Documentation

For

Database

In

3D Printer Queue Application

By

Andrew Doser

March 6, 2019

Table of Contents

[Introduction 1](#_Toc2952243)

[Brief Overview of Databases 2](#_Toc2952244)

[SQLite 2](#_Toc2952245)

[SQL 2](#_Toc2952246)

[How to Use the Database in the 3D Printer Queue Application 3](#_Toc2952247)

[SQLite 3](#_Toc2952248)

[List of the Models 3](#_Toc2952249)

[Create: A Table 5](#_Toc2952250)

[How to Create an SQL Database 6](#_Toc2952251)

[Suitable Platforms for SQL 7](#_Toc2952252)

[How to Create an SQLite Database 8](#_Toc2952253)

# Introduction

The purpose of this documentation is to assist the 3D print club in maintaining and updating the database in the 3D Printer Queue Application. In order to fulfill this purpose, this documentation will be focusing three different topics.

1. To provide an overview of databases.
2. To provide on how to use the database in the 3D Printer Queue Application
3. To provide on how to edit and maintain the local and cloud databases.

This document will focus preparing the reader for code implementations of a database with pictures. In preparation for that, the following subjects will be covered:

* Creating an SQL/SQLite database
* CRUD operations on database in C#
* Looking at the contents in database

This information is important when wanting to change the database in anyway such as a name of an attribute inside the database. If changing the database without prior knowledge, the user can easily make a catastrophic mistake and create an unusable application.

The end of the document will explain best practices when it comes to maintaining a local and cloud database. It will examine to change names inside the database and to add more tables for storing data. There will be additional resources linked at the end of this document to provide help with creating a database in either SQL/SQLite.

# Brief Overview of Databases

In this chapter we will be going over the two types of databases that were used in the 3D Printer Application (3D Printer App). There will be a brief overview of what the databases are, why they were chosen, and which platform they were used for.

## SQLite

There are three reasons for using SQLite for the mobile platform of the 3D Printer Application.

* SQLite requires less processing
* SQLite can store local data
* SQLite stores data like SQL

When compared with other options available and how fast the team wanted the application to run, it made sense to keep track of the data using SQLite. SQLite makes it easy to store data and then import it to a cloud database when needed. This will allow the user to make pulls from the cloud database without wasting a lot of the battery life on the user’s phone.

## SQL

There are three reasons for using SQL for the Web platform of the 3D Printer Application.

* SQL is a common database language found in many enterprise environments
* SQL integrated well with .Net Framework and Razor pages
* SQL has a lot of resources for managing data effectively

After looking at other Web platform solutions for a database, the team chose SQL. The focus for the Web platform was to look at data on the administrator side of the 3D Printer Application. SQL not only provides that but it is integrated with the API the team chose for the Web platform. Making it the best solution for the 3D Printer Application.

# How to Use the Database in the 3D Printer Queue Application

This chapter will take go through accessing the database in the code behind files of both the Web and Mobile platforms of the 3D Printer Queue Application. It will cover the steps such as inserting, deleting, reading, and creating in both the web and mobile platforms. If a reader would like to follow along with this documentation please have the following installed.

* Microsoft Visual Studio 2015 (or Higher)
* Microsoft SQL Server Management Studio 2015 (or Higher)
* Microsoft SQL Server 2017 Developer Edition (or Higher)
* Nuget Package called, “sqlite-net-pcl” by Frank A. Krueger

## SQLite

This section will be looking into the models of the database and how to create, update, delete, and read the database on the mobile platform

### List of the Models

In this section there will be a list of the models and the data they were meant to store. A model is a C# class file that is used as a data template when creating a table in SQLite. Generally, a model only requires the PrimaryKey to be defined in order to create a table. However, by adding these attributes shown. A programmer can limit the data that can be taken in and make sure that there isn’t any duplicate or extraneous data.

**public** **class** User

{

[PrimaryKey, AutoIncrement]

**public** **int** ID { **get**; **set**; }

[MaxLength(50), Unique]

**public** **string** Email { **get**; **set**; }

[MaxLength(50)]

**public** **string** Name { **get**; **set**; }

**public** **int** Admin { **get**; **set**; }

[MaxLength(50)]

**public** **string** Password { **get**; **set**; }

[OneToMany(CascadeOperations = CascadeOperation.All)]

**public** List<Request> Requests { **get**; **set**; }

}

**public** **class** Printer

{

[PrimaryKey,AutoIncrement]

**public** **int** ID { **get**; **set**; }

[MaxLength(50), Unique]

**public** **string** Name { **get**; **set**; }

[ForeignKey(**typeof**(Status))]

**public** **int** StatusID { **get**; **set**; }

[ForeignKey(**typeof**(PrintColor))]

**public** **int** ColorID { **get**; **set**; }

**public** **int** ProjectsQueued { **get**; **set**; }

[OneToMany(CascadeOperations = CascadeOperation.All)]

**public** List<Request> Requests { **get**; **set**; }

[ManyToOne]

**public** Status status { **get**; **set**; }

[ManyToOne]

**public** PrintColor color { **get**; **set**; }

}

**public** **class** PrintColor

{

[PrimaryKey, AutoIncrement]

**public** **int** ID { **get**; **set**; }

[Unique, NotNull]

**public** **string** Name { **get**; **set**; }

[Unique, NotNull]

**public** **string** HexValue { **get**; **set**; }

[OneToMany(CascadeOperations = CascadeOperation.All)]

**public** List<Printer> Printers { **get**; **set**; }

}

**public** **class** Request

{

[PrimaryKey, AutoIncrement]

**public** **int** ID { **get**; **set**; }

[ForeignKey(**typeof**(Printer))]

**public** **int** PrinterID { **get**; **set**; }

[ForeignKey(**typeof**(Status))]

**public** **int** StatusID { **get**; **set**; }

[ForeignKey(**typeof**(User))]

**public** **int** UserID { **get**; **set**; }

**public** DateTime DateMade { **get**; **set**; }

**public** DateTime DateRequested { **get**; **set**; }

**public** **int** Duration { **get**; **set**; }

**public** **string** ProjectName { **get**; **set**; }

**public** **string** Description { **get**; **set**; }

**public** **string** File { **get**; **set**; }

**public** **string** Personal { **get**; **set**; }

[ManyToOne]

**public** User user { **get**; **set**; }

[ManyToOne]

**public** Status status { **get**; **set**; }

[ManyToOne]

**public** Printer printer { **get**; **set**; }

}

**public** **class** Status

{

[PrimaryKey, AutoIncrement]

**public** **int** ID { **get**; **set**; }

[Unique,NotNull,MaxLength(10)]

**public** **string** Name { **get**; **set**; }

[OneToMany(CascadeOperations = CascadeOperation.All)]

**public** List<Printer> Printers { **get**; **set**; }

[OneToMany(CascadeOperations = CascadeOperation.All)]

**public** List<Request> Requests { **get**; **set**; }

}

Shown above are the current models used to create the SQLite Database in the Mobile platform. After a model is properly created, an SQLite Table can be created.

### Create: A Table and Row

To create a table you need to first locate SQLite database location. What is shown below is a coding statement to create a connection to the SQLite database.

**using** (SQLiteConnection conn = **new** SQLiteConnection(App.DatabaseLocation))

In this example, App.DatabaseLocation is a public static string inside the App class that has the file location for reading and writing data to. SQLiteConnectionis variable that creates the connection to the database file in the Application. If no such file exists, SQLiteConnection will create a database file. Since the variable was declared in a using statement, the programmer must make calls to the database inside the using statement. As shown below.

**using** (SQLiteConnection conn = **new** SQLiteConnection(App.DatabaseLocation))

{

conn.CreateTable<User>();

}

When CreateTable is called, SQLiteConnection first checks if a table was made with the model listed in the angled brackets. If such a table does not exist, then SQLiteConnection will create a table. If not, then SQLite Connection will do nothing. Once a table is created, a programmer can then insert data into the table. An example is shown below.

**int** Num = 0;

**using** (SQLiteConnection conn = **new** SQLiteConnection(App.DatabaseLocation))

{

conn.CreateTable<User>();

Num = conn.Insert(user);

}

In “conn.Insert(user),” user is an object that was created earlier as a User class that has the appropriate data into it. Also, “conn.Insert()” does return an integer based on how many rows were affected. In this case, since there was only one insert, Num will be one in this example. Furthermore, when a programmer creates a connection to the database, they should keep the operations they do inside the using statement to a minimum. Opening a connection to a database, even a local one, is an expensive operation. Doing multiple actions while having an open connection can cause problems with the data inside the SQLite database.

### Read: A Table

To read a table requires almost the same series of steps from the section “Create: A Table and Row.” However, instead for creating and inserting, the currently created table will be read from.

**var** users = **new** List<User>();

**using** (SQLiteConnection conn = **new** SQLiteConnection(App.DatabaseLocation))

{

users = conn.Table<User>().ToList();

}

As shown above, first a connection to the database is created, then a call to the User table is made. When “conn.Table<User>()” is called, is returns every row in the table. However, in order to be able to use the data effectively, it must be stored into a list. Also, please note that variable users was declared outside the using statement. When a variable is declared inside a using statement, the variable cannot be accessed outside of it.

After the list is returned, a programmer can find the user their looking for by doing a LINQ query as shown below.

**var** user = users.SingleOrDefault(g => g.Email == userNameEntry.Text);

LINQ statements are helpful when sorting data in a List. For more information on LINQ queries please go to:

<https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/concepts/linq/getting-started-with-linq>

### Update: A Row

To update a row inside a table requires a connection to the database and the item the programmer wants to update. The amount of preparation involved is the same amount as the section for “Read: A Table”. An example is shown below.

**var** users = **new** List<User>();

**using** (SQLiteConnection conn = **new** SQLiteConnection(App.DatabaseLocation))

{

users = conn.Table<User>().ToList();

}

**var** user = users.SingleOrDefault(g => g.Email == “BigBird@gmail.com”);

As shown above, the programmer pulls the current list of users and then searches that list for a specific user. They then change the user and update it to the database as shown below.

user.Name = "Matt Vogel";

**using** (SQLiteConnection conn = **new** SQLiteConnection(App.DatabaseLocation))

{

conn.Update(user);

}

Now, as shown, the user.Name was changed and was then updated to the database. Since user is of the User class, SQLite knows which table to access and then finds an item matching the PrimaryKey in the table to update the item. Everything in object is updated except for its PrimaryKey.

### Delete: A Table and Row

There are two ways to delete. A programmer can delete a single row or multiple rows. In the first example it will show how to delete a single row.

**using** (SQLiteConnection conn = **new** SQLiteConnection(App.DatabaseLocation))

{

conn.CreateTable<User>();

**var** user = conn.Table<User>().ToList().SingleOrDefault(

u => u.Name.Contains("Matt Vogel"));

conn.Delete(user);

}

In the example shown, there is only a single instance of a user named, “ Matt Vogel,” which is being deleted. However, if programmer wanted to delete an entire table, then the programmer would have to do the example shown below.

**using** (SQLiteConnection conn = **new** SQLiteConnection(App.DatabaseLocation))

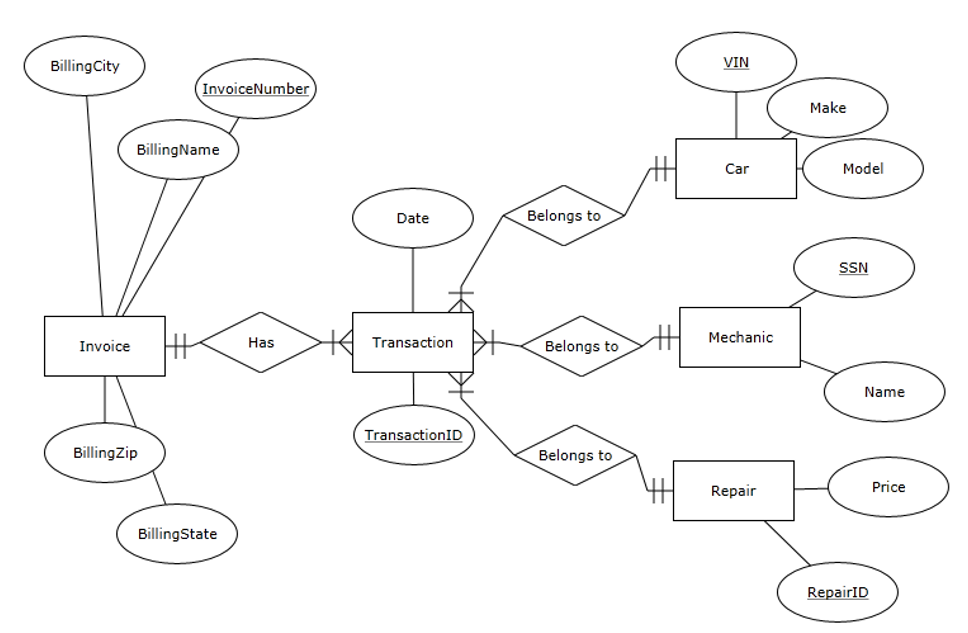
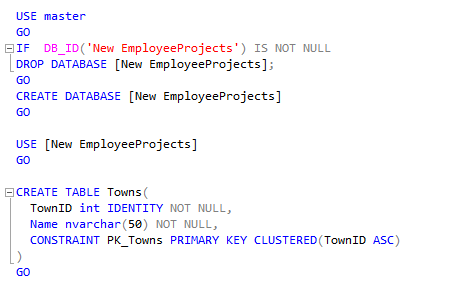
{

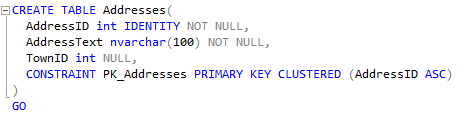
conn.DropTable<User>();

}

## How to Create an SQL Database

What will be Shown be next will be a brief example of creating an SQL database and to give the reader an idea of the effort involved in creating one.

1. Plan out the Database
   1. In order to use SQL there is a number of steps involved. First would be to plan the database using an Entity Relationship Diagram (ERD). Once a database is created, it become tricky to make changes to it without damaging the data stored in it. Therefore, planning out the database first using an ERD is helpful in catching errors before they happen. An example of an ERD is shown below.
   2. From the ERD above, there is a level of complexity when creating a relational database. The programmer must understand what level of relationship an entity should have with another entity, what entities should know of each other, and how data should be split up. When creating a database, a programmer needs to plan these things out or encounter errors when he/she relates the wrong entities.
2. Create the Database
   1. To create an SQL Database requires an SQL Server installation and then the Installation of SQL Server Management Tools (Will be covered later in this section). However, once that is installed, the creation of an SQL Database is quite easy. An example is shown below.  
      
   2. What is shown above is SQL that creates a Database called New EmployeeProjects and creates a Table called Towns. Whenever a table is created an ID is created as well. This allows for references to different tables and thus a relationship between different entities to be created. And example of a relationship is shown below.



* 1. The table Addresses has an AddressID but also a TownID. This means that this table has a relationship to the Towns table.

In SQL there are a lot of key words a programmer can use to create different columns in a table to hold different types of data and to specify how the data is stored. There can be restrictions placed and also rules created for data storage and updating the database. However, what is shown here is an oversimplified database to help the reader understand the basic idea in creating a database in SQL.

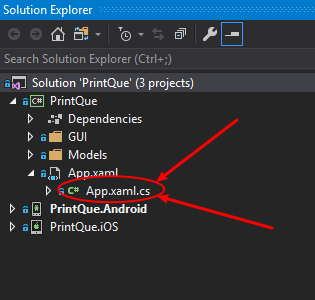
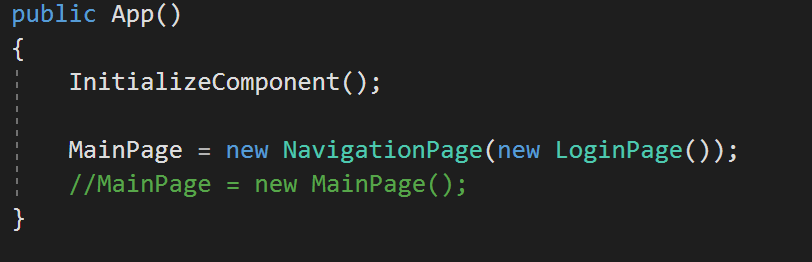
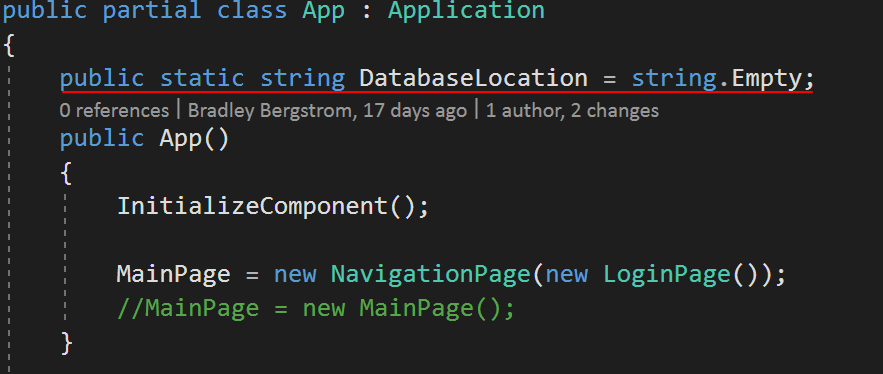
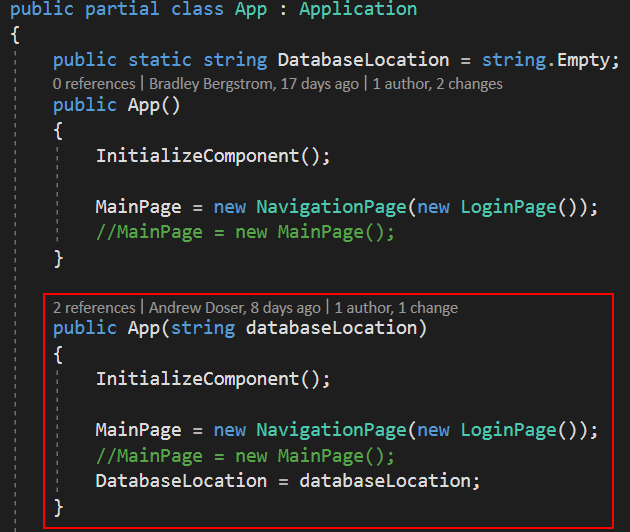
## Suitable Platforms for SQL

SQL is suitable for platforms such as Desktop Computers or Web Applications. This is due to SQL requiring more processing power than a phone could produce when running SQL queries. It is also because SQL Databases usually hold large amounts of data especially in an enterprise environment.

The platform that was chosen for an SQL database for the 3D printer Application was the Web Application. Due to the complexity of the database and how much power the Admin needed when managing the database, the team decided on using an SQL database for the Web Application of the 3D printer Application.

## How to Create an SQLite Database

Since SQLite does not use a server, but rather embedded into applications, there are many ways to create an SQLite database. However, there are still some similar concepts that can used across all applications. The following example will be a simplified way of how to create an SQLite database in Xamarin using the programming language C# for deployment to the Android phone operating system.

1. First Create a database location string.
   1. Any SQLite Database requires a database location string. This is for SQLite to know where it should put the data and where it should read the data. If there is no location string, then using SQLite becomes drastically difficult. Thus, to create database location string in Xamarin, there several steps to follow.
   2. First locate the app constructor in the solution explorer.  
      
   3. Open App.xaml.cs  
      
      1. What is shown above is a basic App constructor. The programmer will need to change the app constructor to take in a string so that they can pass the location of where to store the database.
   4. Create Database Location String in App.xaml.cs  
      
   5. Create new constructor  
      
   6. Now go to main activity in the Android project.  
      