

ASSIGNMENT #3: AIRLINE ADMIN SYSTEM

Initial Submission Due: Monday, Aug. 6, by 11:59 pm

Your final two assignments involve implementing a simple airline reservation system. In this part you will create a WinForm for use by airline administrators in selecting planes and creating flights. In the next part (Assignment #4, due Aug. 15), you will create a web site that allows customers to book seats.

Overview of Use Cases

To give you an idea of the scope of the project, here is a list of the primary use cases for the two assignments. That is, these are the activities that users of the software must be able to carry out. Details are discussed later in this document.

Administrators (Assignment 3)

- Select planes for the airline from a list of available types
- Create flights
- Inspect reservations by flight

Customers (Assignment 4)

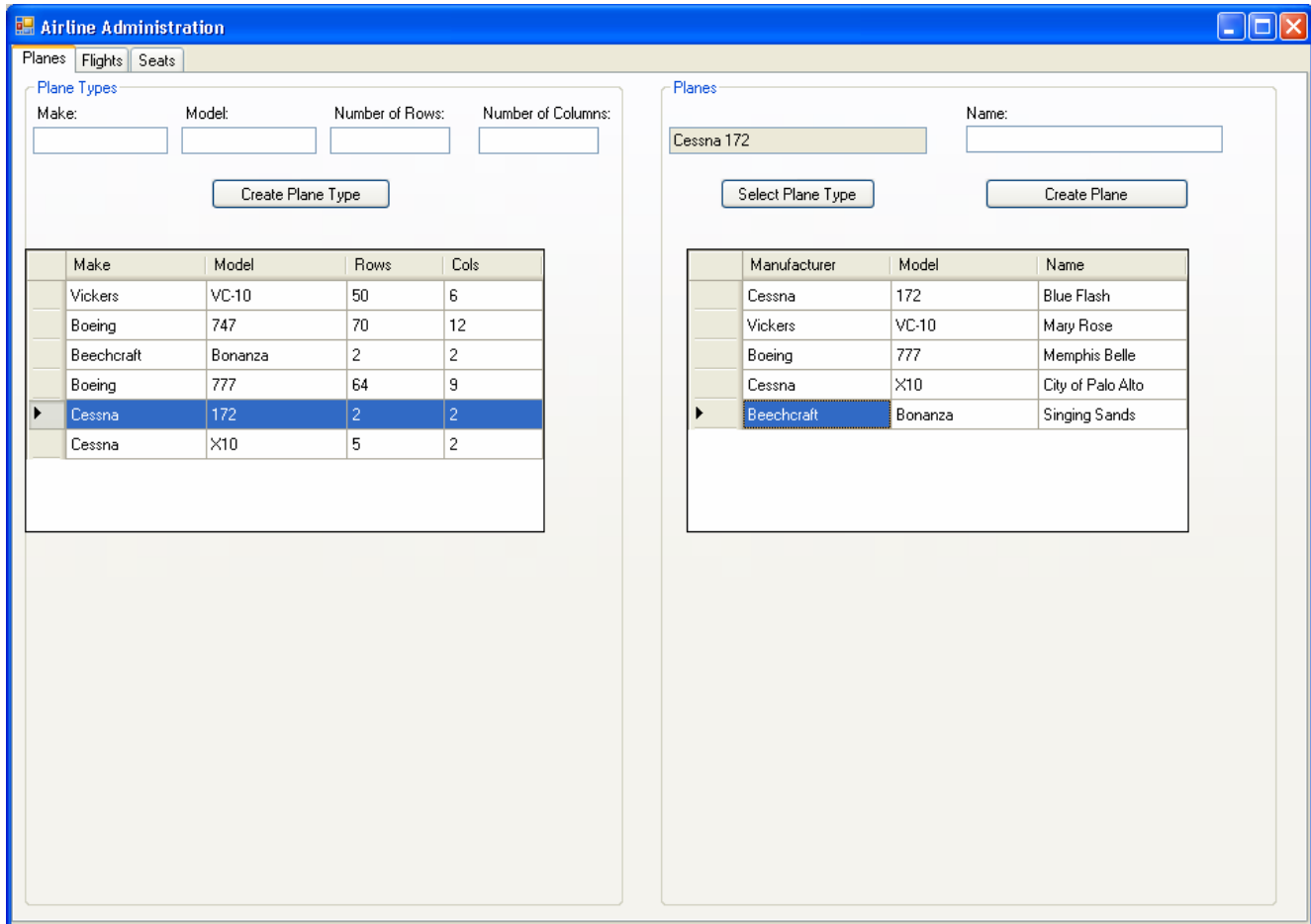
- Search for flights
- Establish a username/password
- Log in
- Select a flight and specify passengers
- See a seating chart and make selections
- Receive notification of concurrency violations during seat selection
- See the reservations they have made
- Log out

There will be ample opportunities for extending these basic capabilities.

The Administrator Application

The Admin App will be a WinForm that accesses and updates a SQL database. The easiest way to get a feel for it is to take a look at the version we have created. You should use these screen shots as a strong guideline as to what we are expecting in the final product.

The Admin App uses a tabbed interface to separate the three main activities. When the program starts, the Planes tab should appear:



There are two group boxes on this tab. PlaneTypes, on the left, shows a grid with details of the types of airplanes that the airline may add to its fleet. We will give you some plane types to start with, but we are also indicating that the administrator is able to add new types to the database. The latter capability is not actually realistic and you needn't implement it, but you may find it helpful for your testing to be able to create planes of particular sizes right here in the application.

On the right, the Planes group box gives the administrator the ability to place planes in the fleet. The administrator first selects a plane type in the Plane Types grid by clicking on or in the row, then presses the Select Plane Type button in the Planes Group box on the right. That cause information about the plane (here, "Cessna 172") to appear in the text box above the button.

Since the airline might have several planes of the same type, each plane is given a name by entering it into the Name text box, and then the plane is added to the fleet. In the picture, the administrator is about to provide a name for a second Cessna 172.

Next, the administrator would move on to the Flights tab, shown below. It is always possible to return to the Planes tab to add more planes.

The screenshot shows the 'Airline Administration' application window with the 'Flights' tab selected. The interface is divided into three main sections:

- Airports:** A list of airports with columns 'AirportCode' and 'DisplayName'. A filter dropdown is set to 'Code'. The 'Show All' button is visible.
- Planes:** A table showing plane details with columns 'Manufacturer', 'Model', and 'Name'. The table contains the following data:

Manufacturer	Model	Name
Cessna	172	Blue Flash
Vickers	VC-10	Mary Rose
Boeing	777	Memphis Belle
Cessna	X10	City of Palo Alto
Beechcraft	Bonanza	Singing Sands
- Flights:** A form for creating a new flight with fields for 'Flight Number', 'Select Origin', 'Select Destination', 'Departure Time', and 'Arrival Time'. A 'Generate Flight' button is present. Below the form is a table of existing flights:

Number	Origin	Destination	DepartureTime	ArrivalTime
7	SFO	AKL	5/21/2007 7:00 PM	5/23/2007 5:30 AM
8	AKL	SFO	5/28/2007 4:00 PM	5/28/2007 5:20 PM
42	CHC	AKL	5/24/2007 6:30 PM	5/24/2007 7:30 PM
233	AKL	DUD	5/30/2007 4:00 PM	5/30/2007 5:31 PM
234	DUD	AKL	5/30/2007 7:11 PM	5/30/2007 8:50 PM
240	CHC	IVC	7/26/2007 12:16 PM	7/26/2007 1:16 PM
306	AKL	CHC	8/20/2007 4:00 PM	9/27/2007 5:41 PM
4350	LHR	CBG	7/30/2007 10:05 AM	7/30/2007 10:48 AM

Here there are three group boxes: Airports, which provides airport codes for all the world's airports; Planes, which shows the planes in the fleet; and Flights, which shows all the existing flights and allows new ones to be created. The steps for creating a new flight are as follows:

1. Enter a new flight number in the text box.
2. Select an airport in the Airports list, and click Select Origin. This causes the code for the airport to be entered in the box to the right of the button. To facilitate finding the correct code, the code list can be filtered by Code or by name—more on that later.
3. Select another airport from the list and click Select Destination.
4. Use the two DateTimePicker controls to select departure and arrival times for the flight.
5. Click on a plane in the Planes grid, then click on Select Plane to choose the plane for the flight. The plane name should appear in the text box next to the button.
6. Click Generate Flight to create the flight, which will then appear in the grid. Note that we have not shown the plane in this grid, but that information is, of course, recorded, and adding information about the plane would be a good idea.

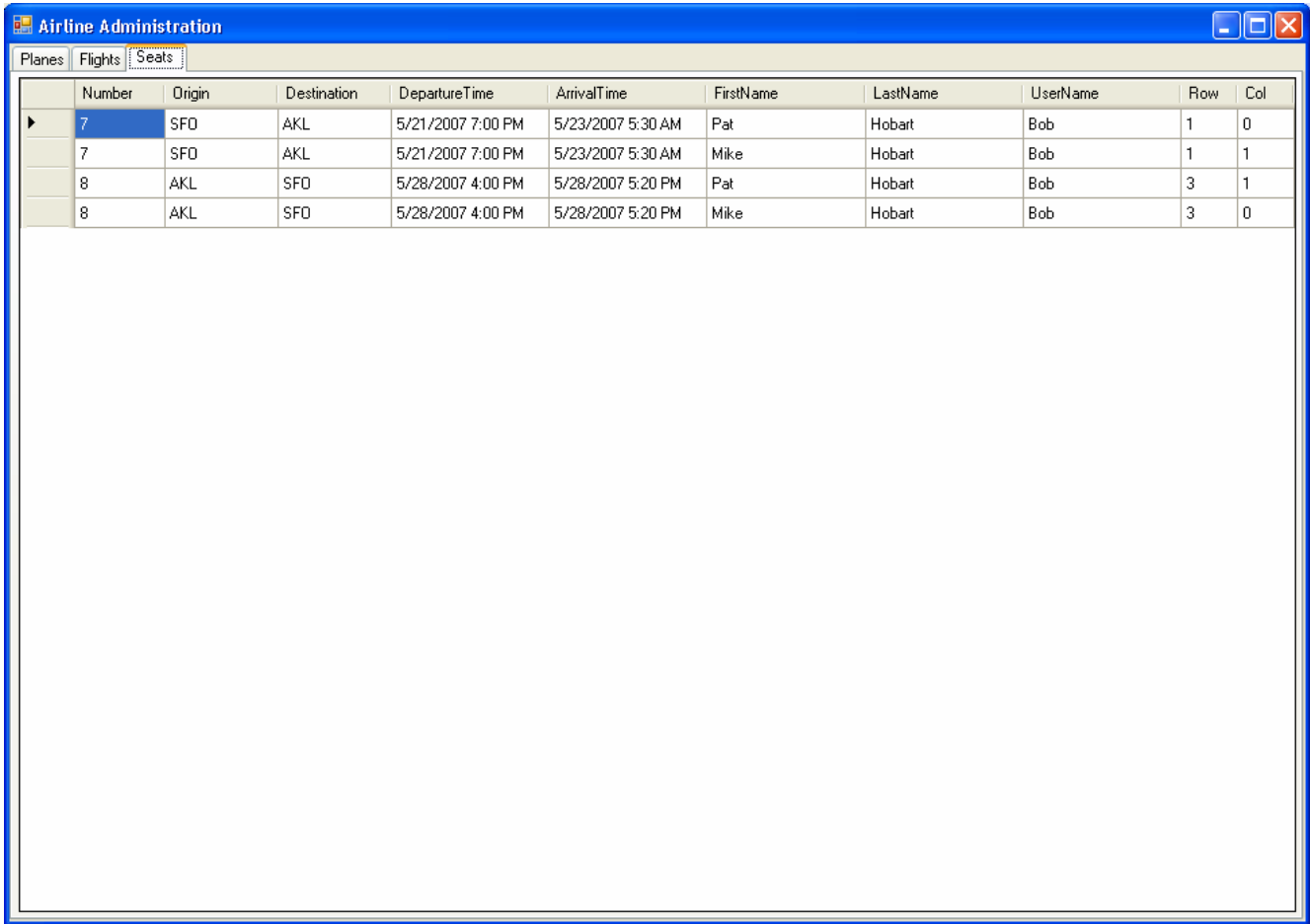
Since there are 2606 airport codes in our list, we need to provide a way to filter them. So, rather than have you your `airportsBindingSource` get its data from the `Airports` table, you might want to use a `DataView` instead (check this class out in the documentation). You won't find this in the Toolbox by default, but you can add it by right-clicking there and selecting `Choose Items...`. Or, simply declare a `DataView` in your code at the Form level. Then, assuming the `DataView` is called `airportView`, you would have a method like this:

```
private void FilterAirportView(string columnName, string filterString)
{
    airportView.Table = airlineDataSet.Airports;
    airportView.RowFilter = columnName + " LIKE '" + filterString + "'";
    airportView.Sort = columnName + " ASC";
    airportsBindingSource.DataSource = airportView;
    airportsBindingSource.DataMember = null;
}
```

You would call this method when the `TextChanged` event occurs in the filter text box (to the left of the `Show All` button in the diagram above). You would provide the appropriate column name ("`AirportCode`" or "`DisplayName`") depending on how the `ComboBox` to the left is set, and you provide the filter string from the text box. The result is that the `DataGridView` that is bound to the `airportsBindingSource` will get a filtered version of the data rather than the whole table, and the display will change as each character of the filter is typed.

If you are filtering by codes, you should add the wildcard character `'%'` to the end of the filter string before calling the method above. If you are filtering by name, it is probably best to add the wildcard to the beginning and end, so that the search works like the `string.Contains` method. An administrator wanting to see only codes from New Zealand could filter by display name, type `"zea"`, and there would be the results.

Finally, the administrator can view the current seat reservations. The grid on the `Seats` tab will look like the one on the next page:



The screenshot shows a window titled "Airline Administration" with three tabs: "Planes", "Flights", and "Seats". The "Seats" tab is active, displaying a table with the following data:

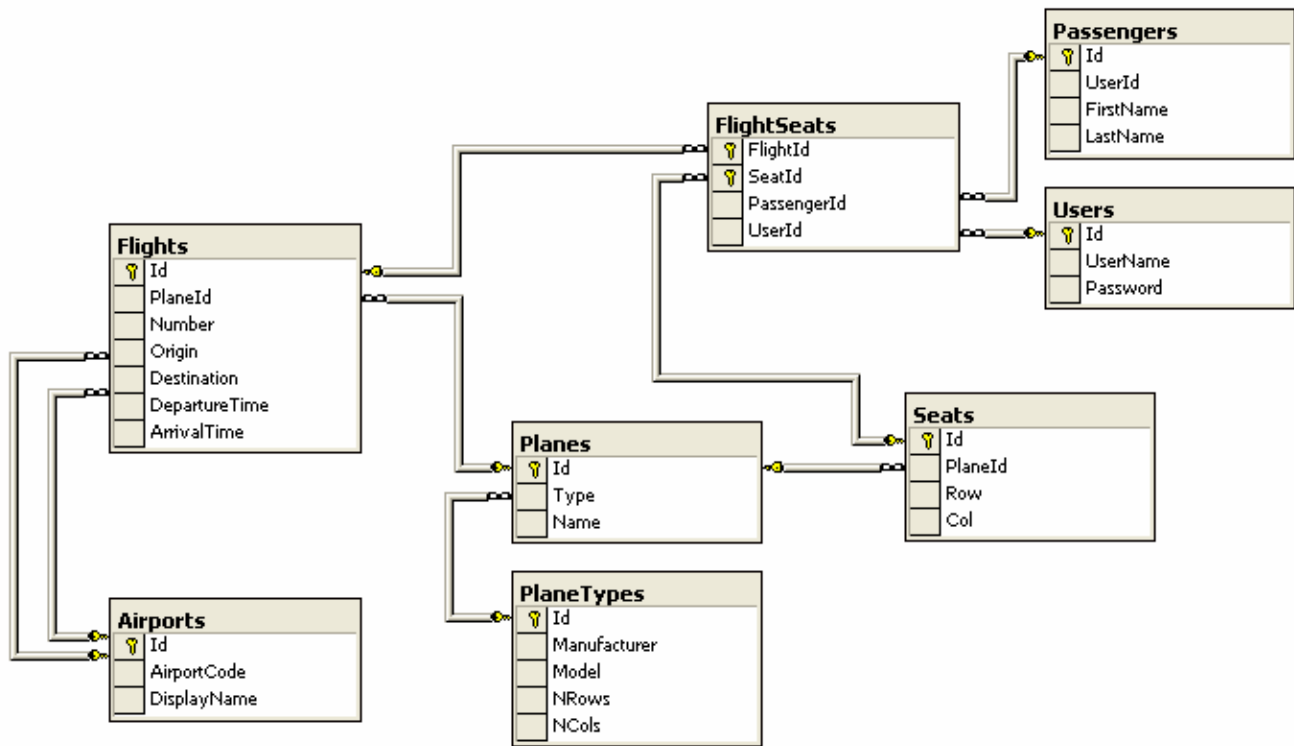
	Number	Origin	Destination	DepartureTime	ArrivalTime	FirstName	LastName	UserName	Row	Col
▶	7	SFO	AKL	5/21/2007 7:00 PM	5/23/2007 5:30 AM	Pat	Hobart	Bob	1	0
	7	SFO	AKL	5/21/2007 7:00 PM	5/23/2007 5:30 AM	Mike	Hobart	Bob	1	1
	8	AKL	SFO	5/28/2007 4:00 PM	5/28/2007 5:20 PM	Pat	Hobart	Bob	3	1
	8	AKL	SFO	5/28/2007 4:00 PM	5/28/2007 5:20 PM	Mike	Hobart	Bob	3	0

Until you have the Customer App running well enough to actually make some reservations, there will be no data to display here, so in your initial submission of the Admin App, you may leave this tab blank. As you can see, it will show not only the passenger names, but the username of the person who made the booking.

You should submit your final Admin App along with your Customer App, on **August 15**.

The Database

The database you will be using contains eight tables, related as shown in this diagram:



We will provide you with a script, called `Airline.sql`, which will create the database on your machine. Start Microsoft SQL Server Management Express, and use File | Open to open the file (if prompted, connect to your SQLEXPRESS server). Then click Execute in the Toolbar, and the database will be created. One good way to verify this is to expand the Airline database, right-click Diagrams, and create a new diagram. Add the tables shown above when in the wizard, and you should get a result similar to the above.

Now we will discuss each table, moving basically from the bottom of the diagram toward the top.

Airports. As you can see, this table holds the airport codes (e.g., SFO) and display names (e.g., San Francisco International Airport) for each airport, as well as an Id. We'll discuss the data type for the Id's in the various tables later, but the idea is that all references to a particular airport (such as in the Flights table) use the Id's, which are the primary keys for the Airports table. We will provide an XML file that you can use to populate your table with the world's airport codes. Just write a small program that uses the Airports table as a data source, and use the `ReadXML` method of the `DataTable` as shown in class. You will only run this once—at that point your table is ready to go.

PlaneTypes. Another basic table is the one that holds information about the different types of planes that are available. Again, we will provide an XML file with some starter data, so you might want to add code to fill this table to the small program mentioned above. You can leave it at that, or include

the capability for the administrator to add new kinds of planes. Note that we are taking a simplified view of the seating arrangements. Each plane has a certain number of rows, and a certain number of seats on each row. We refer to the latter as the number of "columns", for lack of a better term. In any event, you do not have to worry about the situation where different rows on a given type of plane have different numbers of seats. As with most tables, each plane type has an Id.

Planes. The Planes table is updated when the Administrator clicks the Create Plane button shown earlier in the UI diagram. Each row contains in the Type column the Id of the appropriate plane type.

Seats. When a new plane is created, a row is also added to the Seats table for each seat on the plane. The number of rows and columns can, of course, be found from the PlaneTypes table. For example, if a small plane has 5 rows and 2 columns, then 10 rows would be added to the Seats table when the plane is created, with each row containing the `PlaneId` of the newly created plane. The purpose of the Seats table is to avoid redundant storage of row and column information when reservations are made. For example, if a given plane is scheduled for 10 flights, then potentially we might store the row and column information for each the person in, say row 2 seat 1, 10 times. We avoid this by giving each seat an Id in the Seats table, which we can then refer to in other tables. The Seats table might also be used to store other information, such as when the seat was last upholstered, or what celebrities have sat in it, but we do not include that here. Just to be sure you are clear on this table, if you think about the entire fleet of planes owned by the airline, the total number of seats on them is the number of rows in the Seats table, regardless of how many flights the planes are assigned to.

Flights. A row is added to the Flights table when the Administrator clicks the Generate Flights button. If you check the UI diagram above, you will see that this is fairly straightforward once the required information is filled in.

FlightsSeats. Somewhere we have to allow for the fact that each physical seat on a plane may, at any give time, be involved in several bookings if the plane is assigned to several flights. We do this in the `FlightsSeats` table, to which rows are added for each seat on a plane every time a flight is created using the plane. These entries represent bookable seats, and each holds the appropriate `SeatId`. Initially the `PassengerId` and `UserId` of these rows will be the database value *Null*. That is not the same as the usual `null` in C#. You should look up `DBNull` in the documentation, and note that when you are preparing a `FlightsSeatsRow` for addition to the table, your typed dataset provides methods `flightSeatsRow.SetPassengerIdNull()` and `flightSeatsRow.SetUserIdNull()` for storing the *Null* values in the row. It also provides methods `flightSeatsRow.IsPassengerIdNull()` and `flightSeatsRow.IsUserIdNull()` for testing for *Null* values. If you use these methods, you will not confuse the two kinds of "nulls". One other thing to notice about this table is that it has a multi-column key. Flights have lots of seats, and a given seat on a plane can be booked on lots of flights. So to specify a particular entry in this table, you have to refer to a particular `SeatId` and a particular `FlightId`. In the typed dataset, this table will have a method called `FindByFlightIdSeatId`.

Users. The Users table is filled in by the Customer App, which you will not deal with in this part of the assignment. Each customer creates a `UserName` and a `Password`, with the latter being stored in the table in an encrypted form.

Passengers. This is the final table, also filled in by the Customer App. The idea is that users (i.e., customers) may make reservations for people other than themselves, and this table stores the passenger names and the user who created the passenger. We do not require the passenger names to be unique—if two users book seats for "John Smith", we will just make two entries in the Passengers table.

Additional Tables

If you look back at the UI, you will see how the tables in the database support the various grids of information shown. You will also notice that in some cases then information shown in a single grid comes from more than one table. To handle these cases, you will probably want to use the technique shown in class for adding new TableAdapters (and thus tables) to your data source, based on the JOIN of two or more tables.

For example, the Flights display will have to join the Flights table to the Airports table twice, once to get the airport code for the origin, and once for the destination. You do this by using an alias for one of the references to the Airports table, something like this (Airports2 is the alias):

```
SELECT Flights.Number, Airports.AirportCode AS Origin, Airports2.AirportCode AS
       Destination, Flights.DepartureTime, Flights.ArrivalTime
FROM Flights INNER JOIN Airports ON Flights.Origin = Airports.Id
INNER JOIN Airports AS Airports2 ON Flights.Destination = Airports2.Id
```

Note that in the first line, "AS Origin" and "AS Destination" are there merely to provide the right column names in the result. On the last line, "AS Airports2" provides an alias for the Airports table for use in the second JOIN. If you decide to add information about the plane to the grid, then another JOIN will be involved. As demonstrated in class, the Query Designer of Visual Studio is very good at helping you get these queries right.

A UI Hint

There is a problem that appears more than once in this design. We will use the Flights group box as an example.

When the Select Plane button is clicked, the name of the selected plane should appear in the appropriate text box. The Administrator has no interest in seeing the `PlaneId` (for reasons that will become obvious in the next section). But when it is time to create the flight, you will find that you actually need the plane's row from the Planes table, so that you have the `PlaneType Id` for looking up the number of rows and columns. The problem is: how do you get the `PlanesRow`?

If we make the restriction that the names are unique, you could search for the row by name, but we don't rule out the possibility of two planes having the same name. We would prefer to do the lookup in the Planes table using the plane's Id, but the Id does not appear to be in the grid that is being shown.

The answer is to use the technique shown in class (now you begin to see that all those demonstrations were not randomly chosen :-). You can include the `PlaneId` as a column in the joined table, then go into the `ColumnsCollection` for the `DataGridView` in the Visual Studio designer, and mark that column as invisible. Now the `Id` will not be visible, but it will be there, and you can use it with the `Planes` table's `FindById` method to get the `PlanesRow`.

We will also mention that every control, such as a text box, has a property of type object called `Tag`. You can set this to be any object, then access it later and cast it to its original type. This can be a handy way to keep around some information that you don't want to display on the screen.

What Are the Id's?

Now we come to the question of the `Id` fields in the various tables. We know that we need to `Ids` to be unique, since they will be the primary keys for the tables.

Different database practitioners approach this in different ways. One way is to say that in the SQL database, the column is an "Identity", and to provide an initial value and an auto-increment. Then every time a row is added, the SQL server will automatically provide the next value for the `Id` (and no `Id` need be provided by the client when adding a row). This works just fine, but the problem is that the client can't be sure what `Id` will be assigned, so the only safe thing to do is fetch the row back immediately after adding it, which requires another round trip to the database. Code to do this will be generated automatically if you design the database to use an Identity column.

If we want to avoid the extra round trip, we would like the client to be able to generate a key that is guaranteed to be unique, and give it to the database along with the rest of the values when a row is added. We can do this by generating a Globally Unique Identifier, or "Guid".

Guids are long integers that are generated from some combination of things like the MAC address of the machine and the time of day. The chances of two Guids ever being the same are so small that we can consider them to be unique across all machines. There is a `struct` called `Guid` defined in the `System` namespace, and the static method `Guid.NewGuid()` returns a new `Guid`.

The `Guid` approach is used in many commercial settings, and that is what we are going to use in this assignment. The corresponding SQL data type is called a "uniqueidentifier". Methods in the typed dataset such as `Seats.AddSeatsRow` will want an `Id` as their first parameter, and you can provide this by calling `Guid.NewGuid()`.

The downside of `Guids` is that they should never be shown to the user of the software, since they are meaningless, large numbers. If you call `ToString()` on a `Guid` you get something like "BCFF1C64-9D83-488C-B82E-67B8214AD8B5". So we make sure to hide the `Id`'s even if they are in the grids. This also means that debugging can be a little difficult until you get used to it. You sometimes do have to look at `Guids`; for example, you might want to make those `Id` columns visible during debugging. However, the `Guids` almost always differ in their first four characters, so using them is not as troublesome as it might seem.

For a good discussion of the pros and cons of using GUIDs and other kinds of keys, see [Programming Microsoft ADO.NET Applications, Advanced Topics \(2005 Edition\)](#) by Glenn Johnson (MS Press).

Getting Started

The main thing here is to get started early. You may need to write a few test programs to learn the appropriate techniques. You also need to download and run the Airports.sql script to create the database, and you need to write and run the little helper program to read the XML data before you can work on the real program. You will find the files in the Assignments section of the class web site.

Your Admin App with the first two tabs functional is due in a week. Then you will have two weeks to do the Customer App and finish the Seats tab of the Admin App.

Note that we have not discussed cancelling flights or removing them from the database when they are completed. Those would make nice extensions, as would some filtering options on the Seats tab.

Good luck!