# Project #3: Multi-User BikeHike App

**Complete By:** Monday, July 23rd @ 11:59pm
**Assignment:** completion of following programming exercise
**Policy:** Individual work only, late work *is* accepted (see "Policy" section at end of doc for more details)
**Submission:** electronic submission of VS project folders to BB

## Programming Exercise: A Multi-User BikeHike App

In project #1 you designed and populated a BikeHike database. In project #2, you designed a Windows desktop app to allow for the rental of bikes. Here in project #3, you're going to extend this app, in particular to properly support the notion of multiple users accessing the BikeHike database concurrently.

## Before you start --- our solution, or yours?

You have the option of building on your solution to project #2, or building upon our solution. Either approach is fine, the choice is entirely yours. If you prefer to use our solution, it can be downloaded from the course web page. Note that you'll want to download our solutions to both project #1 and project #2, since you will need to modify our database design:

Project #1: https://www.dropbox.com/s/ispksmvwsoc89pt/CreateDBApp.zip?dl=0

Project #2: https://www.dropbox.com/s/c4z1on1i2ctvyi2/BikeHikeApp.zip?dl=0

You'll want to download, extract, and copy each app to your desktop (or some non-cloud based folder on your local machine). Open and run project #1 to generate a BikeHike database. Then place a copy of this database into the bin\Debug folder of project #2, open the project, and make sure it runs successfully.

## Requirements

The main goal is to extend the BikeHike app to work correctly in the presence of multiple users accessing the database concurrently. You'll simulate multiple users by running multiple instances of the program simultaneously. To this end, here are the requirements for your app here in project #3. I would recommend addressing the requirements in this order:

1. If you are building upon your own solution, add a "Reset Database" button or menu item. This makes testing much easier by resetting the database back to its initial state. What needs to be done depends on your database design, but for the BikeHike design discussed in class you need to do 3 things:

   a. Delete all rows in the RentalDetails table
   b. Delete all rows in the Rentals table
   c. Set all Rented bits to 0 (not rented) in the Bikes table

   Our solution to project #2 already provides this functionality via the File menu.

2. Add 2 or more indexes to your database to speed up the execution of the app (the database is too small to actually see any performance improvement, but the exercise is still worthwhile). You'll need to modify your project #1 DDL to create the indexes, and then take advantage of these indexes by modifying your SQL queries in project #2. You'll want to add "WITH" hints to make sure the DBMS actually uses your indexes. Searches and joins are likely candidates for taking advantage of indexes.

   **Note**: you must use "WITH" hints in your SQL so we can see what indexes you added; this is necessary for grading purposes (and to ensure that SQL Server actually uses your indexes).

3. Add transactions to the application where appropriate; this implies for both (a) easier error handling, and (b) correct operation in the presence of multiple users. At a minimum, the "Reset" functionality from step 1 above should be wrapped in a transaction, as well as the operations to rent bikes and return bikes. You are required to use (a) ADO.NET transactions, and (b) Try-Catch-Finally to properly Commit or Rollback in the presence of exceptions (and retry at least 3 times in the presence of deadlock).

   **Note**: if you are using the DataAccessTier, you'll need to redesign the API since in its current form the DataAccessTier opens and closes the connection per query. When executing a transaction, the *same* connection must be used so that the transaction remains in effect across all the queries --- closing a connection prematurely will cause the current transaction to rollback. This is a really good --- and realistic --- API design problem.

4. Add a way to simulate and test your multi-user app. In particular, add a text box so the user can enter a delay in milliseconds (e.g. 1000 => 1 sec). Then add delays inside your transaction-processing code. For example, when renting bikes, add a delay after you check if the customer and bikes are available for rent, but before you update the database. Likewise, when returning bikes, add a delay after you update the database to make sure other users don't see these changes until the Tx is committed. Here's the C# code to perform a delay, where "txtTimeInMS" is the name of the text box:

```
int timeInMS;
if (System.Int32.TryParse(this.txtTimeInMS.Text, out timeInMS) == true)
  ;
else
```

```
            timeInMS = 0;  // no delay:

        System.Threading.Thread.Sleep(timeInMS);
```

5. Lastly, once steps 1-4 are done, redesign the app to be more object-oriented. In particular, design and add a **RentalCart** class to keep track of the customer and the bikes that he/she wants to rent. When the "Rent" button is pressed, the info for the rental must come from the RentalCart object, not the UI. The RentalCart class can be as simple as a constructor + some data members, or more involved to include methods that perform various operations (such as the rental).

6. **Graduate Students**: read about stored procedures, and add a stored procedure to the database (via project 1) that takes a customer ID and if that customer ID is out on a rental, returns all bikes associated with that rental. The stored procedure should update the database accordingly, and return the total price of the rental; the stored procedure must use a transaction. The UI should provide two different "return" buttons --- one that executes using dynamic SQL as in project 2, and another that calls the stored procedure. The user should be able to use either button (i.e. both should work correctly to return from a rental).

   **Note**: undergrads are encouraged (but not required) to implement this feature. If an undergrad successfully implements this feature, he/she will earn a free late day on a future project (which can also be applied to this project or an earlier project).

## Electronic Submission

First, we want you to submit project 1 *and* project 3 --- since both were modified for this assignment. Create a folder named say **P3Both**, and then place copies of your project 1 and project 3 folders into P3Both. Next, be sure there exists a header comment at the top of your Project 3 main C# source code file ("Program.cs") along the lines of:

```
//
// Multi-user BikeHike Windows app, using transactions.
//
// <<YOUR NAME HERE>>
// U. of Illinois, Chicago
// CS480, Summer 2018
// Project #3
//
```

To submit, find your combined **P3Both** folder. Create an archive (.zip) of this folder: right-click, Send To, and select "Compressed (zipped) folder". Then submit the resulting .zip file on Blackboard: open "Projects" and submit under "Project 03: Multi-User BikeHike App".

Your program should be readable with proper indentation and naming conventions; commenting is expected where appropriate. You may submit as many times as you want before the due date, but we grade

the last version submitted. This implies that if you submit a version before the due date and then another after the due date, we will grade the version submitted after the due date — we will *not* grade both and then give you the better grade. We grade the last one submitted. In general, do not submit after the due date unless you had a non-working program before the due date.

## Policy

Late work *is* accepted. You may submit as late as **24** hours after the deadline for a penalty of 10%. After **24** hours (i.e. after Tuesday, July 24th @ 11:59pm), no submissions will be accepted.

All work is to be done individually — group work is not allowed. While I encourage you to talk to your peers and learn from them (e.g. via Piazza), this interaction must be superficial with regards to all work submitted for grading. This means you *cannot* work in teams, you cannot work side-by-side, you cannot submit someone else's work (partial or complete) as your own. The University's policy is described here:

https://dos.uic.edu/docs/Student%20Disciplinary%20Policy%2017-18%20(FINAL).pdf

In particular, note that you are guilty of academic dishonesty if you extend or receive any kind of unauthorized assistance. Absolutely no transfer of program code between students is permitted (paper or electronic), and you may not solicit code from family, friends, or online forums. Other examples of academic dishonesty include emailing your program to another student, copying-pasting code from the internet, working in a group on a homework assignment, and allowing a tutor, TA, or another individual to write an answer for you. Academic dishonesty is unacceptable, and penalties range from failure to expulsion from the university; cases are handled via the official student conduct process described at http://www.uic.edu/depts/dos/studentconductprocess.shtml.