- Model (or Value) Objects - the DTOs or POJOs we mentioned previously.

Following the DAO Pattern

The **DAO pattern** promotes best practices and principles including:

- **Encapsulation**
  - DAO classes keep the logic for communicating with a database separate from the rest of the application logic.
- **Loosely coupled**
  - DAO interface abstracts away the specifics of the underlying data storage, so that the application and DAO have no knowledge of each other. This allows you to replace the DAO with one that accesses a different data source with little to no change to other code.
- **Single responsibility principle**
  - This principle states that every class or function of an application should have responsibility over a single part of that program's functionality. DAO classes have responsibility over a single type of object, such as a relational database table.

State DTO

We create a DTO class called State.

```
public class State {

    private String stateAbbreviation;
    private String stateName;


    public String getStateAbbreviation() {
        return stateAbbreviation;
    }

    public void setStateAbbreviation(String stateAbbreviation) {
        this.stateAbbreviation = stateAbbreviation;
    }

    public String getStateName() {
        return stateName;
    }

    public void setStateName(String stateName) {
        this.stateName = stateName;
    }
}
```

Data Access Interface

We create an interface for the **`StateDao`**.

```
public interface StateDao {

    State getState(String stateAbbreviation);

    State getStateByCapital(long cityId);

    List<State> getStates();
}
```

All objects that implement **`StateDao`** must implement these methods.

DAO Implementation Class

We create a JDBC implementation of the **`StateDao`** interface

```
public class JdbcStateDao implements StateDao {

    private final JdbcTemplate jdbcTemplate;

    public JdbcStateDao(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    // code implementing the StateDAO interface follows
```

DAO Implementation Class

The **JdbcStateDao** implements the **StateDao** interface.

We create a JDBC implementation of the **StateDao** interface

```
public class JdbcStateDao implements StateDao {

    private final JdbcTemplate jdbcTemplate;

    public JdbcStateDao(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    // code implementing the StateDAO interface follows
```

DAO Implementation Class

The **JdbcStateDao** implements the **StateDao** interface.

The constructor takes a **Datasource** which will be used to create the **JdbcTemplate**..

We create a JDBC implementation of the **StateDao** interface

```
public class JdbcStateDao implements StateDao {

    private final JdbcTemplate jdbcTemplate;

    public JdbcStateDao(DataSource dataSource) {
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

    // code implementing the StateDAO interface follows
```

Putting It All Together

Now we can use our DAO to query for all the State objects. Note that we use the **`StateDao`** interface for the dao - if we do it this way, all we have to do is swap the implementation class and all else will be the same.

```
public static void main(String[] args) {
        BasicDataSource dataSource = new BasicDataSource();
        dataSource.setUrl("jdbc:postgresql://localhost:5432/UnitedStates");
        dataSource.setUsername("postgres");
        dataSource.setPassword("postgres1");

        StateDao dao = new JdbcStateDao(dataSource);

        List<State> states = dao.getStates();

        for (State state : states) {
            System.out.println(String.format("%s %s", state.getStateAbbreviation(),
                    state.getStateName()));
        }
    }
```

Putting It All Together

Create the **`JDBCStateDao`**

Now we can use our DAO to query for all the State objects. Note that we use the **`StateDao`** interface for the dao - if we do it this way, all we have to do is swap the implementation class and all else will be the same.

```
public static void main(String[] args) {
        BasicDataSource dataSource = new BasicDataSource();
        dataSource.setUrl("jdbc:postgresql://localhost:5432/UnitedStates");
        dataSource.setUsername("postgres");
        dataSource.setPassword("postgres1");

        StateDao dao = new JdbcStateDao(dataSource);

        List<State> states = dao.getStates();

        for (State state : states) {
            System.out.println(String.format("%s %s", state.getStateAbbreviation(),
                    state.getStateName()));
        }
    }
```

Putting It All Together

Now we can use our DAO to query for all the State objects. Note that we use the **`StateDao`** interface for the dao - if we do it this way, all we have to do is swap the implementation class and all else will be the same.

Create the **`JDBCStateDao`**

```
public static void main(String[] args) {
        BasicDataSource dataSource = new BasicDataSource();
        dataSource.setUrl("jdbc:postgresql://localhost:5432/UnitedStates");
        dataSource.setUsername("postgres");
        dataSource.setPassword("postgres1");

        StateDao dao = new JdbcStateDao(dataSource);

        List<State> states = dao.getStates();

        for (State state : states) {
            System.out.println(String.format("%s %s", state.getStateAbbreviation(),
                    state.getStateName()));
        }
    }
```

Use **`StateDao`** methods

Let's Code!!!!!