

Getting Started with SQLite



Copyright 2021 Boston University. All Rights Reserved.
Authored by Warren Mansur.

Table of Contents

Introduction	3
SQLite Overview	3
Supported Platforms	3
Downloading and Installing DB Browser for SQLite	4
Step 1: Downloading DB Browser for SQLite	4
Step 2: Installing DB Browser for SQLite	5
Creating Your First Table.....	10
Step 1: Launching DB Browser for SQLite	10
Step 2: Create a New Database	11
Step 3: Adding a Table	16
Downloading the JDBC Driver	20
Connecting to your Database in Eclipse	22
Step 1: Creating a New Project	22
Step 2: Adding the JDBC Driver	23
Step 3: Inserting Rows	29
Connecting to your Database in IntelliJ.....	32
Step 1: Creating a New Project	32
Step 2: Adding the JDBC Driver	35
Step 3: Inserting Rows	38
Next Steps.....	41
Appendix A: Source Code	42
Works Cited.....	43

Introduction

This SQLite getting started guide is used by students enrolled in the Master of Science in Software Development and other Computer Science Department programs in both on-campus and online programs. The document describes SQLite, connecting to SQLite in Eclipse and IntelliJ, and working with your first table. Note that as new versions of SQLite are released, some of the screens may look different than the screenshots in this document. Nevertheless, this guide will help get you started quickly on any modern version of version of SQL Server Express.

Why is learning about databases important? Most serious applications have the need for durable storage, that is, storing information for an extended period of time. While storing information in files satisfies the needs for some applications, many require use of a database. Databases support four significant features not supported well by file systems – efficient data access amongst large sets of data, extremely granular security, highly standardized, cross-platform APIs, and structural independence. Databases support retrieving information quickly, oftentimes less than a second, from vast amounts of data. Databases support security even down to individual fields in an extensible manner. Databases have highly standardized APIs for cross-platform access. Lastly, applications that use databases are not dependent upon any particular file system or file structure. Databases provide features needed by many serious applications.

Relational databases are by far the most used databases in the world. Estimates put worldwide usage at about 77% relational, and 23% NoSQL/Search (Solid IT). When data must be shared across many clients and/or servers of an application, server-based databases are utilized, the most popular being Oracle, SQL Server, MySQL, and Postgres (Solid IT). When data does not need to be shared or the need for sharing is very limited, embedded databases can be utilized, the most popular by far being SQLite (Solid IT).

If you can't determine how to proceed or something goes wrong, and web searches don't help, ask your facilitator or instructor for help. Good luck, and have fun!

SQLite Overview

SQLite is the most used, embedded (serverless) relational database in the world. It is open source and free to use. Unlike server-based databases like Oracle and SQL Server, SQLite runs entirely in the application that uses it, and stores all of its durable objects in a single disk file. SQLite can be used across all major platforms, which means the database file can be freely copied and used across devices with difference architectures. SQLite is ideal for applications that would traditionally use files to store data, giving them access to the power of a relational database without the expense and overhead of installing and maintaining a server-based database (SQLite).

Although SQLite is not a replacement for Oracle, SQL Server, or Postgres, the good news is, once you learn to access and use any one modern relational database, you can use the others without much additional effort. All modern relational databases utilize the Structured Query Language (SQL) for data access and manipulation. SQL is highly standardized across databases. Although there are some differences, the significant aspects are the same across databases. In addition, Java supports a standardized API, JDBC, for accessing any database. Connectivity from Java does not differ much between databases. Thus SQLite is an excellent first database for Java developers, because the intricacies of relational databases and connectivity can be learned without the overhead of database installation, yet SQLite is used in serious applications worldwide.

Supported Platforms

SQLite supports all major platforms. If you are using Windows, a Mac, Linux, an Android phone, an iPhone, and some other operating systems, you can use SQLite. For the sake of brevity, the examples and screenshots in this document are for the Microsoft Windows family, including Windows 8 and Windows 10. However, please keep in mind you can follow the same steps for other operating systems; your screens may look a little different, but almost all of the steps are the same.

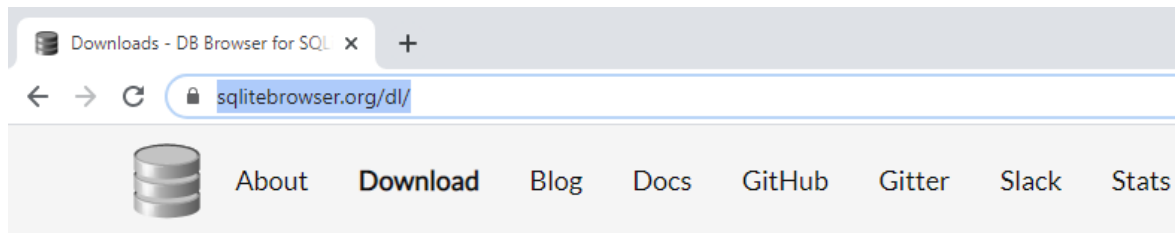
Downloading and Installing DB Browser for SQLite

It is a best practice to manage your database's structure with a SQL client. Typically, we use a SQL client to first add the tables, indexes, and triggers (if needed), as well as any initial data. Then when our application executes, it will add, modify, and remove data as needed, but not modify the structure of the tables and indexes. By separating structure manipulation from data manipulation, we can carefully apply good database design principles, and avoid embedding table structure in our application.

A popular SQL client for SQLite is DB Browser for SQLite. This section has you install the client and use it to create an initial table.

Step 1: Downloading DB Browser for SQLite

Visit Website Go to <https://sqlitebrowser.org/dl/> to get started downloading DB Browser for SQLite. The website is regularly updated, so what you see may be different than the following.



Downloads

Windows

Our latest release (3.11.2) for Windows:

- [DB Browser for SQLite - Standard installer for 32-bit Windows & Linux](#)
- [DB Browser for SQLite - .zip \(no installer\) for 32-bit Windows & Linux](#)
- [DB Browser for SQLite - Standard installer for 64-bit Windows & Linux](#)
- [DB Browser for SQLite - .zip \(no installer\) for 64-bit Windows & Linux](#)

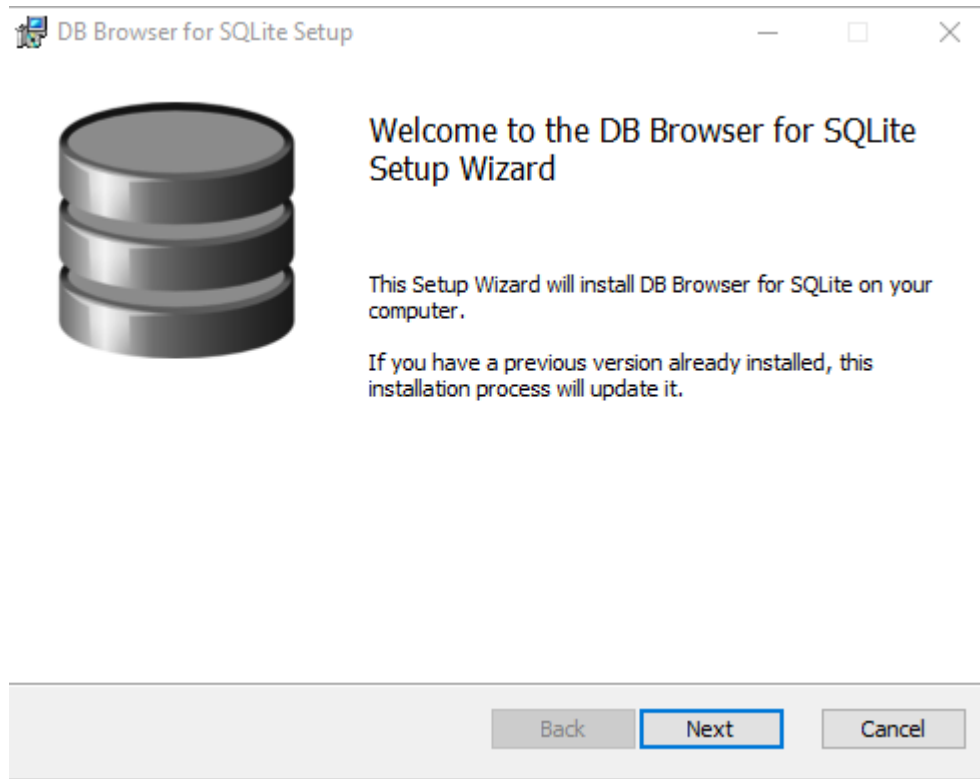
Download

Click the “Standard installer for 64-bit Windows” under the Windows downloads to start the download. If you are using a different operating system, download the appropriate installer.

Step 2: Installing DB Browser for SQLite

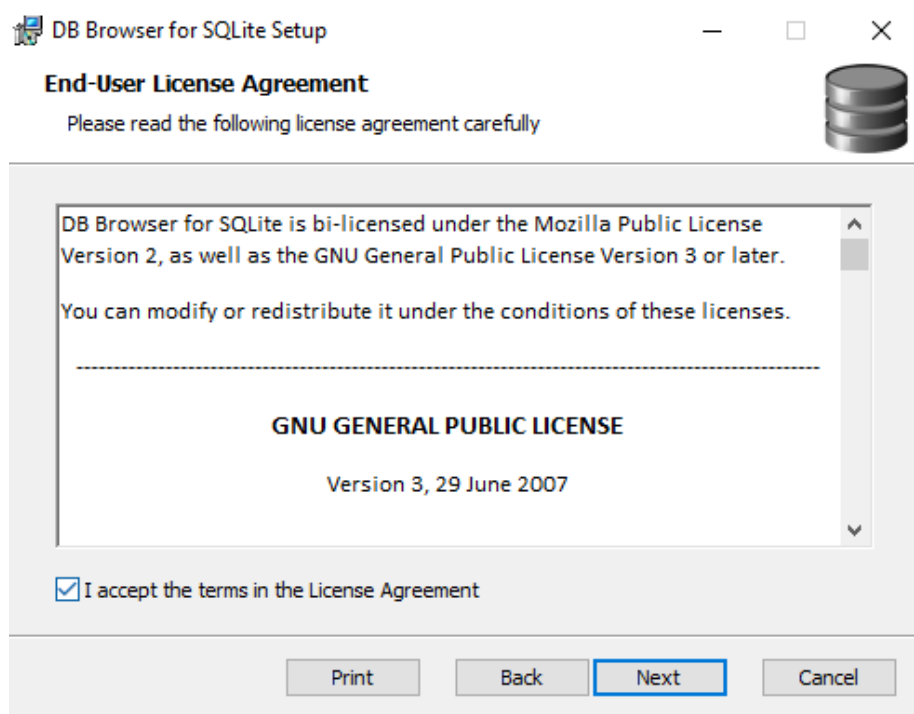
Execute Installer

Your browser will now give you the option to run the executable it downloaded. Go ahead and run it. You'll see a screen like the following.



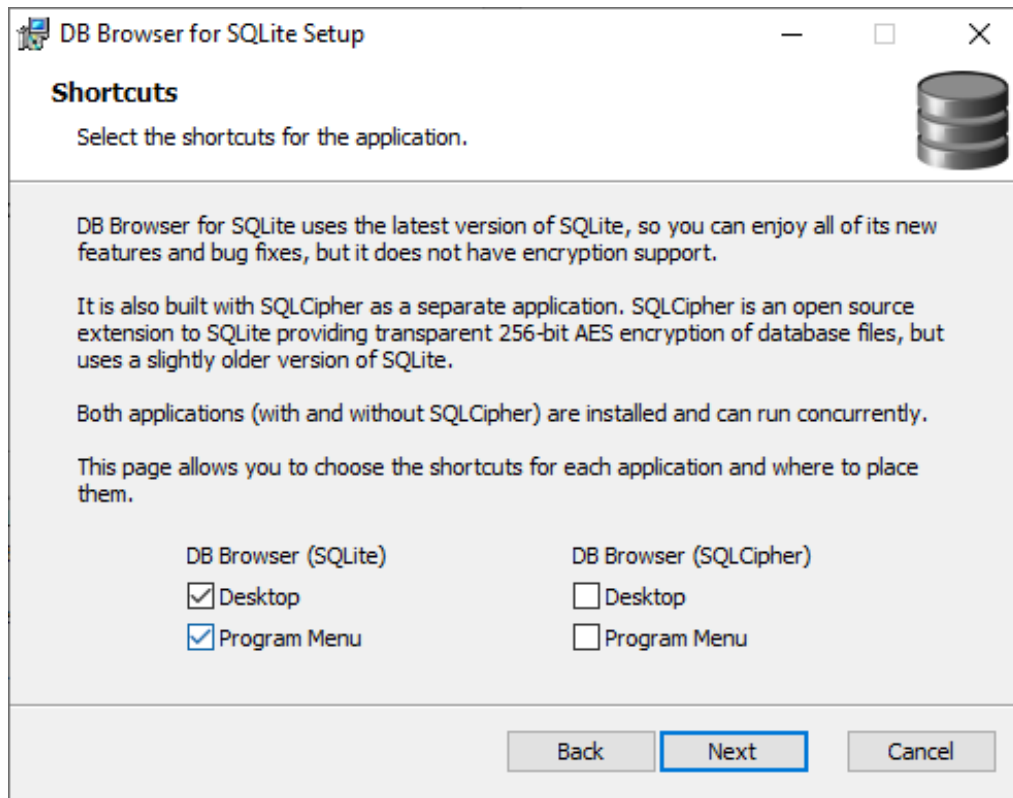
Accept License Agreement

After clicking the Next button, the installer will ask you to accept the license agreement.



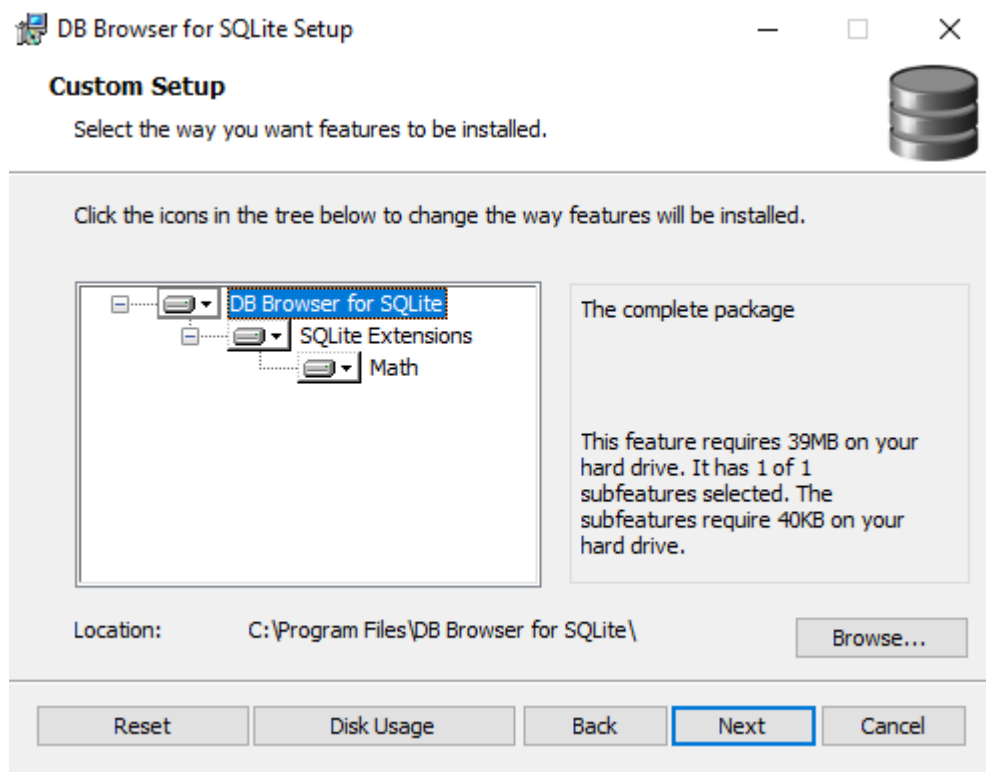
Select DB Browser Options

After clicking the Next button, you'll be asked what you would like to install. Select both the "Desktop" and "Program Menu" options under "DB Browser (SQLite)".



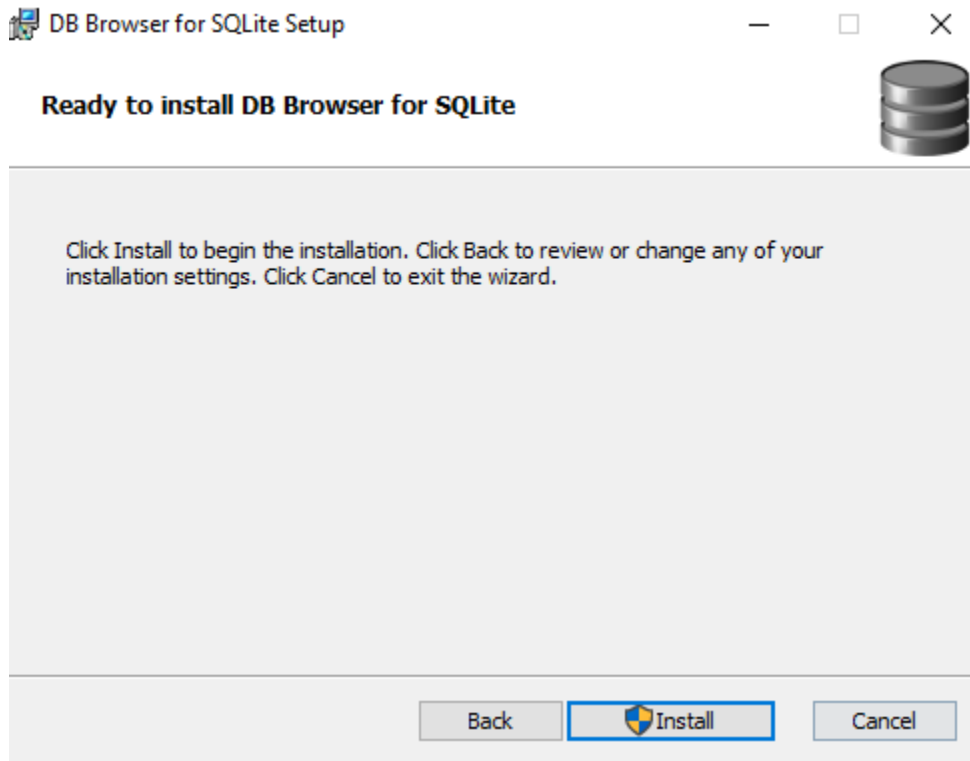
Accept Defaults

After clicking the Next button, you'll be asked if you'd like to change any install options. You don't need to change anything on this screen.

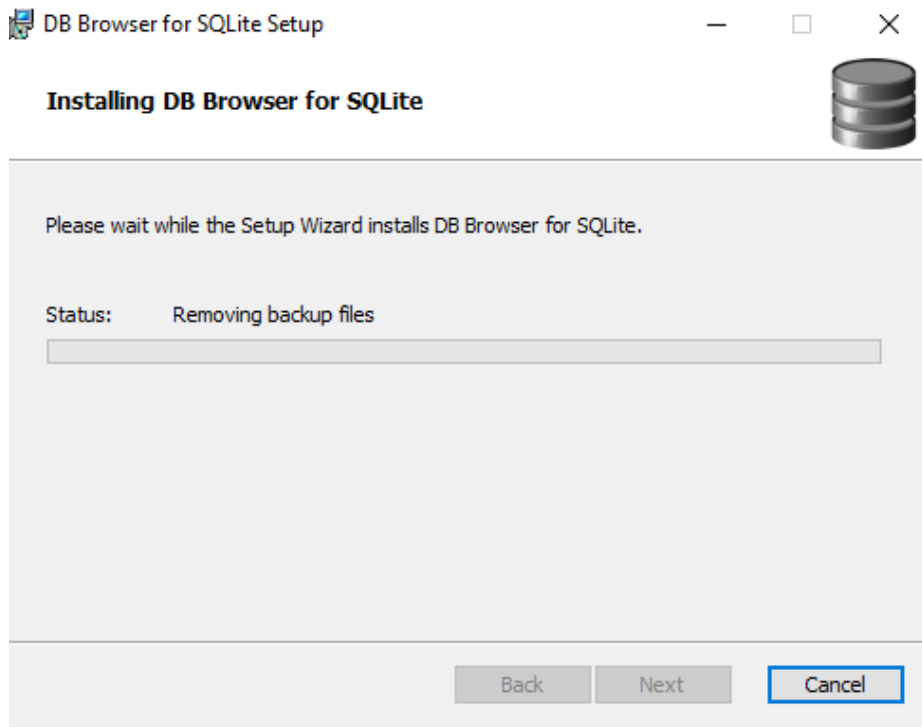


Start the Install

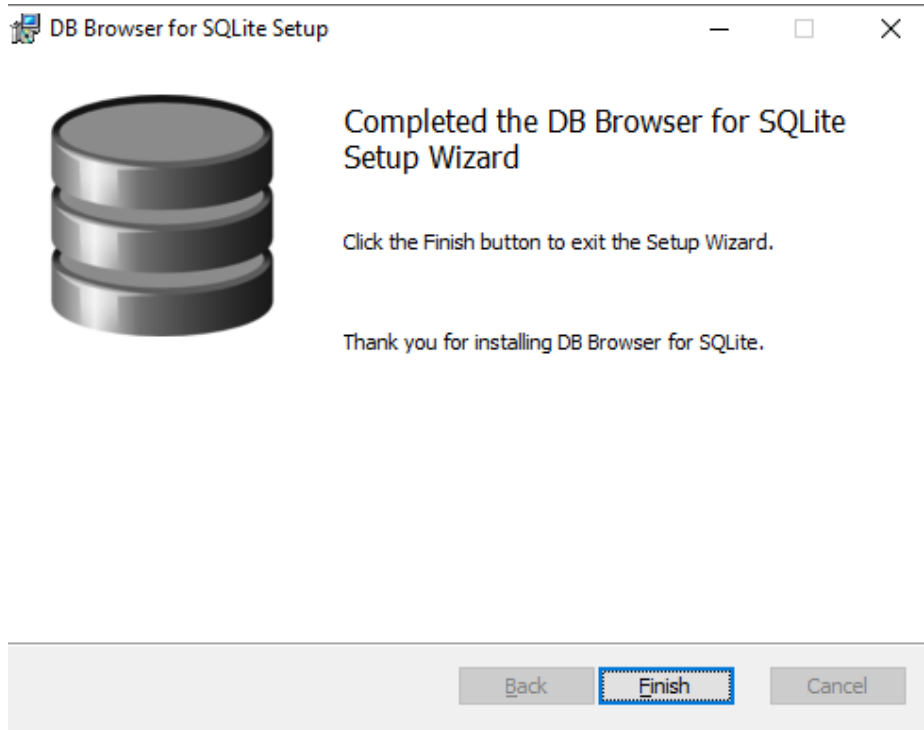
After clicking the Next button, you'll be presented with a screen that asks you to install. Go ahead and click the "Install" button.



You'll see a progress screen first.



Then you'll see a screen indicating the install was successful.



Go ahead and click the Finish button. Congratulations! DB Browser for SQLite is installed on your machine.

Creating Your First Table

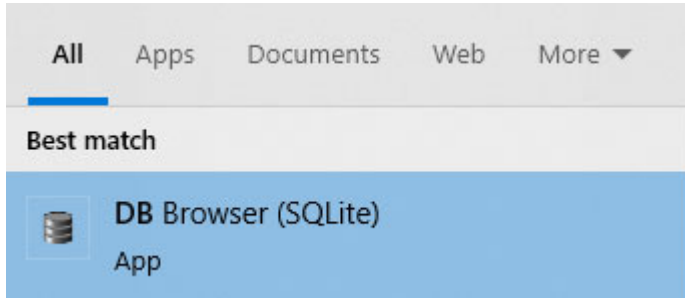
To get you started, we'll walk you through creating your first table in this section. The table will be named "Person" and will store basic information about people. In a later section we'll have you connect to your database and start working with data in the table.

Step 1: Launching DB Browser for SQLite

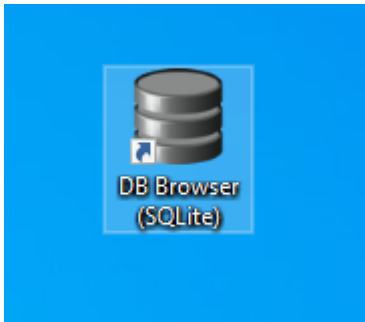
Launch from Windows

You can launch DB Browser for SQLite from Windows, either from the Start Menu, or from the shortcut on the desktop, as shown in the below screenshots.

Start Menu



Desktop

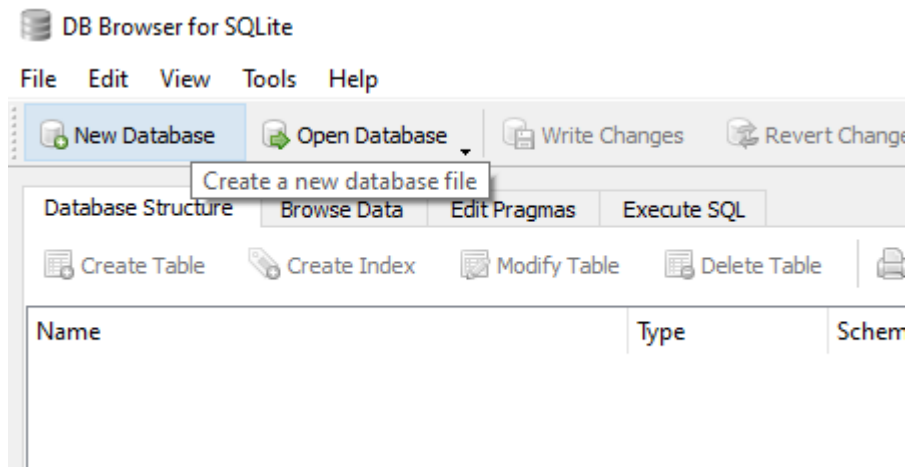


Step 2: Create a New Database

The first thing you need to do after launching the application is create a SQLite database, which will house the tables and other objects you want to work with.

Click New Database

To get started, click the “New Database” button in the top left, as shown below.



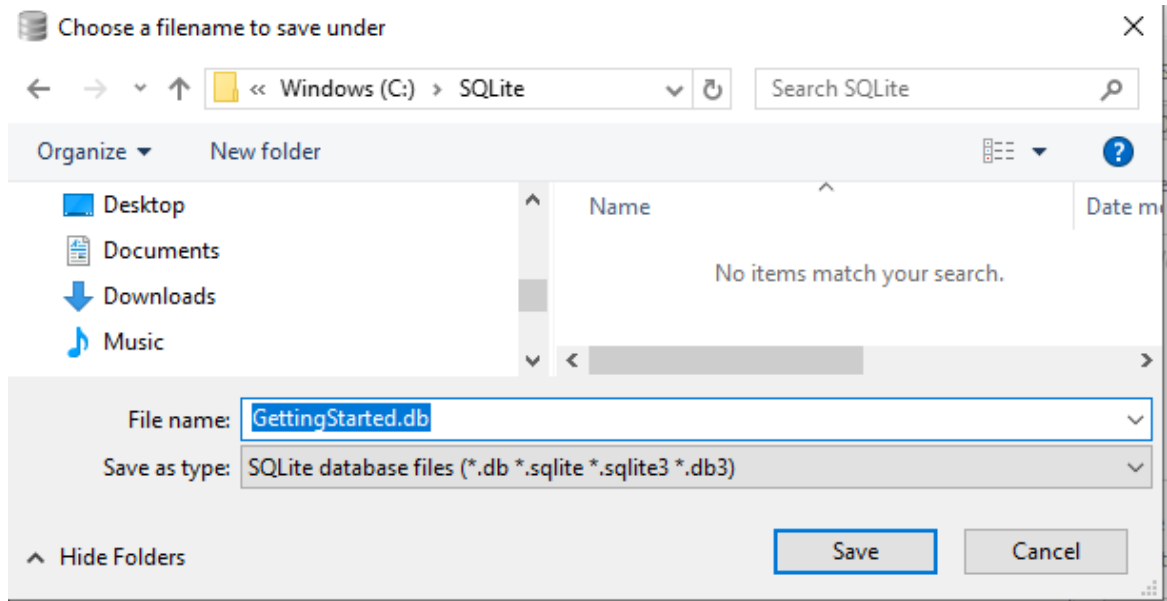
Determine Path

Now you need to choose a folder and filename. It may be necessary to create a folder to keep the database in a place you can remember.

In our example, we have created “C:\SQLite” as the folder, because it is a common location that is easy to remember. Later when you’re developing within the context of your application, you might choose to save your database as part of your Java project. In the real world, the database is commonly saved in its own subdirectory in a source code repository for the project.

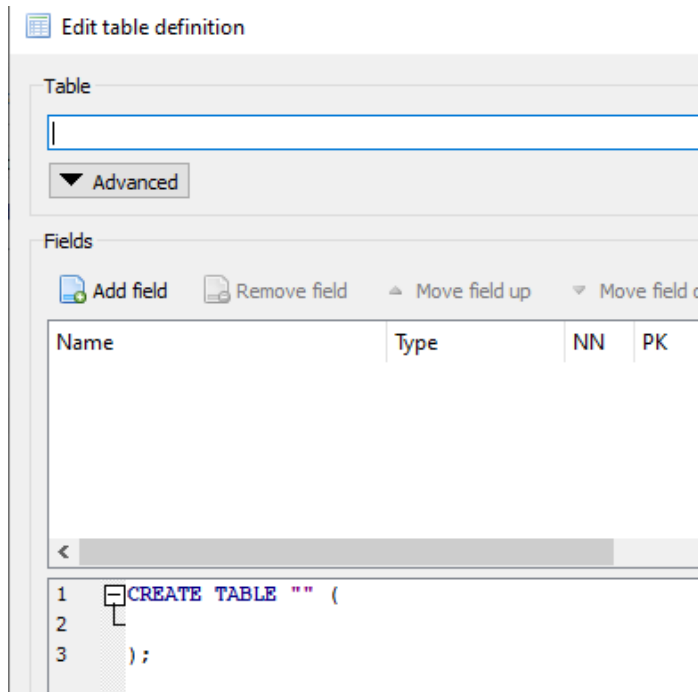
We entered “GettingStarted.db” as the filename. The “GettingStarted” name helps us know it’s for this getting started tutorial. The “.db” file extension is one of the accepted extensions for SQLite files. There are a few other accepted extensions as well; however, “.db” appears to be the most common.

This is shown in the screenshot below.



Close Table Definition Screen

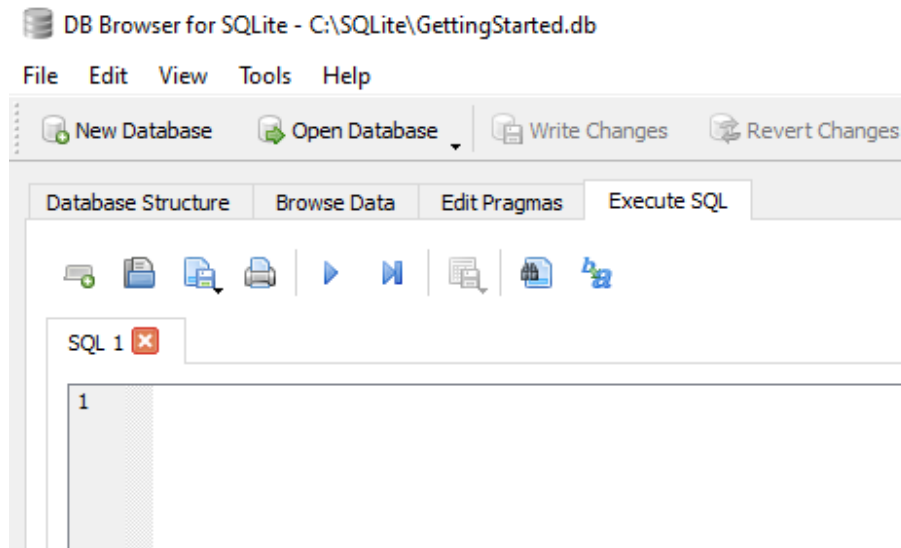
Immediately after clicking the “Save” button to create the database, a window pops up that lets us define tables using a wizard.



Simply close this window, as we will be typing in SQL directly.

Select Execute SQL Tab

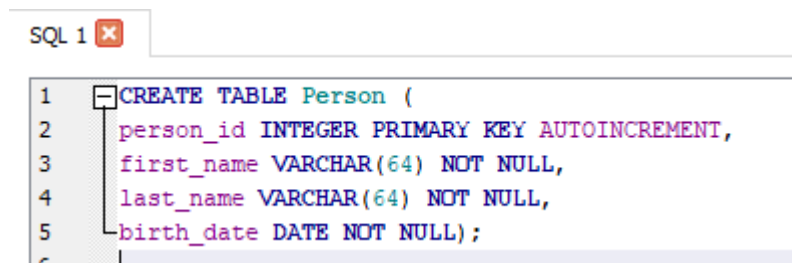
We want to type the CREATE TABLE command in SQL, so we need to click on the “Execute SQL” tab first, as shown below.




Create Table Next, we type the command to create the table. Type this command exactly:

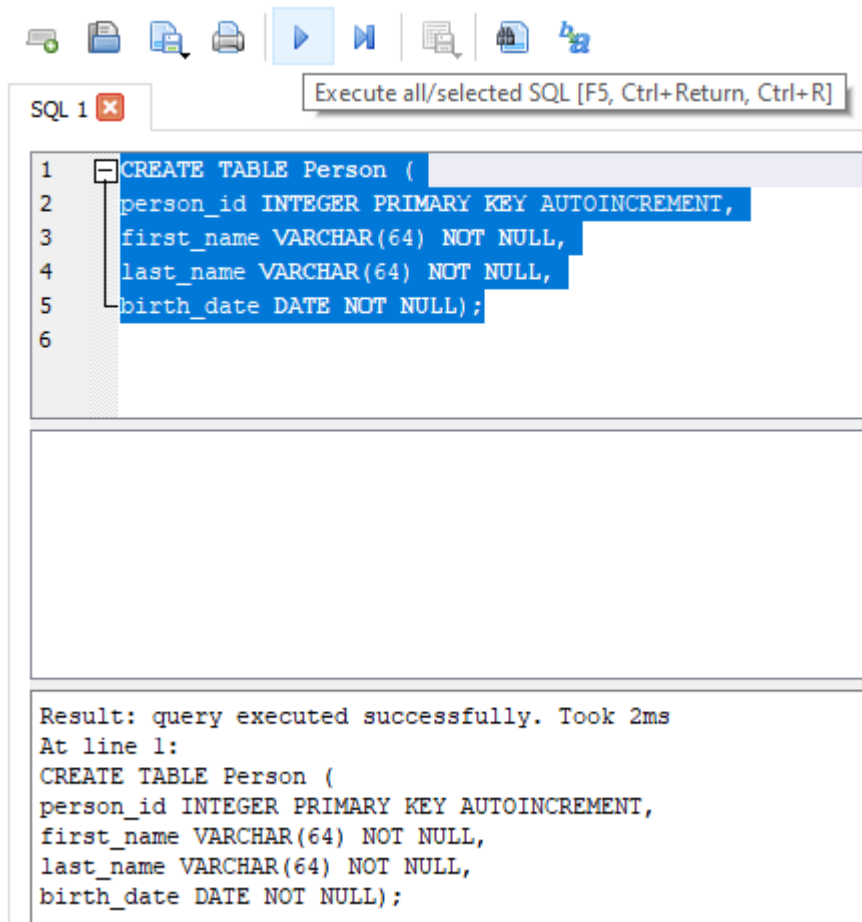
```
CREATE TABLE Person (  
person_id INTEGER PRIMARY KEY AUTOINCREMENT,  
first_name VARCHAR(64) NOT NULL,  
last_name VARCHAR(64) NOT NULL,  
birth_date DATE NOT NULL);
```

This is shown below.



This command creates a table named “Person” with a person_id autoincrementing field, first and last name fields, and a birth date field.

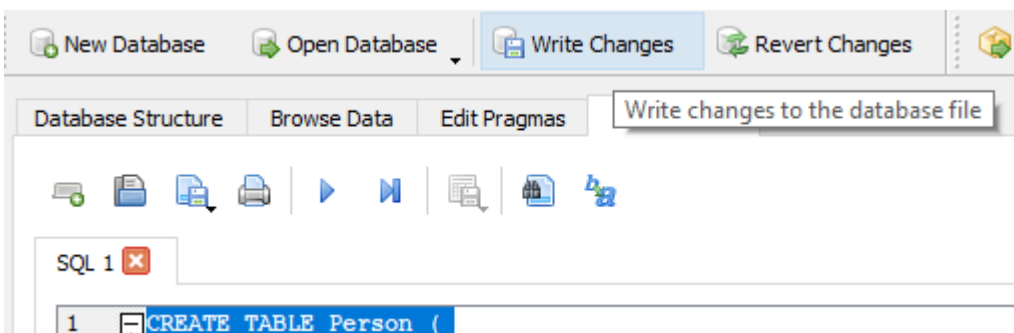
Next click the “Execute all/selected SQL” button, which looks like . After you have done so, a message will appear stating that the table has been created. This is all shown below.



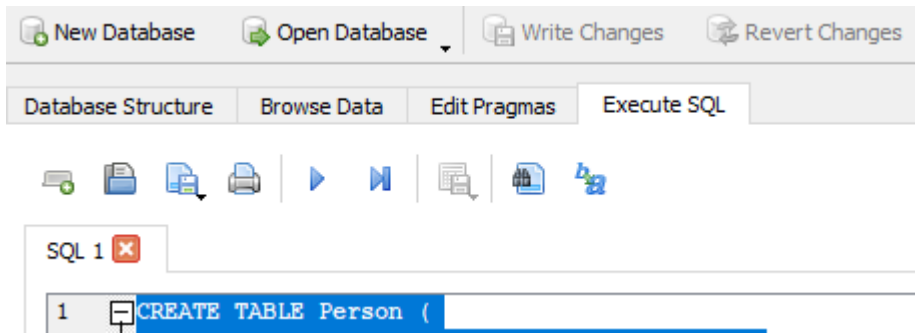
You can see that your person table was successfully created!

Save Changes

In order to save the changes to the database file, you need to click the “Write Changes” button, as shown below.



Once you have done so, the button will gray out, indicating there is nothing more to save.



You can determine if any changes need to be saved by looking at these buttons to see if they are clickable, or grayed out.

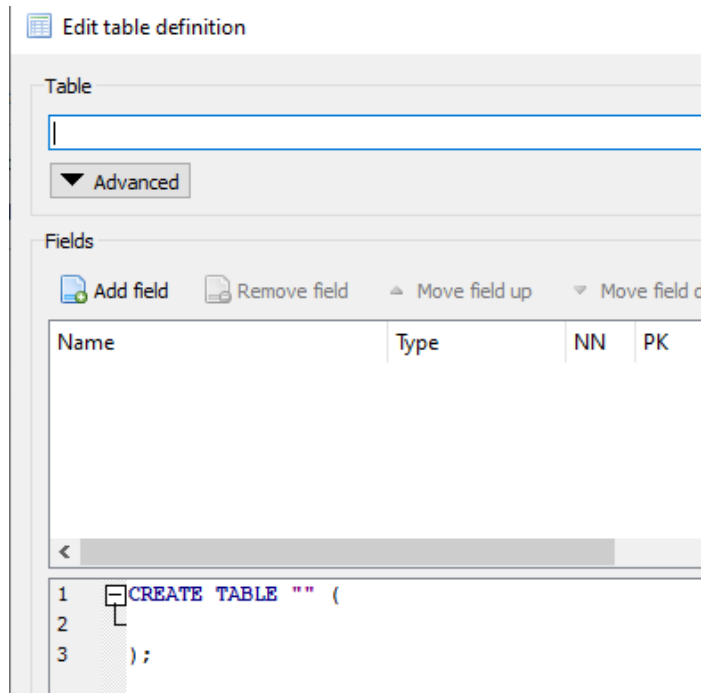
Database Ready for Application

Congratulations! You've created a SQLite database and added a table to it. The database is now ready for an application to connect to it and work with person data, which we'll do in the next section.

Step 3: Adding a Table

Close Table Definition Screen

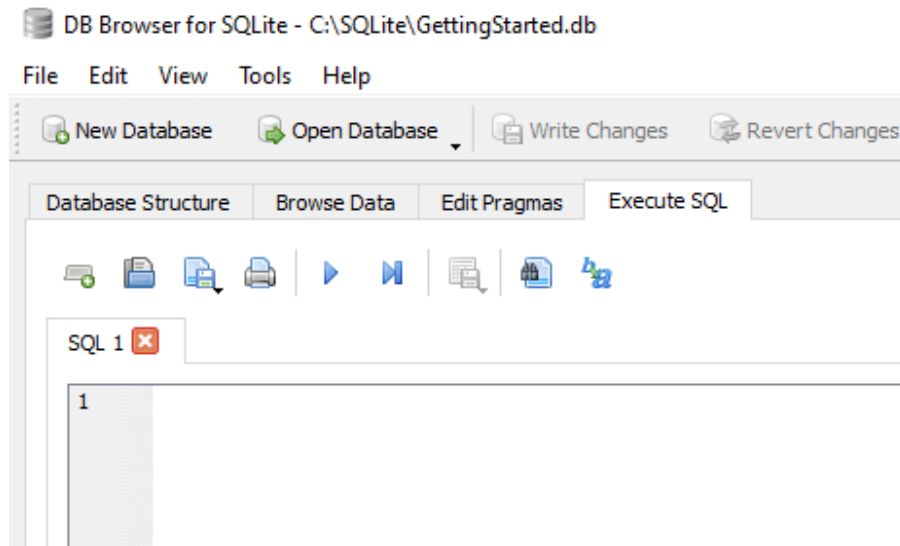
Immediately after clicking the “Save” button to create the database, a window pops up that lets us define tables using a wizard.



Simply close this window, as we will be typing in SQL directly.

Select Execute SQL Tab

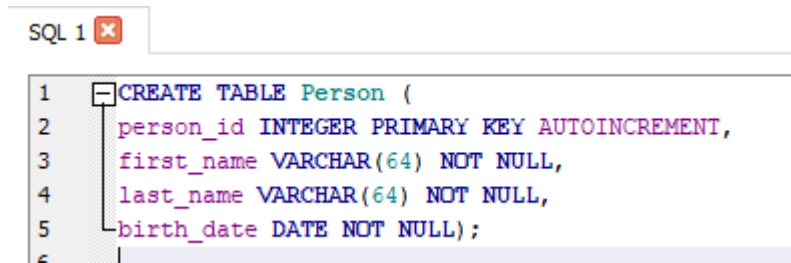
We want to type the CREATE TABLE command in SQL, so we need to click on the “Execute SQL” tab first, as shown below.



Create Table Next, we type the command to create the table. Type this command exactly:


```
CREATE TABLE Person (  
person_id INTEGER PRIMARY KEY AUTOINCREMENT,  
first_name VARCHAR(64) NOT NULL,  
last_name VARCHAR(64) NOT NULL,  
birth_date DATE NOT NULL);
```

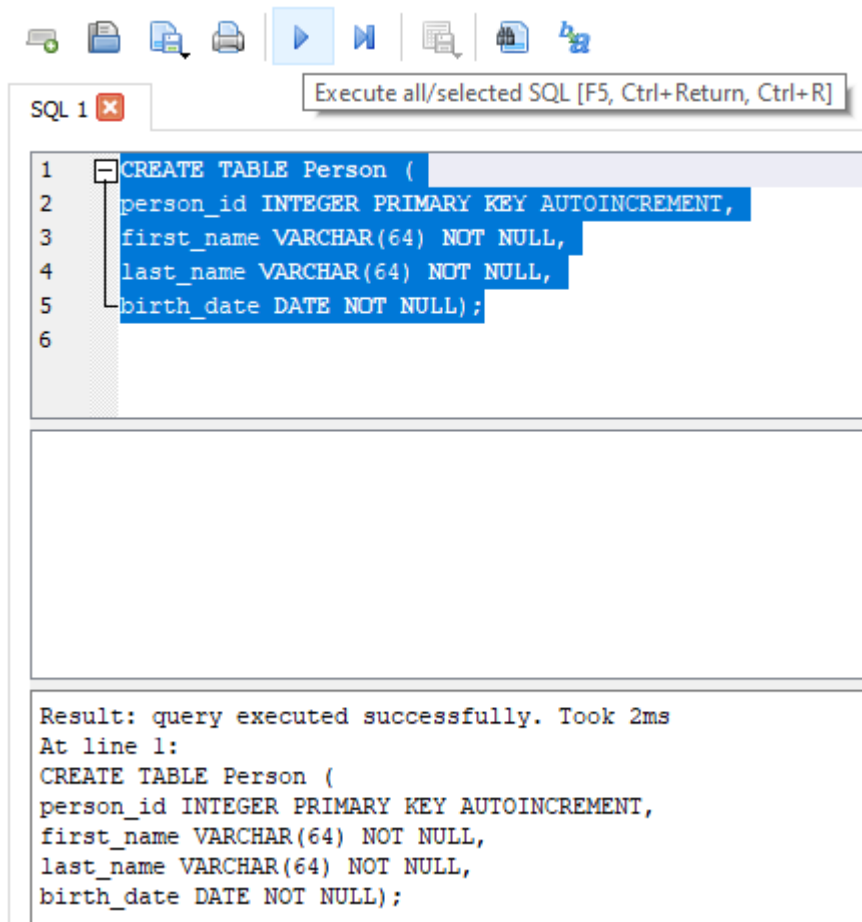
This is shown below.

A screenshot of a SQL editor window. The window title is "SQL 1" with a close button. The editor contains the following SQL command:

```
1 CREATE TABLE Person (  
2   person_id INTEGER PRIMARY KEY AUTOINCREMENT,  
3   first_name VARCHAR(64) NOT NULL,  
4   last_name VARCHAR(64) NOT NULL,  
5   birth_date DATE NOT NULL);  
6
```

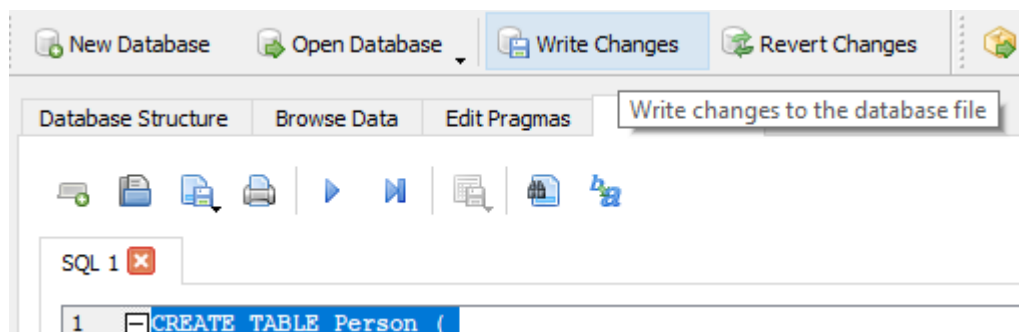
This command creates a table named “Person” with a person_id autoincrementing field, first and last name fields, and a birth date field.

Next click the “Execute all/selected SQL” button, which looks like . After you have done so, a message will appear stating that the table has been created. This is all shown below.

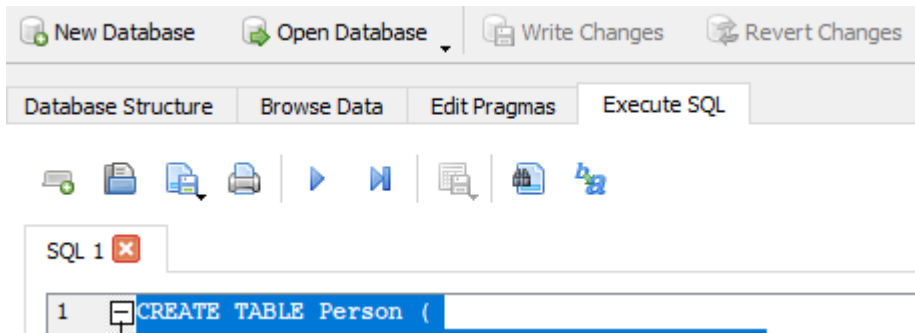


You can see that your person table was successfully created!

Save Changes In order to save the changes to the database file, you need to click the “Write Changes” button, as shown below.



Once you have done so, the button will gray out, indicating there is nothing more to save.



You can determine if any changes need to be saved by looking at these buttons to see if they are clickable, or grayed out.

Database Ready for Application

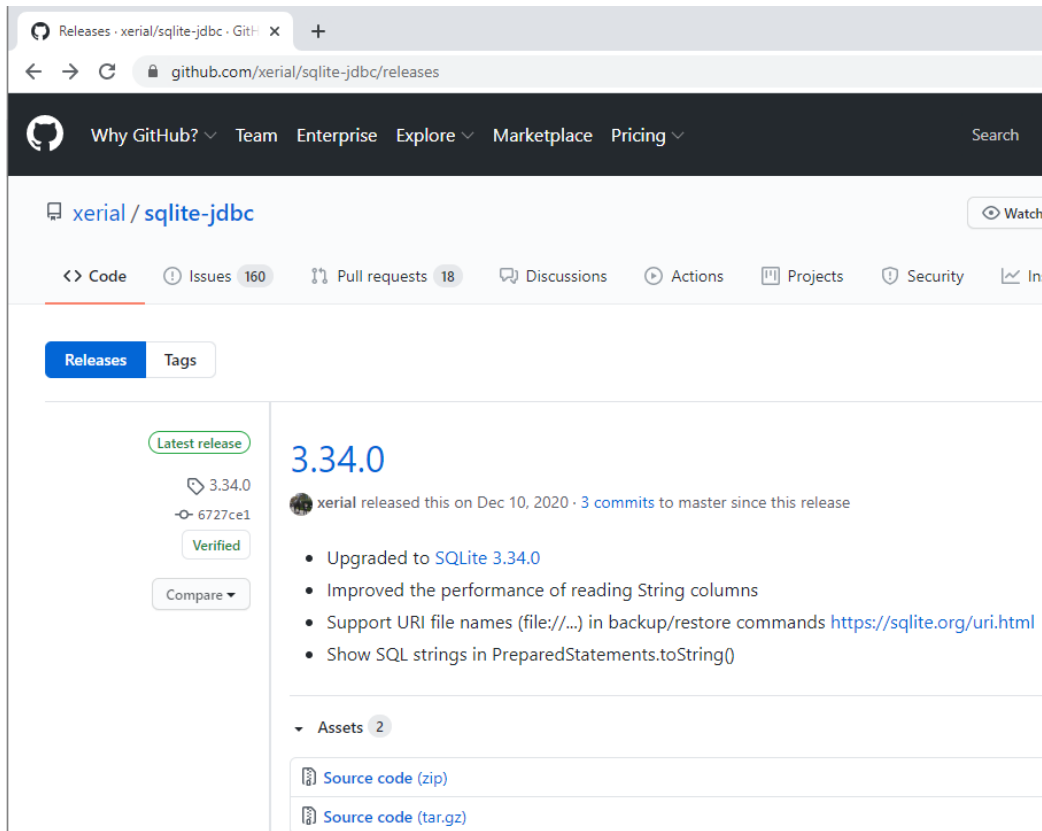
Congratulations! You've created a SQLite database and added a table to it. The database is now ready for an application to connect to it and work with person data, which we'll do in the next section.

Downloading the JDBC Driver

With your database setup, the next step is to download the SQLite JDBC driver so that it can be imported into your Eclipse, IntelliJ, or other IDE project. Java requires the JDBC driver to access the database.

Visit the Website

To get started, visit <https://github.com/xerial/sqlite-jdbc/releases>. You will see a list of drivers there, similar to the below screenshot.



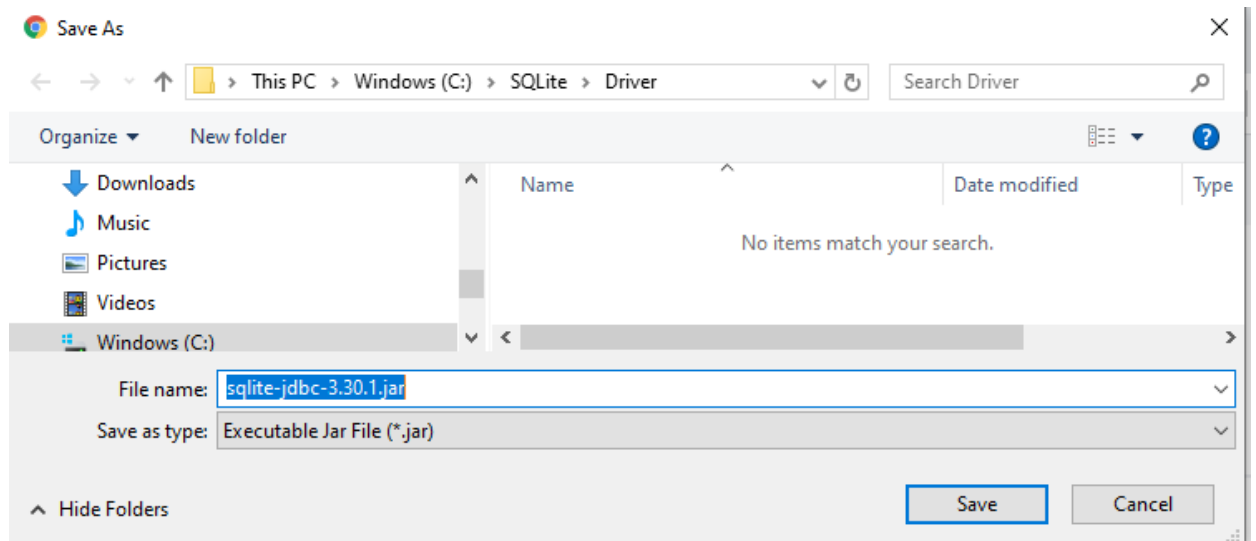
Download Driver

To start the download, click on the link *for the Jar file* for the most recent version available. You don't need the source files, just the jar file. An example Jar file link is shown below.



Once clicked, your browser will ask you where you'd like to save the file. *Make sure to save the driver in a directory you can remember*, because in a later section we will browse to the directory to use the driver in our IDE.

In the example below, it is saved into the C:\SQLite\Driver directory, one subdirectory below where the database itself is stored.



Once downloaded, you are set to import it into Eclipse, IntelliJ, or your other IDE, and may continue with the next section that applies to you.

Connecting to your Database in Eclipse

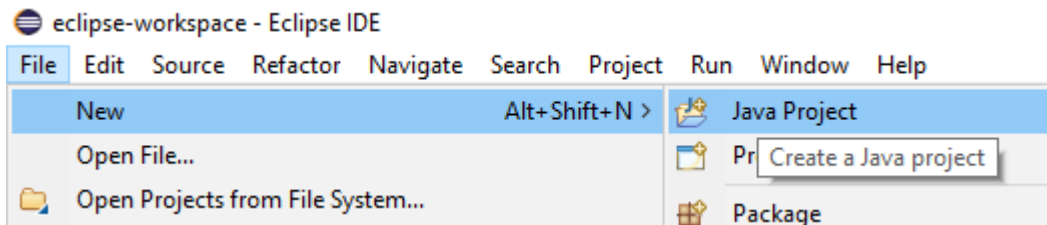
With the database setup and the JDBC driver downloaded, the next step is to use the database in your Java code. To get you started with SQLite, we will explore connecting to your database, adding data into the already created Person table, and querying the Person table.

This section illustrates how to do so in Eclipse. If you are using IntelliJ, you may skip this section and proceed with the next section titled “Connecting to your Database in IntelliJ”.

Step 1: Creating a New Project

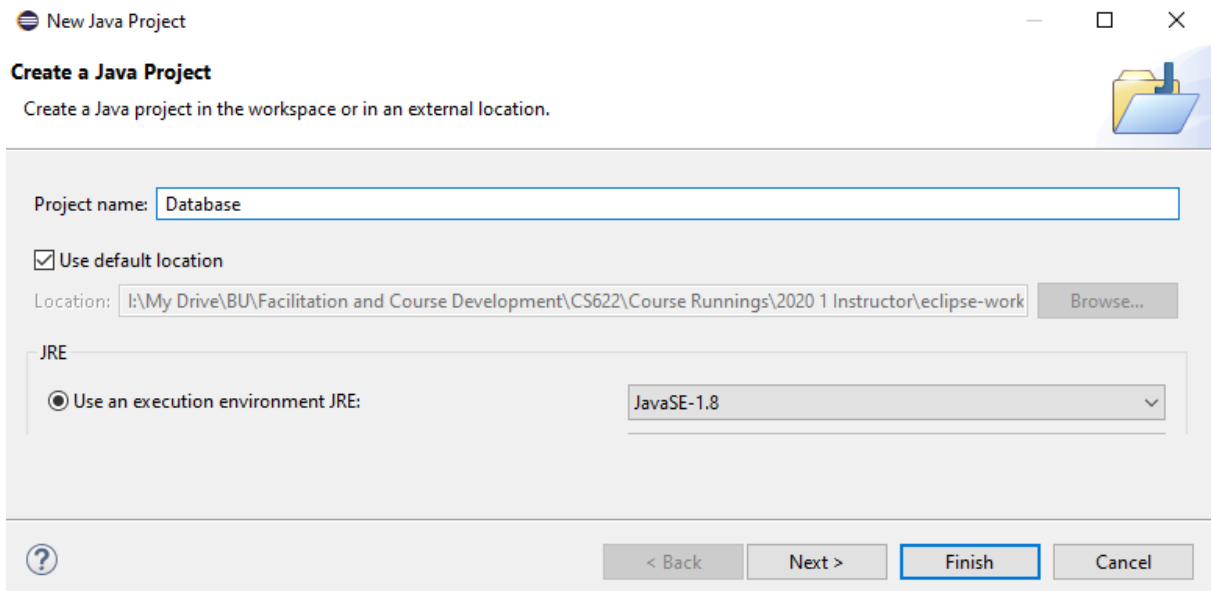
Identify Project

You can create a new Java project which will be used to test your database connection. Or, if you already have a project, you can use that and skip to Step 2. The first step is to initiate creation with File/New/Java Project, as shown below.



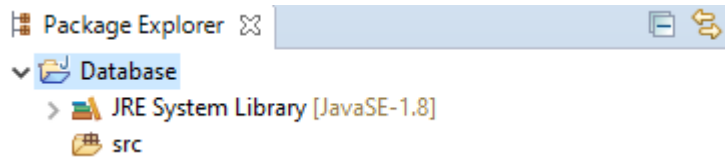
Name Project

We name our project “Database” since we will be testing out using our database.



Once the name is given, click the “Finish” button.

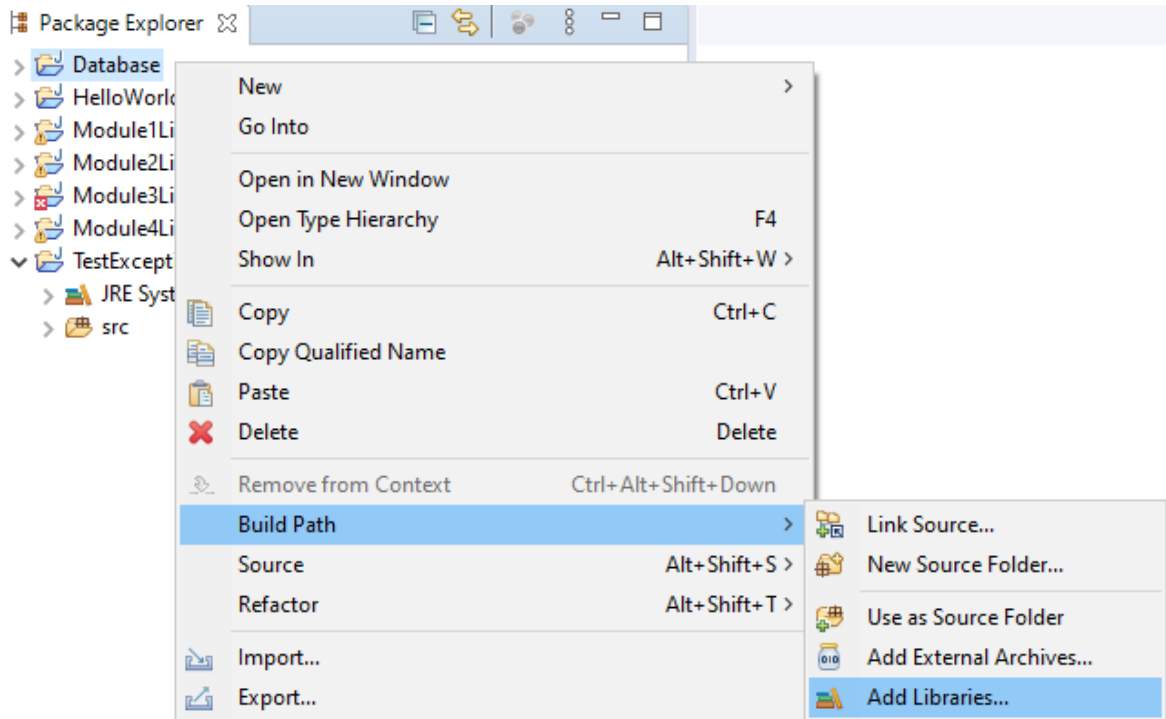
Now our project shows up under Package Explorer.



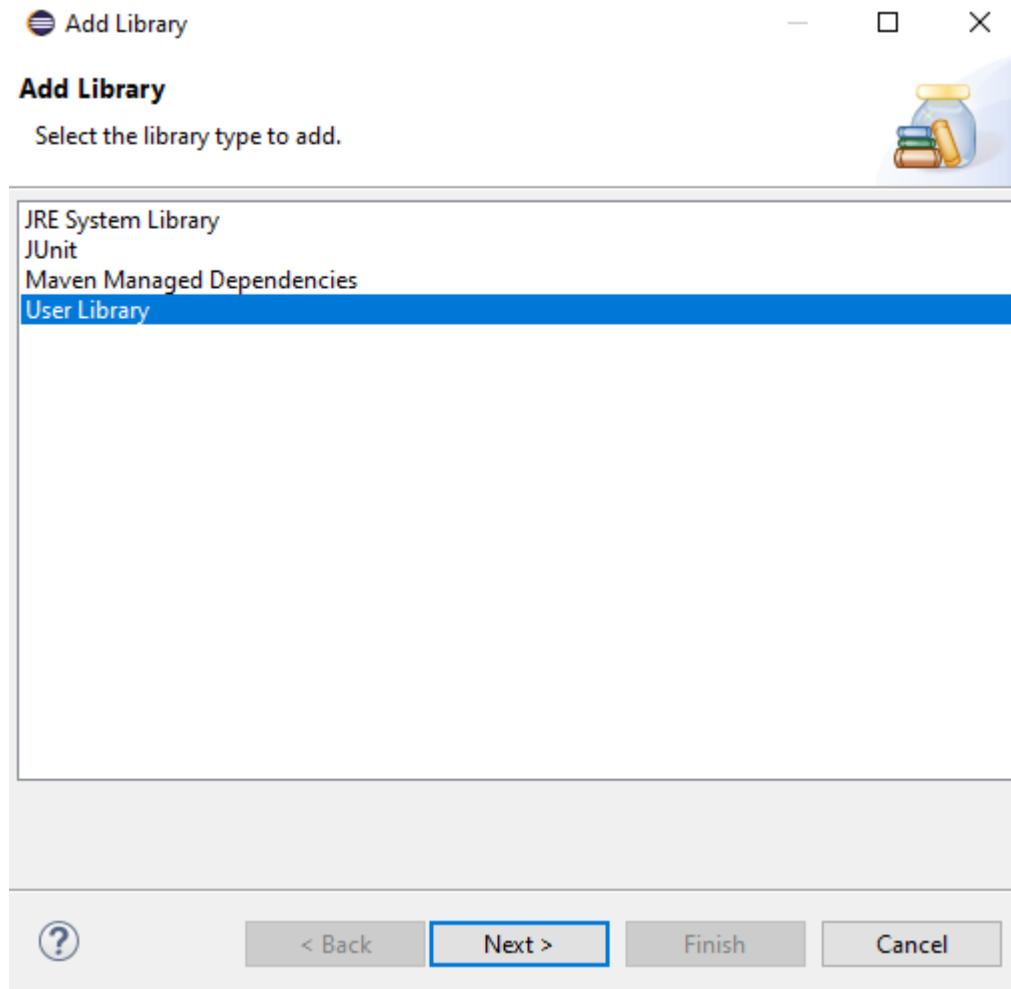
Step 2: Adding the JDBC Driver

Add JDBC Driver Jar

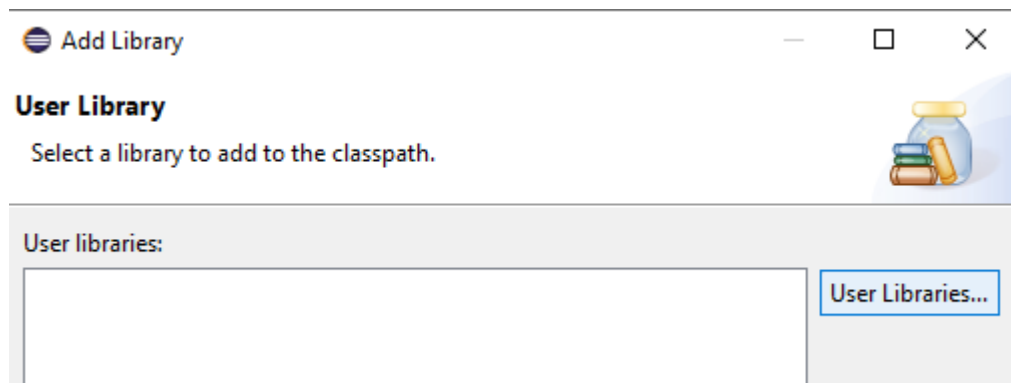
Right click on your project in Package Explorer, then click on Build Path/Add Libraries...



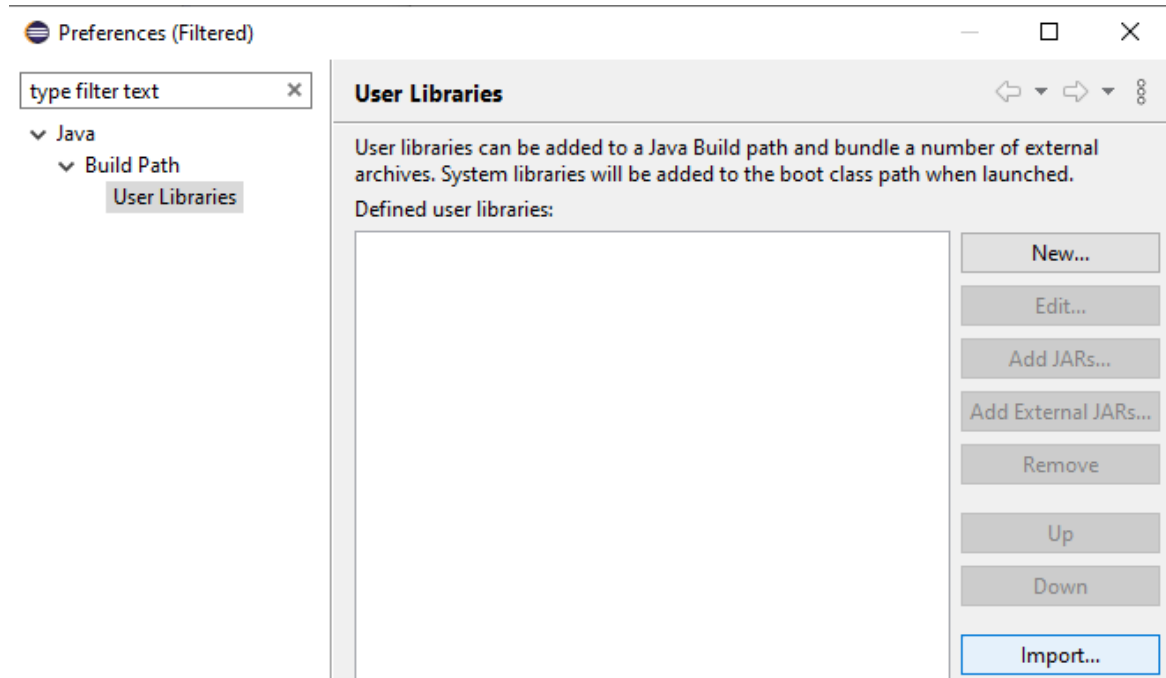
Select "User Library" and click Next.



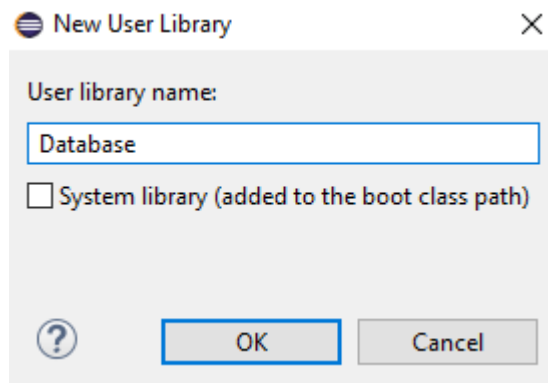
Click the "User Libraries..." button.



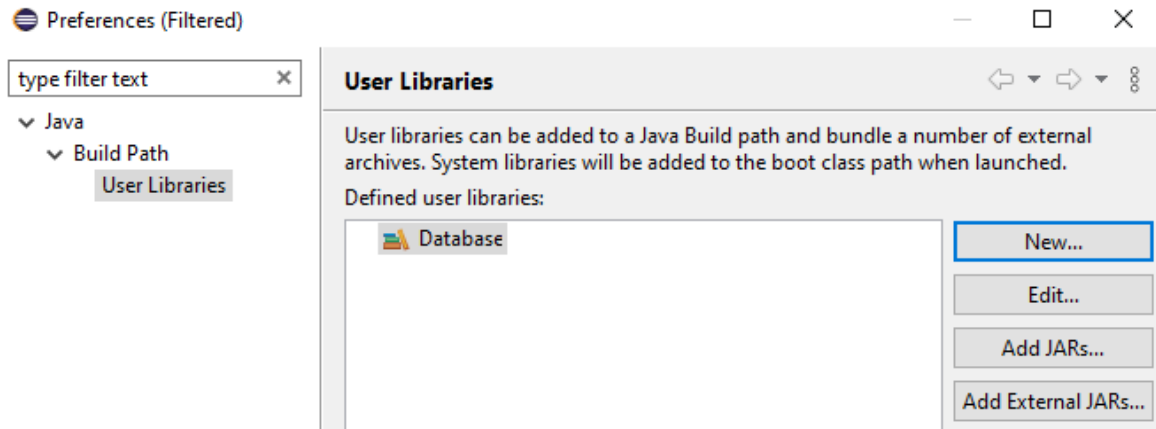
Click “New...”.



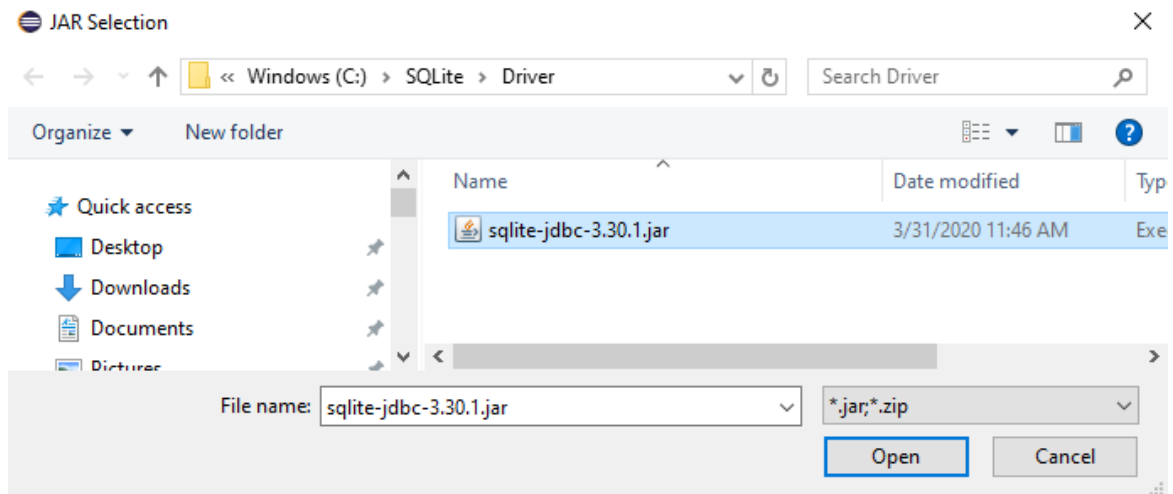
Give it a name. Here we use the name “Database”.



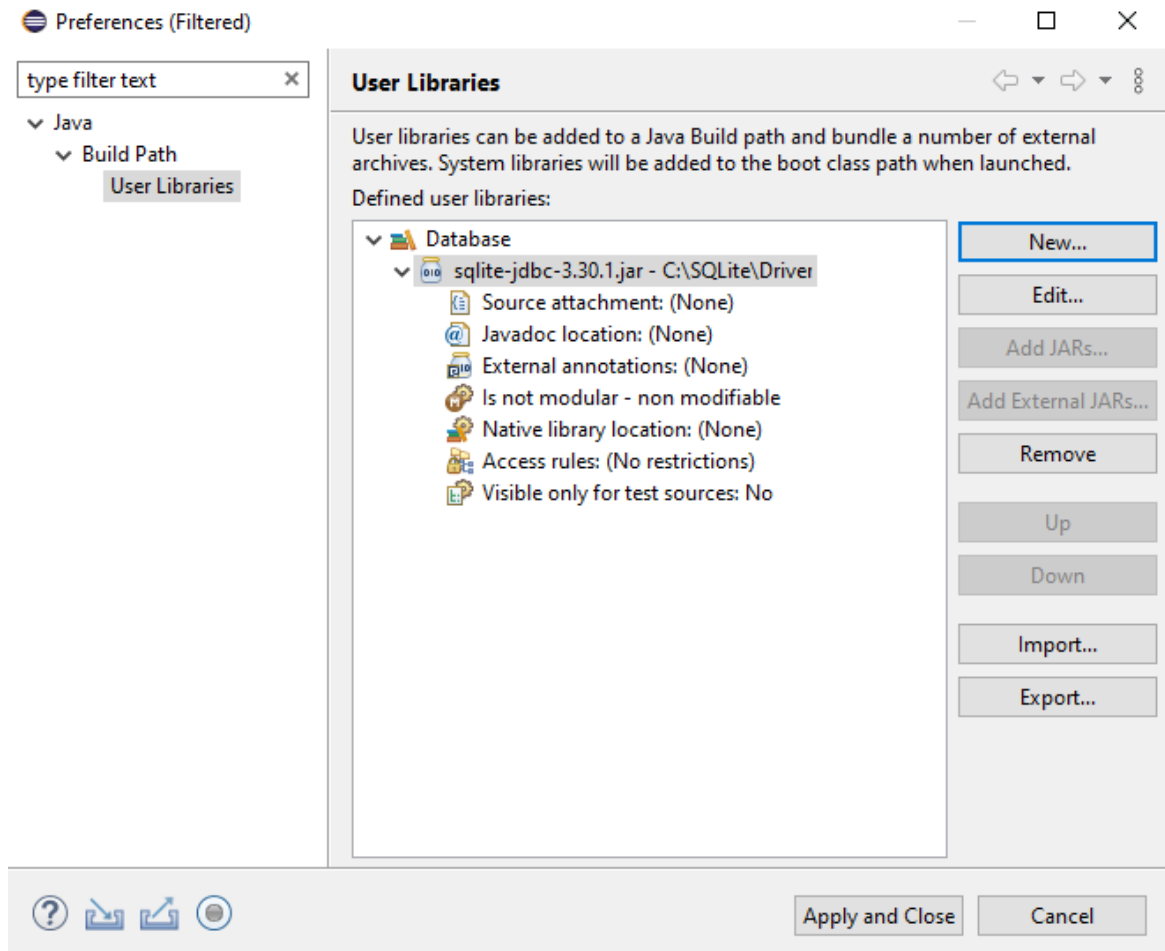
Click “Add External Jar...”.



Browse to the location of the Jar file you extracted and select the file. We had put it into C:\SQLite\Driver below.

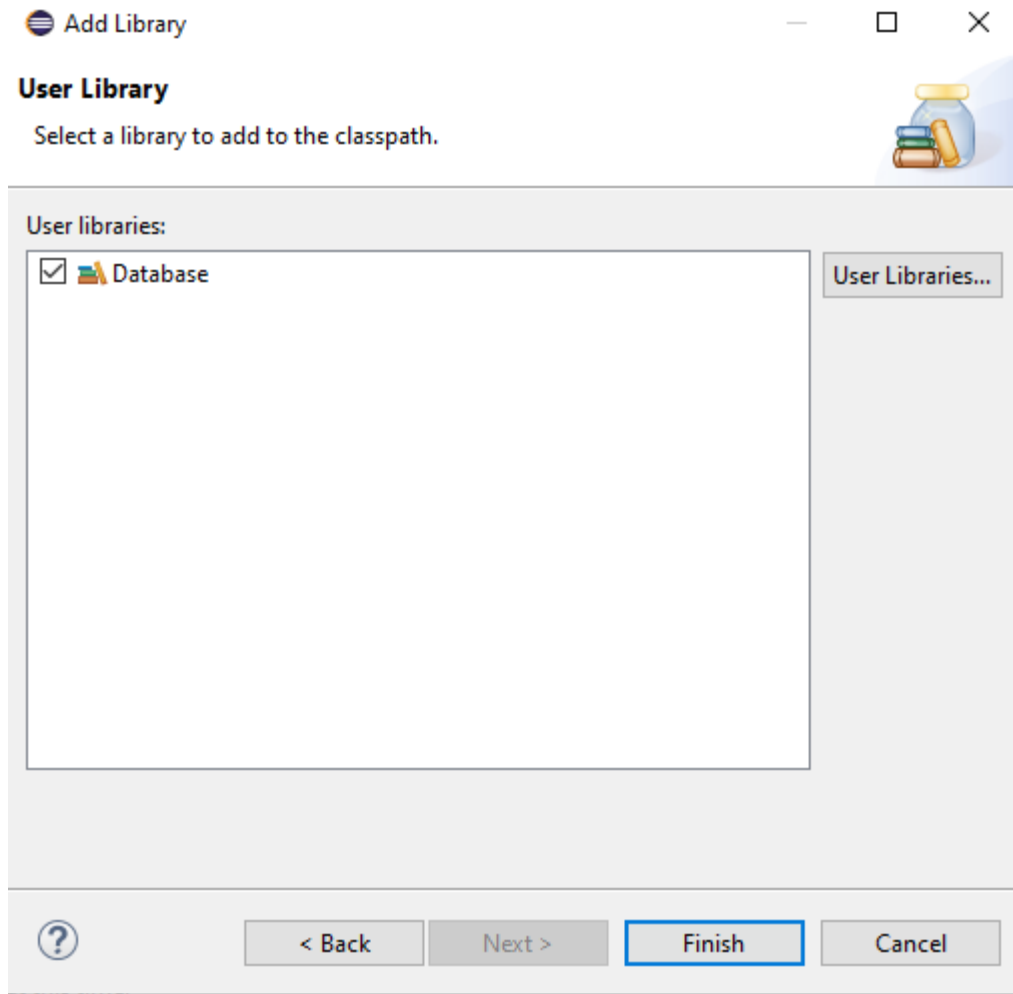


Once you’ve selected the Jar, you should see something like this, showing that the SQLite JDBC jar has been added to the library.

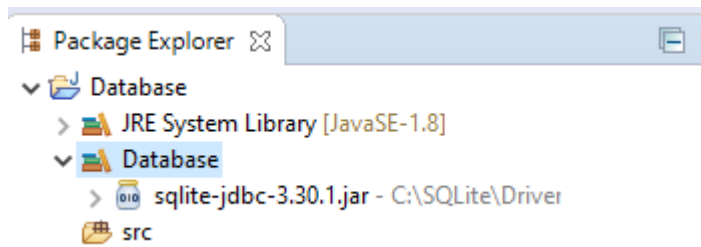


Click “Apply and Close” to apply what you have done.

On the next screen, make sure the “Database” user library is selected, and click the “Finish” button.



In Package Explorer, you will now see that the SQLite jar has been added to your project.



Note that in this process we added an “External” jar because the jar resided in a directory outside of our project directory. You could also copy the Jar into your project if you’d like and use it as an internal jar. It will work either way.

In real-world situations, some organizations have a common Jar directory checked into a repository that can be shared by many projects, just like we have done here with the C:\SQLite\Driver directory. Some organizations put the Jars into each project. Yet other organizations use a robust Jar management tool such as Ivy, which stores the Jars on a server, and are retrieved dynamically by the build process. The important takeaway here is that the SQLite jar must be included in your project, wherever it may be located.

Step 3: Inserting Rows

Create Class Now you create a Java class that connects to your database, inserts two rows, then queries those rows, outputting the results to the screen. The entire class is below.

```
1 package database;
2 import java.sql.*;
3
4 public class UseDatabase {
5     private static void insert(Connection conn) throws SQLException {
6         String sql = "INSERT INTO Person(first_name, last_name, birth_date) VALUES (?, ?, ?)";
7         try (PreparedStatement pstmt = conn.prepareStatement(sql)) {
8             pstmt.setString(1, "Bob");
9             pstmt.setString(2, "Smith");
10            pstmt.setDate(3, Date.valueOf("1976-1-13"));
11            pstmt.executeUpdate();
12
13            pstmt.setString(1, "Jane");
14            pstmt.setString(2, "Elizabeth");
15            pstmt.setDate(3, Date.valueOf("1979-3-15"));
16            pstmt.executeUpdate();
17        }
18    }
19
20    private static void query(Connection conn) throws SQLException {
21        String sql = "SELECT person_id, first_name, last_name, birth_date FROM Person";
22        try (Statement stmt = conn.createStatement();
23             ResultSet rs = stmt.executeQuery(sql)) {
24            while (rs.next()) {
25                System.out.printf("%d\t%-10s\t%-10s\t%tD%n",
26                                rs.getInt(1), rs.getString(2), rs.getString(3), rs.getDate(4));
27            }
28        }
29    }
30
31    public static void main(String[] args) throws SQLException {
32        String url = "jdbc:sqlite:C:/SQLite/GettingStarted.db";
33        try (Connection conn = DriverManager.getConnection(url)) {
34            insert(conn);
35            query(conn);
36        }
37    }
38 }
```

First, let's start with a high-level summary. The main method opens a connection, and passes it to an insert method and query method, respectively. The insert method inserts two rows into the Person table. The query method retrieves those rows and prints them out in a tabular format. Although there are many lines, we explain each line in turn. Note that we do not describe the lines sequentially; rather, we describe them in terms of program flow (which starts in the main method).

- Line 1** We put this class into a "database" package.
- Line 2** We import the java.sql package because we make use of many of its classes.
- Line 32** We start the main method by defining the connection string for our database. The "jdbc:sqlite" portion instructs the JDBC API to use the SQLite driver we included in our project, as opposed to some other driver such as Oracle or SQL Server, the "C:/SQLite/GettingStarted.db" portion instructs the SQLite JDBC driver as to what file to open. Since we had saved our database file to

C:\SQLite\GettingStarted.db, we specify that in the connection string. Although the string may look terse, the information in it is easily understandable.

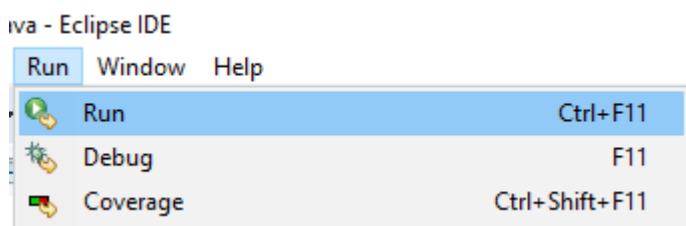
- Line 33** This opens the connection to our database, inside of a try/with block which will automatically close it. In order to work with the database in our application, a connection must be open to the database.
- Line 35** This invokes our insert method which will insert the rows into our database.
- Line 6** Inside the insert method, this is an example of SQL embedded into our application. This INSERT INTO command is used as the instruction to insert a row into the Person table. The “(first_name, last_name, birth_date)” portion describes which columns we are inserting into, and the order which we specify them. In this case, we are inserting into the first_name, last_name, and birth_date columns, respectively. Note that because person_id is an autoincrementing field, we don’t specify that here. The database will automatically assign it a value. The “VALUES (?, ?, ?)” portion indicates that we are inserting parameterized values, as opposed to hardcoding values. We use the “?” to indicate that we are not hardcoding any value, but can change the value at runtime.
- Line 7** We instantiate a PreparedStatement, which is needed to execute the parameterized SQL. By passing the SQL string as an argument, we have told the JDBC driver which SQL command we want it to execute. We put this instantiation inside of try/with block so that it will be closed automatically.
- Lines 8-10** Here we define what the parameters (defined by the “?” in the SQL string) are in turn. The first parameter is “Bob” corresponding to the first_name column, the second parameter is “Smith”, corresponding to the last_name column, and the third parameter is 1/13/1976, corresponding to the birth_date column.
- Line 11** This instructs the JDBC driver to execute the prepared statement with the given parameters. With the parameters set, it will be executing this command (but still using parameters behind the scenes):
- ```
INSERT INTO Person(first_name, last_name, birth_date)
VALUES ('Bob', 'Smith', '1/13/1976')
```
- Lines 13-16** Following similar logic to lines 8-11, these lines insert a new row with first\_name = “Jane”, last\_name = “Elizabeth”, and birth\_date = “3/15/1979”.
- Line 35** This invokes our query method which will retrieve and display the rows we have inserted.
- Line 21** Inside the query method, this SQL string is to select all four columns from the Person table, in this order – person\_id, first\_name, last\_name, birth\_date.
- Lines 22-23** A statement is created on line 22 with the conn.createStatement() method, and on line 23 the query (previously defined on line 21) is executed with the stmt.executeQuery() method. This returns an object of type ResultSet, which has all rows and columns from the results. These are created within a try/with block so that they are automatically closed.
- Line 24** This while loop uses the rs.next() method to iterate through each row. As long as there is another row in the result set, the next() method will return true.

**Lines 25-26** This prints out the results of the query in a tabular format. Note that the `ResultSet` class provides methods such as `getString()`, `getDate()`, and `getInt()`, to retrieve the specific fields in a row. The correct method must be used for the correct datatype. For example, `getInt()` must be used for `person_id` since it is an integer, and `getDate()` must be used for `birth_date` since it is a date. The first argument of these methods is the column number. For example, the `getInt()` call specifies "1" as the column number since `person_id` is queried first. The `getDate()` call specifies "4" as the column number is `birth_date` is queried last.

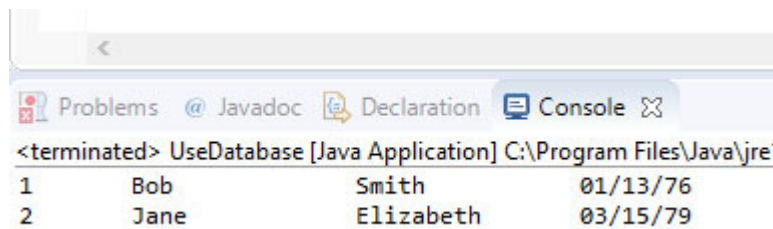
NOTE: The source code is available in Appendix A so that you may copy and paste as needed.

## Run Class

With the class defined, we can now run it with the Run/Run menu command.



After the class executes, you see the output as below.



Notice that the `person_id` autoincrement column starts at 1 and increments upwards by 1. Further notice that Bob Smith born on 1/13/1976 is listed first, followed by Jane Elizabeth born on 3/15/1979.

## Connecting to your Database in IntelliJ

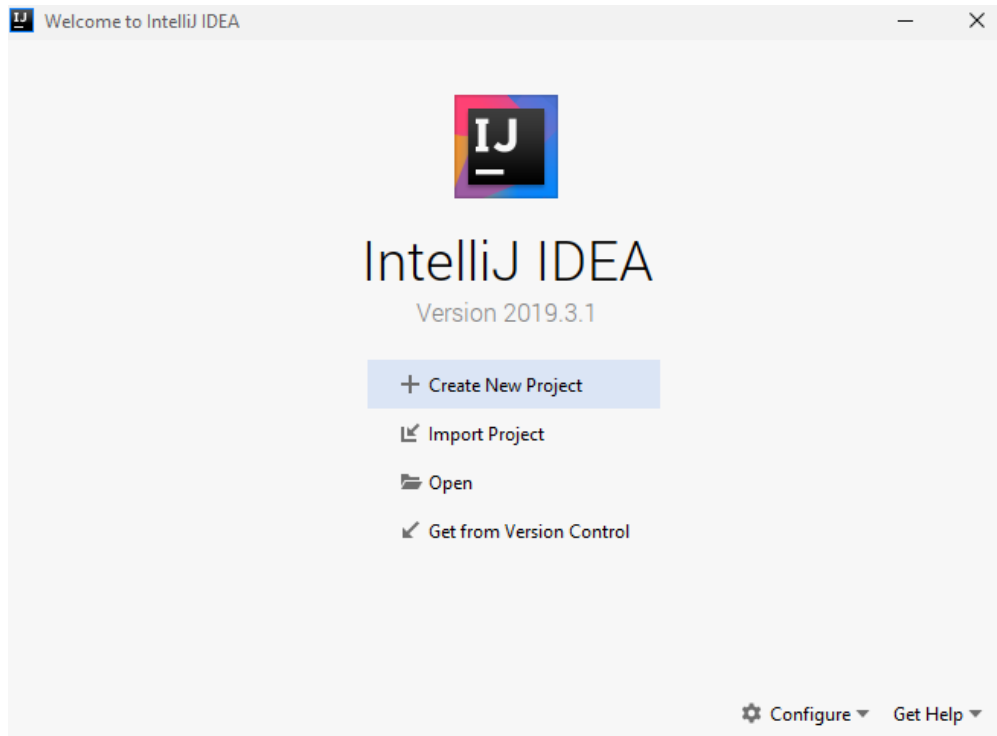
With the database setup and the JDBC driver downloaded, the next step is to use the database in your Java code. To get you started with SQLite, we will explore connecting to your database, adding data into the already created Person table, and querying the Person table.

This section illustrates how to do so in IntelliJ. If you are using Eclipse, you may skip this section and proceed with the prior section titled “Connecting to your Database in Eclipse”.

### Step 1: Creating a New Project

#### Create a Project

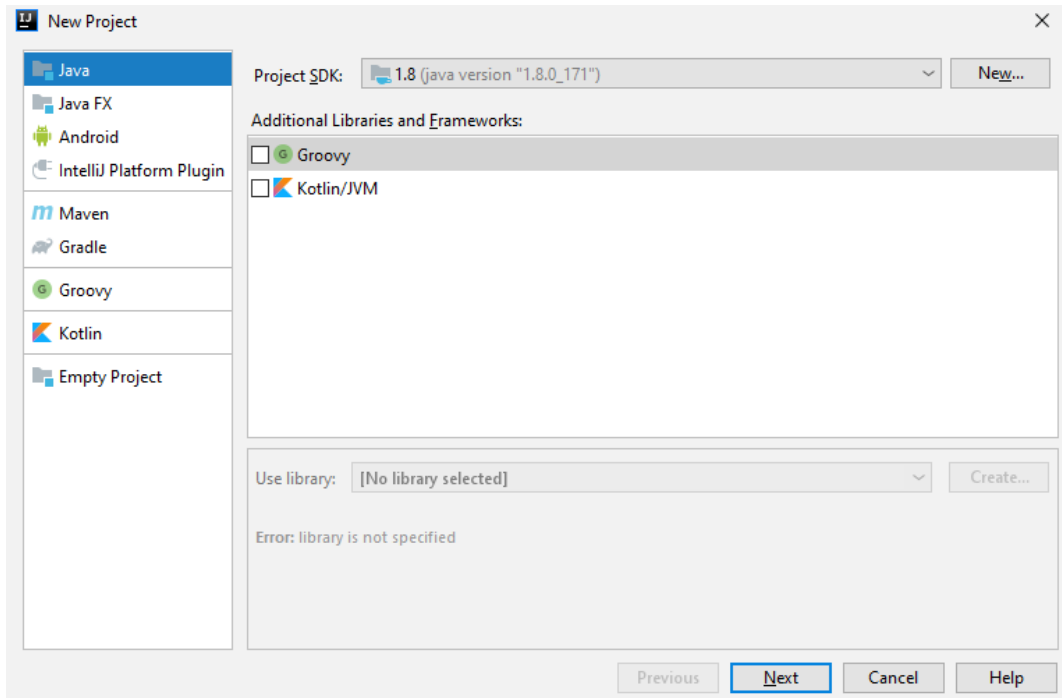
Click the “Create New Project” option to get started creating your project.





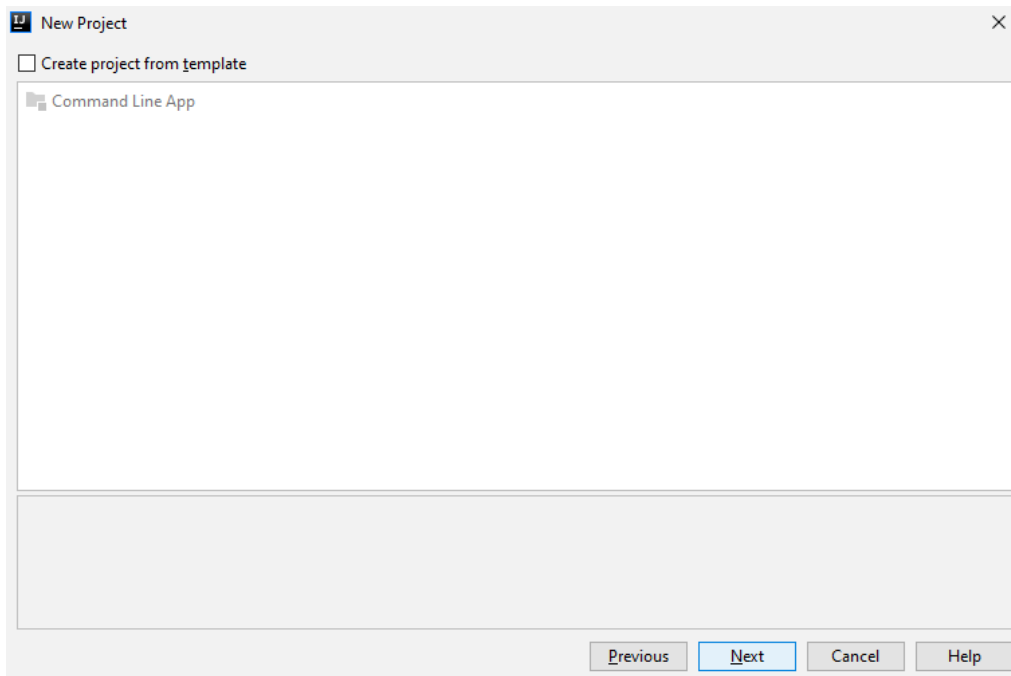
## Choose Project Type

Your next step is to choose the “Java” project type. There are other types of projects you can create, but a typical project is a Java project which allows you type Java code and execute it. The Java project type is selected by default, so just click the Next button to continue.



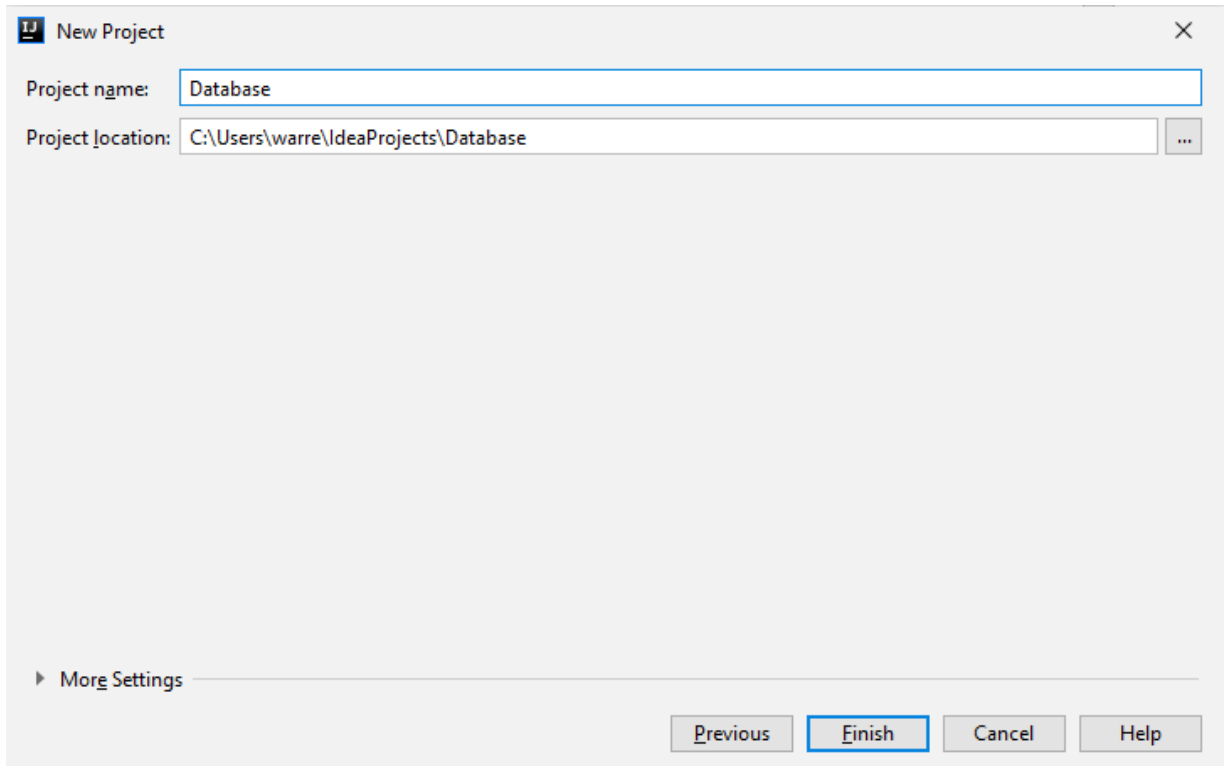
## Accept Template Defaults

Next, the screen prompts you to decide whether you’re creating your project from a template or not. We are creating a project from scratch (which is typical), so leave the “Create from project template” checkbox unchecked and click the Next button.



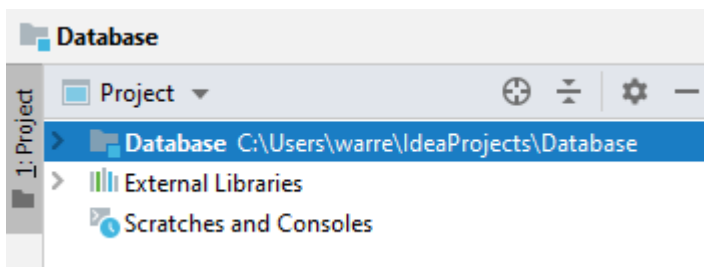
## Name Project

We name our project “Database” since we will be testing out using our database.



Once the name is given, click the “Finish” button.

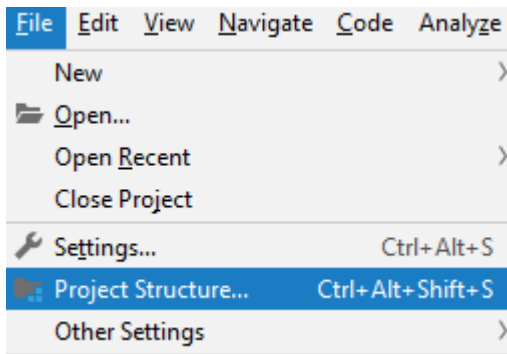
Now our project shows up in the Project window.



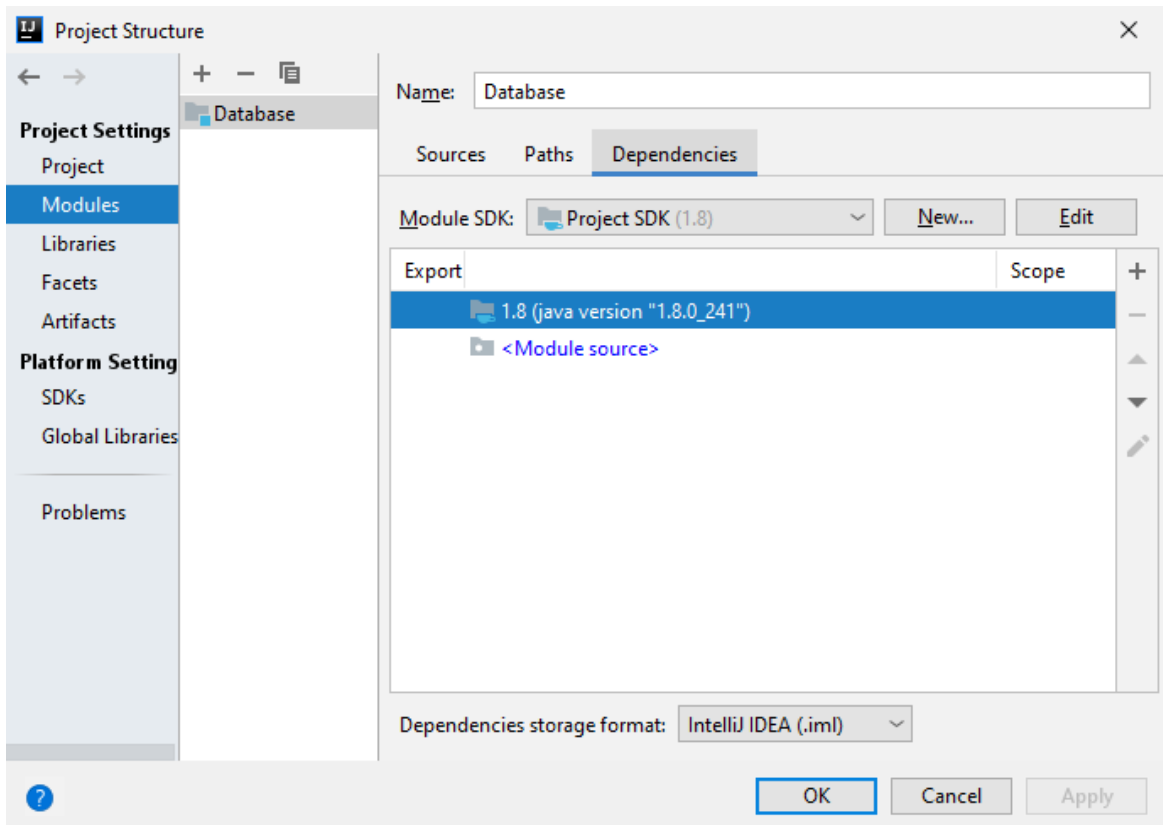
## Step 2: Adding the JDBC Driver

### Add JDBC Driver Jar

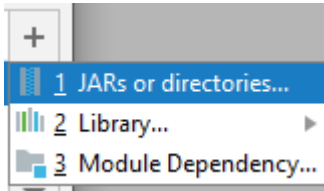
Start by accessing the File/Project Structure menu option.



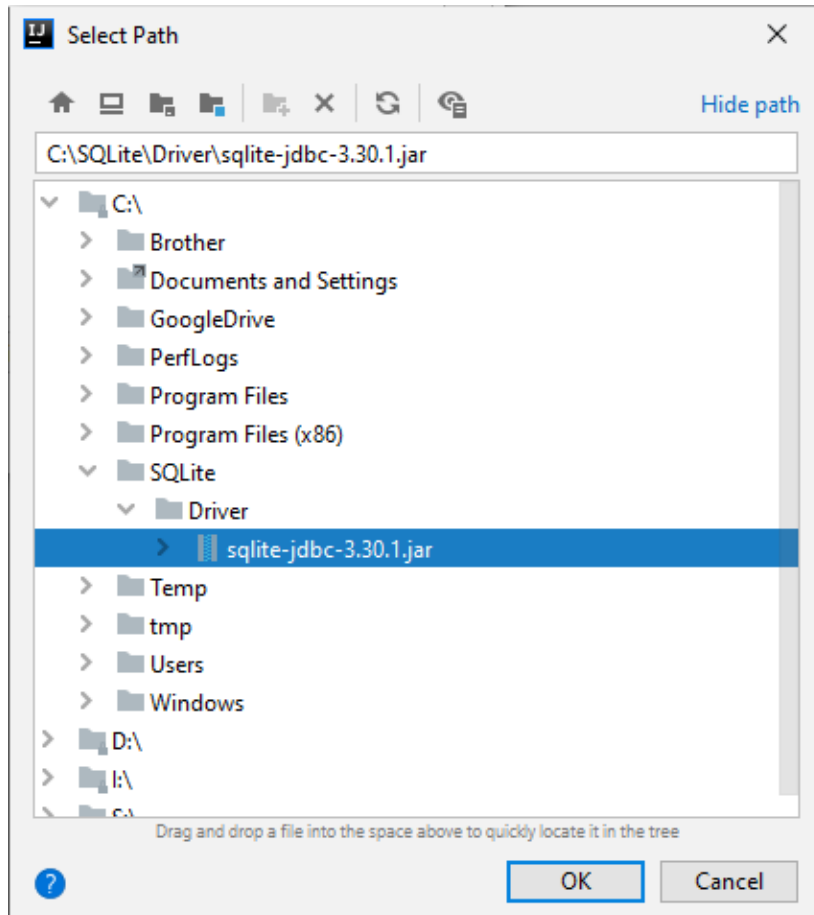
On the new window that appears, click on the Module option, then the Dependencies tab, as shown below.



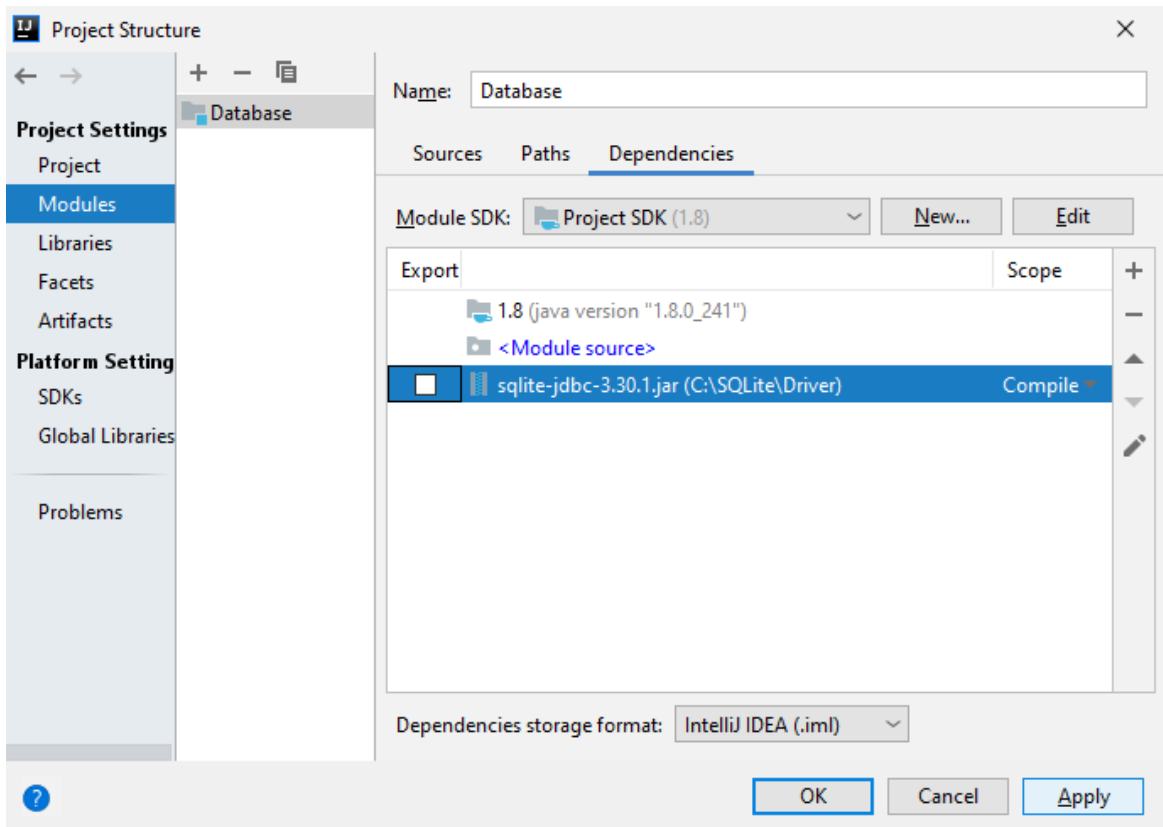
Then click on the plus sign, +, then click on “JARs or directories”, as shown below.



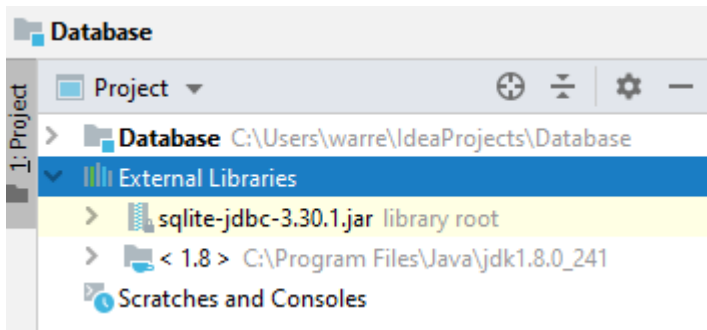
From there, select the driver that was downloaded in the C:\SQLite\Driver directory, and click the OK button.



You'll now see the jar file included. Click the OK button to close out of the Project Structure window.



Under the External Libraries on the Project window, you'll see the jar as well.



Note that in this process we added an “External” jar because the jar resided in a directory outside of our project directory. You could also copy the Jar into your project if you'd like and use it as an internal jar. It will work either way.

In real-world situations, some organizations have a common Jar directory checked into a repository that can be shared by many projects, just like we have done here with the C:\SQLite\Driver directory. Some organizations put the Jars into each project. Yet other organizations use a robust Jar management tool such as Ivy, which stores the Jars on a server, and are retrieved dynamically by the build process. The important takeaway here is that the SQLite jar must be included in your project, wherever it may be located.

## Step 3: Inserting Rows

**Create Class** Now you create a Java class that connects to your database, inserts two rows, then queries those rows, outputting the results to the screen. The entire class is below.

```
1 package database;
2 import java.sql.*;
3
4 public class UseDatabase {
5 @ private static void insert(Connection conn) throws SQLException {
6 String sql = "INSERT INTO Person(first_name, last_name, birth_date) VALUES (?, ?, ?)";
7 try (PreparedStatement pstmt = conn.prepareStatement(sql)) {
8 pstmt.setString(parameterIndex: 1, x: "Bob");
9 pstmt.setString(parameterIndex: 2, x: "Smith");
10 pstmt.setDate(parameterIndex: 3, Date.valueOf("1976-1-13"));
11 pstmt.executeUpdate();
12
13 pstmt.setString(parameterIndex: 1, x: "Jane");
14 pstmt.setString(parameterIndex: 2, x: "Elizabeth");
15 pstmt.setDate(parameterIndex: 3, Date.valueOf("1979-3-15"));
16 pstmt.executeUpdate();
17 }
18 }
19
20 private static void query(Connection conn) throws SQLException {
21 String sql = "SELECT person_id, first_name, last_name, birth_date FROM Person";
22 try (Statement stmt = conn.createStatement();
23 ResultSet rs = stmt.executeQuery(sql)) {
24 while (rs.next()) {
25 System.out.printf("%d\t%-10s\t%-10s\t%tD%n", rs.getInt(columnIndex: 1), rs.getString(columnIndex: 2),
26 rs.getString(columnIndex: 3), rs.getDate(columnIndex: 4));
27 }
28 }
29 }
30
31 public static void main(String[] args) throws SQLException {
32 String url = "jdbc:sqlite:C:/SQLite/GettingStarted.db";
33 try (Connection conn = DriverManager.getConnection(url)) {
34 insert(conn);
35 query(conn);
36 }
37 }
38 }
```

First, let's start with a high-level summary. The main method opens a connection, and passes it to an insert method and query method, respectively. The insert method inserts two rows into the Person table. The query method retrieves those rows and prints them out in a tabular format. Although there are many lines, we explain each line in turn. Note that we do not describe the lines sequentially; rather, we describe them in terms of program flow (which starts in the main method).

**Line 1** We put this class into a "database" package.

**Line 2** We import the java.sql package because we make use of many of its classes.

**Line 32** We start the main method by defining the connection string for our database. The "jdbc:sqlite" portion instructs the JDBC API to use the SQLite driver we included in our project, as opposed to some other driver such as Oracle or SQL Server, the "C:/SQLite/GettingStarted.db" portion instructs the SQLite JDBC

driver as to what file to open. Since we had saved our database file to C:\SQLite\GettingStarted.db, we specify that in the connection string. Although the string may look terse, the information in it is easily understandable.

- Line 33** This opens the connection to our database, inside of a try/with block which will automatically close it. In order to work with the database in our application, a connection must be open to the database.
- Line 35** This invokes our insert method which will insert the rows into our database.
- Line 6** Inside the insert method, this is an example of SQL embedded into our application. This INSERT INTO command is used as the instruction to insert a row into the Person table. The “(first\_name, last\_name, birth\_date)” portion describes which columns we are inserting into, and the order which we specify them. In this case, we are inserting into the first\_name, last\_name, and birth\_date columns, respectively. Note that because person\_id is an autoincrementing field, we don’t specify that here. The database will automatically assign it a value. The “VALUES (?, ?, ?)” portion indicates that we are inserting parameterized values, as opposed to hardcoding values. We use the “?” to indicate that we are not hardcoding any value, but can change the value at runtime.
- Line 7** We instantiate a PreparedStatement, which is needed to execute the parameterized SQL. By passing the SQL string as an argument, we have told the JDBC driver which SQL command we want it to execute. We put this instantiation inside of try/with block so that it will be closed automatically.
- Lines 8-10** Here we define what the parameters (defined by the “?” in the SQL string) are in turn. The first parameter is “Bob” corresponding to the first\_name column, the second parameter is “Smith”, corresponding to the last\_name column, and the third parameter is 1/13/1976, corresponding to the birth\_date column.
- Line 11** This instructs the JDBC driver to execute the prepared statement with the given parameters. With the parameters set, it will be executing this command (but still using parameters behind the scenes):
- ```
INSERT INTO Person(first_name, last_name, birth_date)
VALUES ('Bob', 'Smith', '1/13/1976')
```
- Lines 13-16** Following similar logic to lines 8-11, these lines insert a new row with first_name = “Jane”, last_name = “Elizabeth”, and birth_date = “3/15/1979”.
- Line 35** This invokes our query method which will retrieve and display the rows we have inserted.
- Line 21** Inside the query method, this SQL string is to select all four columns from the Person table, in this order – person_id, first_name, last_name, birth_date.
- Lines 22-23** A statement is created on line 22 with the conn.createStatement() method, and on line 23 the query (previously defined on line 21) is executed with the stmt.executeQuery() method. This returns an object of type ResultSet, which has all rows and columns from the results. These are created within a try/with block so that they are automatically closed.

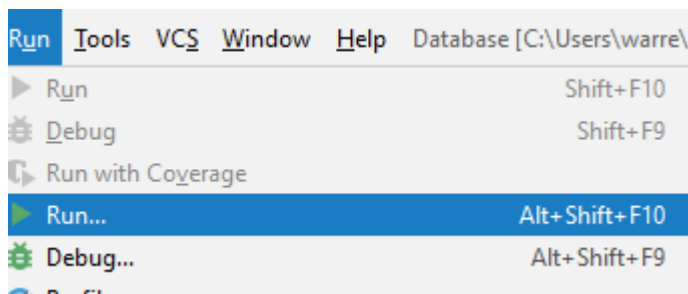
Line 24 This while loop uses the `rs.next()` method to iterate through each row. As long as there is another row in the result set, the `next()` method will return true.

Lines 25-26 This prints out the results of the query in a tabular format. Note that the `ResultSet` class provides methods such as `getString()`, `getDate()`, and `getInt()`, to retrieve the specific fields in a row. The correct method must be used for the correct datatype. For example, `getInt()` must be used for `person_id` since it is an integer, and `getDate()` must be used for `birth_date` since it is a date. The first argument of these methods is the column number. For example, the `getInt()` call specifies "1" as the column number since `person_id` is queried first. The `getDate()` call specifies "4" as the column number since `birth_date` is queried last.

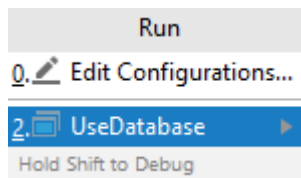
NOTE: The source code is available in Appendix A so that you may copy and paste as needed.

Run Class

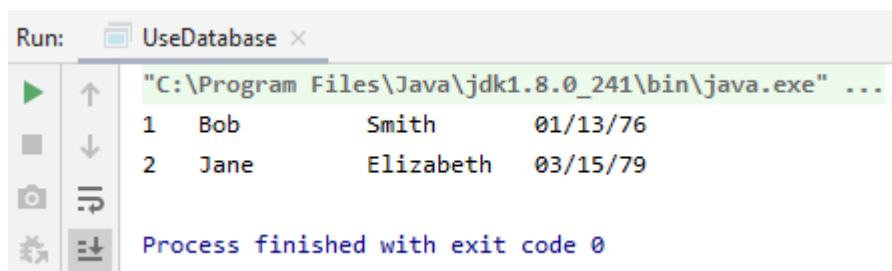
With the class defined, we can now run it with the Run/Run menu command.



Select the `UseDatabase` class to execute.



After the class executes, you see the output as below.



Notice that the `person_id` autoincrement column starts at 1 and increments upwards by 1. Further notice that Bob Smith born on 1/13/1976 is listed first, followed by Jane Elizabeth born on 3/15/1979.

Next Steps

Congratulations! You are now well on your way to working with SQLite. You have used a SQL client, DB Browser for SQLite, to create a database as well as a table. You have used the SQLite JDBC driver to add data to the table and retrieved the data in Java. Your instructor may ask you to use other SQL commands and JDBC features, and you now have a framework from which to do so. Don't worry. If you can create one database, you can create many. If you can execute one SQL command in Java, you can execute many. You are well on your way.

Appendix A: Source Code

The source code for the UseDatabase class is available below so that you can copy and paste it as needed.

```
package database;
import java.sql.*;

public class UseDatabase {
    private static void insert(Connection conn) throws SQLException {
        String sql = "INSERT INTO Person(first_name, last_name, birth_date) VALUES (?, ?, ?)";
        try (PreparedStatement pstmt = conn.prepareStatement(sql)) {
            pstmt.setString(1, "Bob");
            pstmt.setString(2, "Smith");
            pstmt.setDate(3, Date.valueOf("1976-1-13"));
            pstmt.executeUpdate();

            pstmt.setString(1, "Jane");
            pstmt.setString(2, "Elizabeth");
            pstmt.setDate(3, Date.valueOf("1979-3-15"));
            pstmt.executeUpdate();
        }
    }

    private static void query(Connection conn) throws SQLException {
        String sql = "SELECT person_id, first_name, last_name, birth_date FROM Person";
        try (Statement stmt = conn.createStatement();
            ResultSet rs = stmt.executeQuery(sql)) {
            while (rs.next()) {
                System.out.printf("%d\t%-10s\t%-10s\t%tD%n",
                    rs.getInt(1), rs.getString(2), rs.getString(3), rs.getDate(4));
            }
        }
    }

    public static void main(String[] args) throws SQLException {
        String url = "jdbc:sqlite:C:/SQLite/GettingStarted.db";
        try (Connection conn = DriverManager.getConnection(url)) {
            insert(conn);
            query(conn);
        }
    }
}
```

Works Cited

Solid IT (March 2020). DB-Engines Ranking. Retrieved March 23, 2020, from <https://db-engines.com/en/ranking>

SQLite (March 2020). About SQLite. Retrieved March 27, 2020, from <https://www.sqlite.org/about.html>.