**Lottery Example**

This Example shows a basic structure for Reading a single customer for an application (emulating a logon but with no logon feature at the moment). The Customer data can be amended.

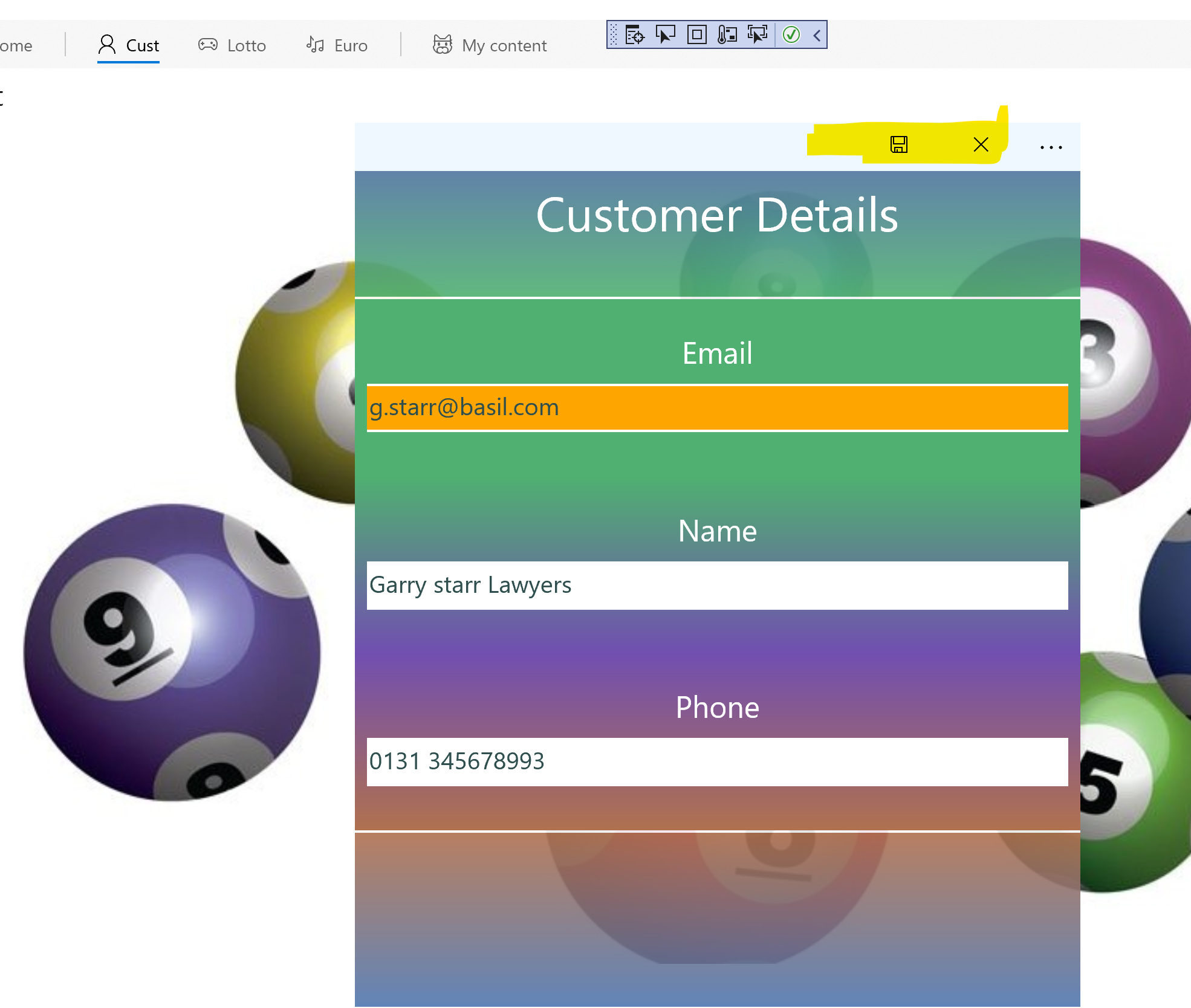
This example is using an sqlite database for example purposes of how the sqlite part will work, a locl database is not best suited to a lottery app. This would have to work alongside a central database for the lottery app to operate in reality. The sqlite part would mainly store paid for purchased historic lottery tickets.

This example follows the Microsoft.EntityFrameworkCore framework to interact with database operations. This follows part of the MVVC architecture.

In this architecture the entities within the database are stored within a temporary repository in the application. This repository data would have to be written back to the database if changes are to be saved.

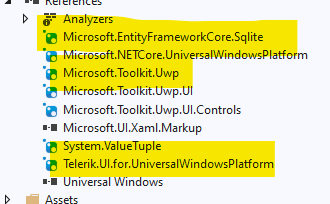
For now the application is very simple, this is to try to show you the code behind the MVVC and the repositories.

The application has a main page with a navigation system and the working Customer page only. The logged on customer is displayed on the screen and the details can be updated and then saved by pressing the save icon or cancelled by pressing the x icon. There is also a save/don’t save/cancel dialogue box on exit of the page if changes are made and not saved or cancelled.



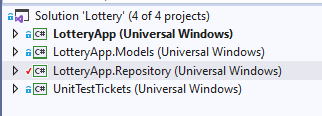
**For the project the following Nuget Packages were installed:**

Add the Nuget packages required:

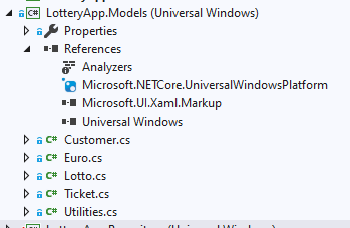


**The Project structure for the solution will change as follows:**

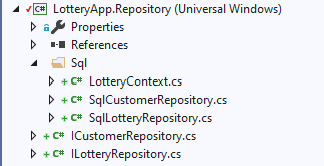
* LotteryApp is a UWP project file, storing the aplication logic
* LotteryApp.Models is a UWP Class Library, storing the business models
* LotteryApp.Repository is a UWP Class Library, storing the Database models
* UnitTestTickets is a UWP Test project, storing the test logic



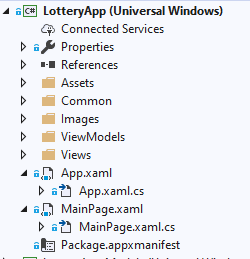
LotteryApp.Models holds the following classes, emulating the business model:



LotteryApp.Repository: So far has the following files, these are to provide the interactions with the database backend using SQL via Entity Framework Core 2.0



Lottery App will have the main logic for the App and the User Interface, so far the files are as follows:



In the Repository project within the Sql folder update the file named LotteryContext.cs

This is a class which derives from DbContext and contains DbSet<TEntity> properties for each entity in the model If the DbSet<TEntity> have a public setter, they are automatically initialized when an instance of the derived context is created.

DbContext instance represents a session with the database and can be used to query and save instances of your entities. DbContext is a combination of the unit of work and repository patterns.



using System;  
using System.Collections.Generic;  
using System.Linq;  
using System.Text;  
using System.Threading.Tasks;  
using LotteryApp.Models;  
using Microsoft.EntityFrameworkCore;

namespace LotteryApp.Repository  
{  
 public class LotteryContext : DbContext  
 {  
 /// <summary>  
 /// Creates a new DbContext.  
 /// </summary>  
  
 public LotteryContext(DbContextOptions<LotteryContext> options) : base(options)  
 { }

/// <summary>  
 /// Gets the Customers DbSet based on the Customer model.  
 /// </summary>

public DbSet<Customer> Customers { get; set; } // DbSet is used to query and save instances of the entity Customers

// \*\*\*\* Further DbSet’s would be added here for each entity to be accessed within the Lottery Application  
 }  
}

Customer.cs - Found in LotteryApp.Models

The Customer.cs Model is updated as follows to allow for it to work alongside the DbSet Customers and the entity in the database Customer.

The IEquatable<Customer> is inherited and the Equals method coded. This automatically checks to see if there is a customer with the same details and if it does it returns the Customer which is already there rather than creating a new instance of the customer.

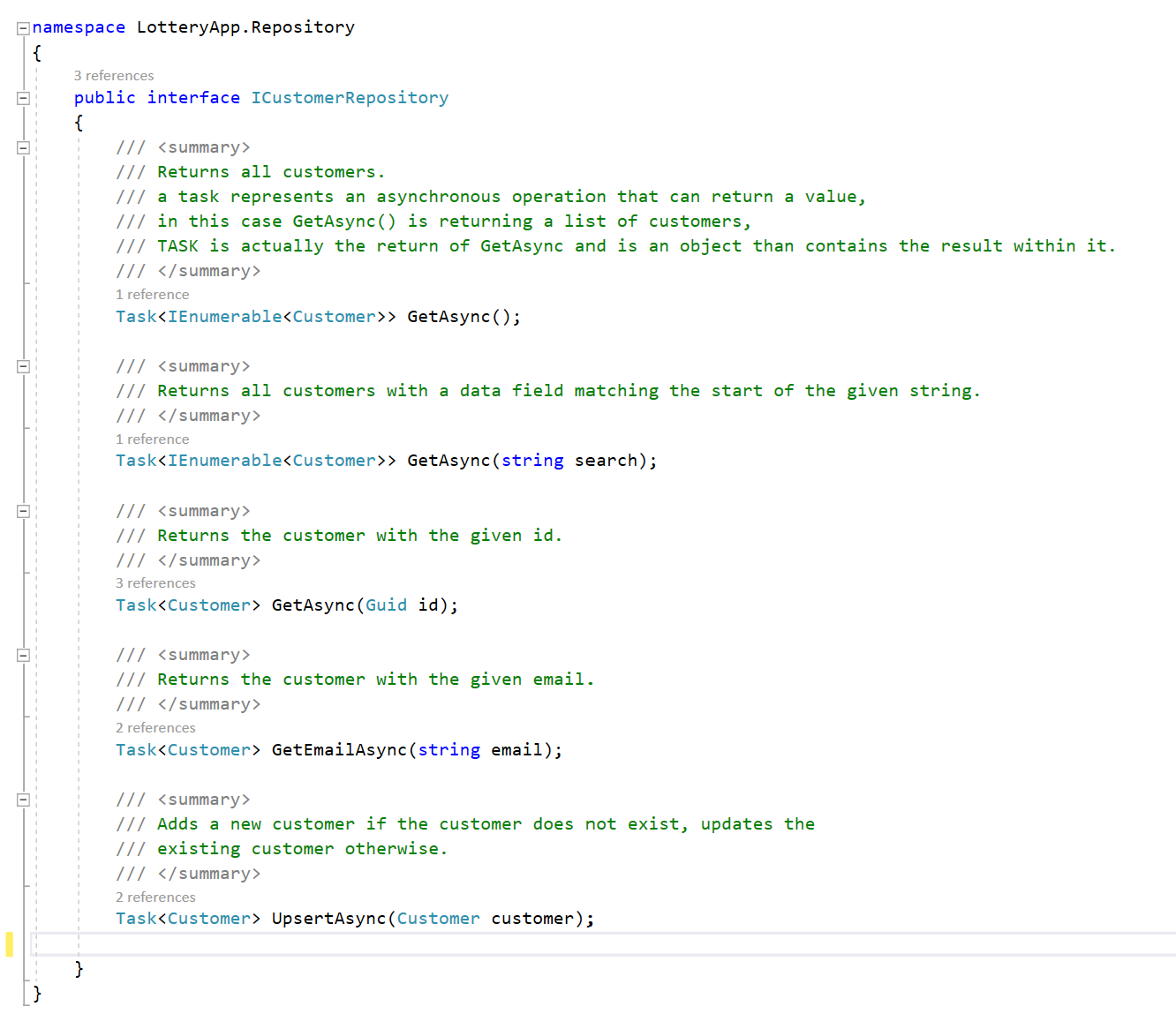
CustID has been added to match the database Entity, Guid will automatically create an ID in sequence.



ICustomerRepository.cs, defines methods for the customer interacting with the back end database, these methods are implemented in SqlCustomerRepository.cs

It defines properties and methods for the Customer to asynchronously read from the database and to handle the create/update/delete operations. Note: Task in this case is the return resultset object from the database and contains the data as part of the object.

Note: the part after task TASK<Customer> specifies what type of object is being returned from the database. In this case Customer is an instance of the modelled Customer.cs in the models folder.



namespace LotteryApp.Repository  
{  
 public interface ICustomerRepository  
 {

/// <summary>

/// Returns all customers.

/// a task represents an asynchronous operation that can return a value,

/// in this case GetAsync() is returning a list of customers,

/// TASK is actually the return of GetAsync and is an object than contains the result within it.

/// </summary>

Task<IEnumerable<Customer>> GetAsync();

/// <summary>

/// Returns all customers with a data field matching the start of the given string.

/// </summary>

Task<IEnumerable<Customer>> GetAsync(string search);

/// <summary>

/// Returns the customer with the given id.

/// </summary>

Task<Customer> GetAsync(Guid id);

/// <summary>

/// Returns the customer with the given email.

/// </summary>

Task<Customer> GetEmailAsync(string email);

/// <summary>

/// Adds a new customer if the customer does not exist, updates the

/// existing customer otherwise.

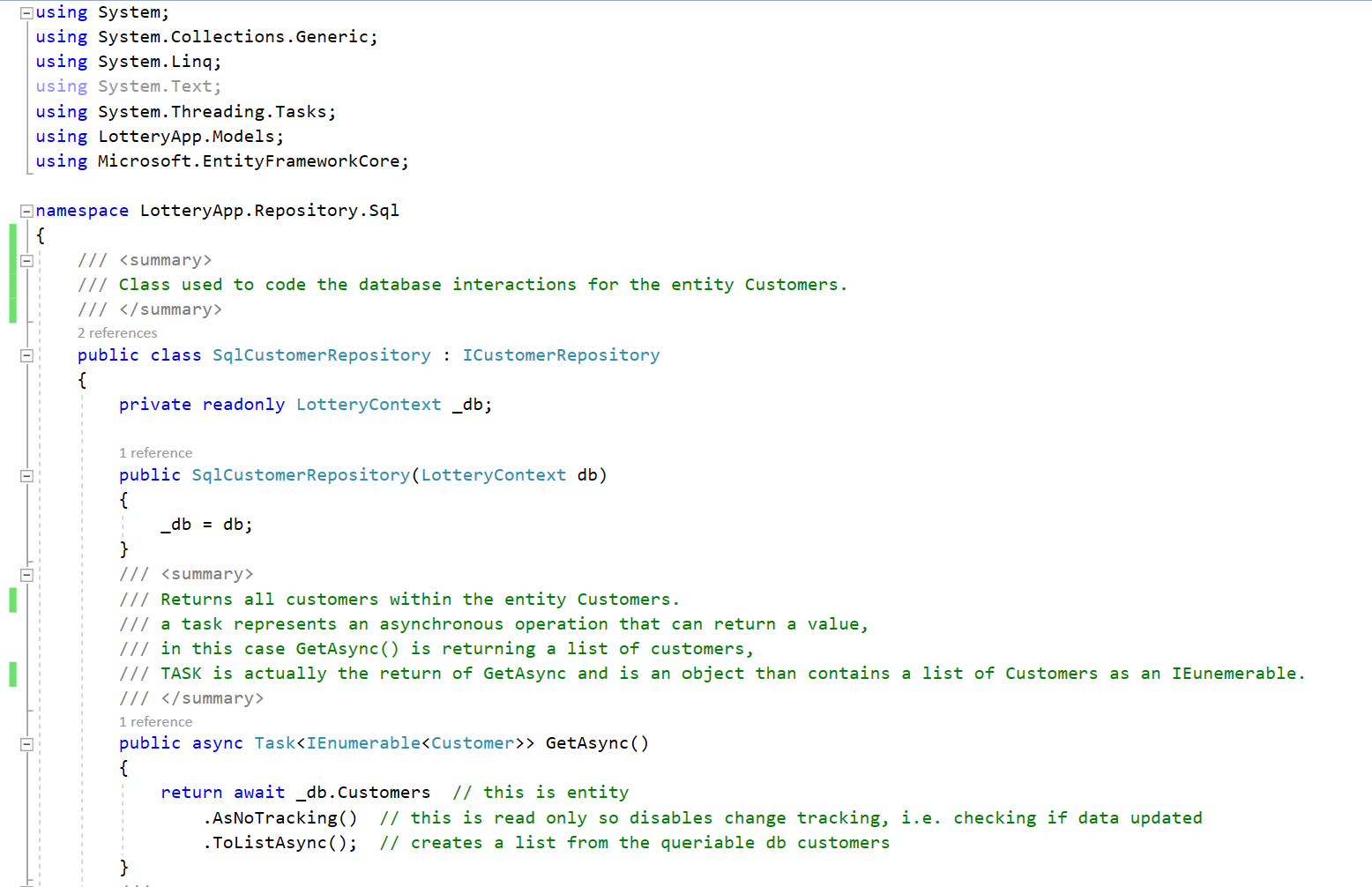
/// </summary>

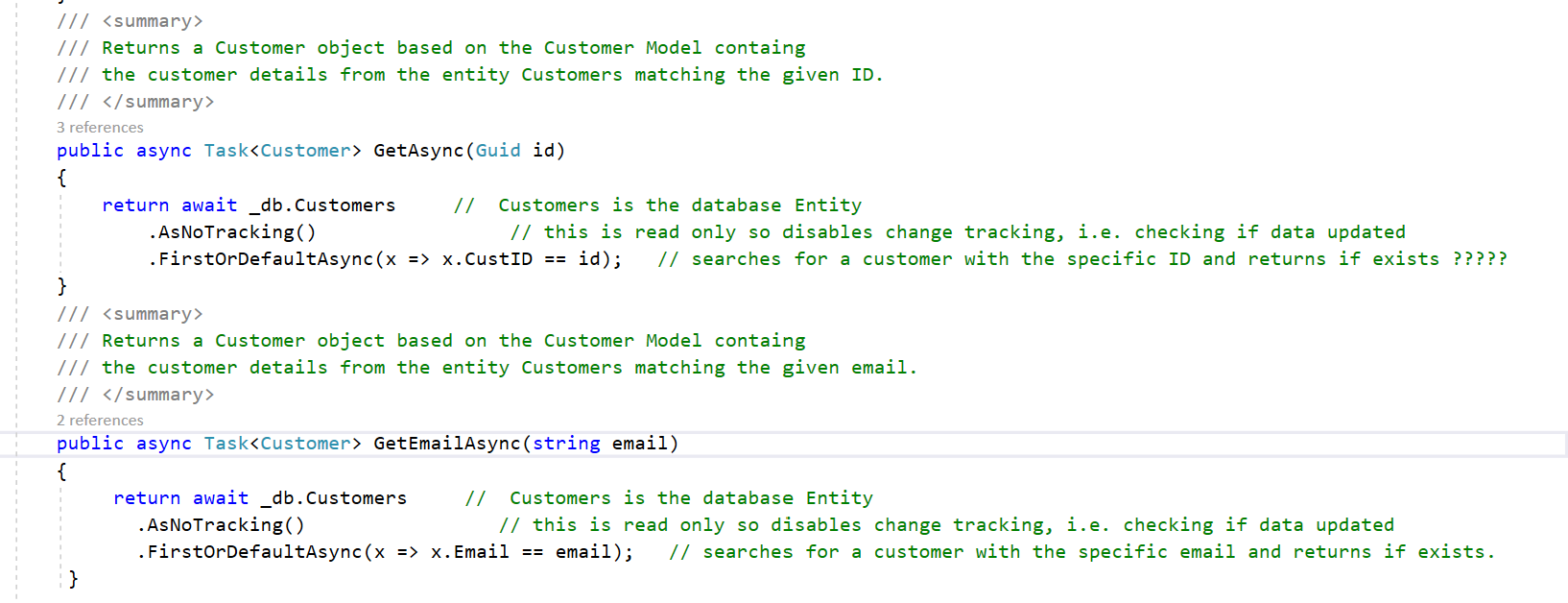
Task<Customer> UpsertAsync(Customer customer);

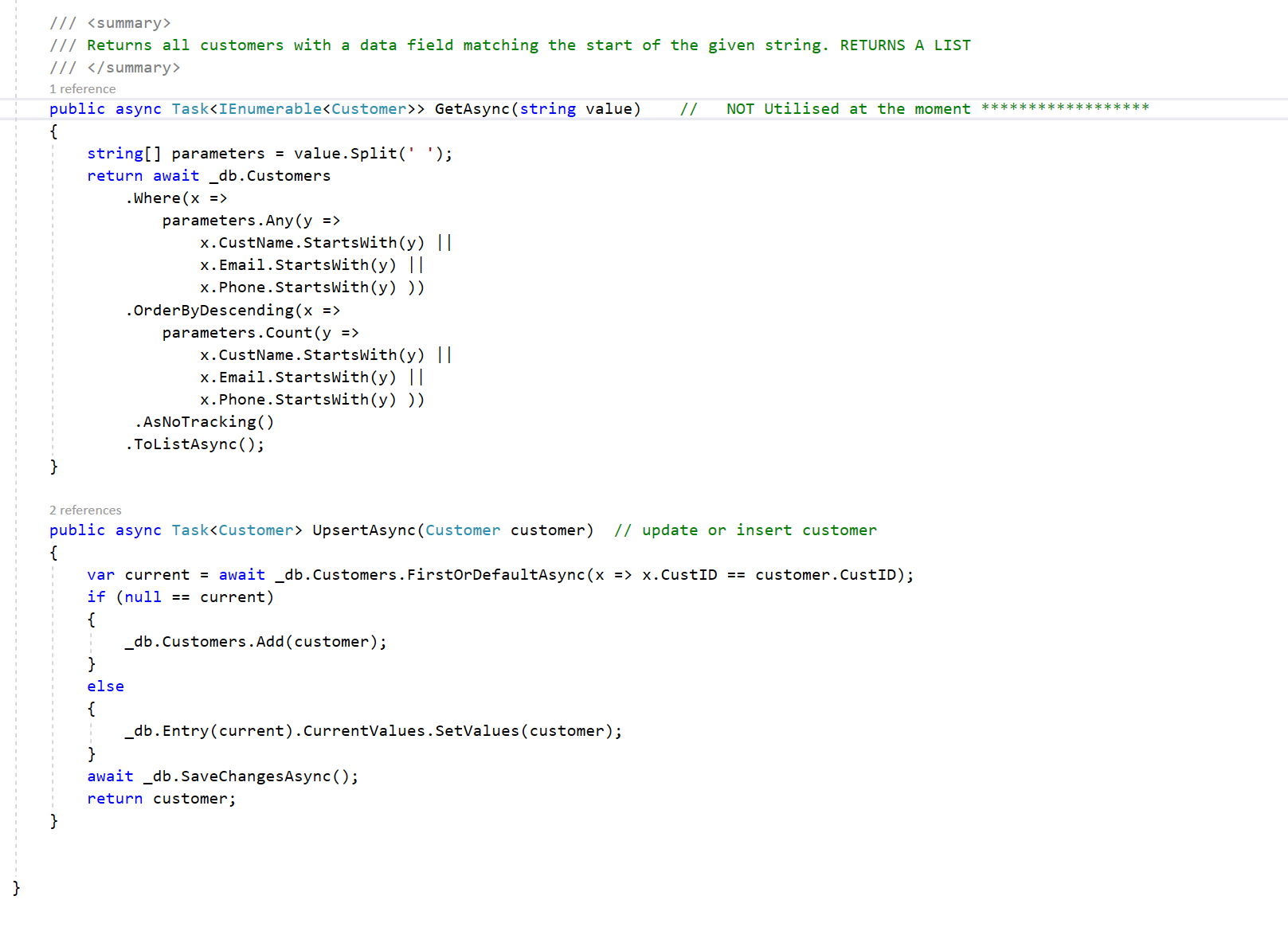
}

}

SqlCustomer Repository implements the ICustomerRepository Code with the code specific to the customer data and requirements for access to the backend database.







Note the constructor sets up the links to the database session created in App.xaml.cs, each of the Tasks codes the connects to the dbContext(database connection) and within that the dbSet (entity).

Note: Lambada and LINQ expressions are used to query the database see return customer with a given ID.

In the Entity Framework based applications, the DbContext / Object Context is responsible for tracking the changes done in the objects, so the correct update is done to the database when the SaveChanges() method of the context is called. When we retrieve entities using an object query, the Entity Framework puts these entities in a cache and tracks whatever changes are made on these entities until the savechanges method is called.  
  
Sometimes we do not want to track some entities because the data is only used for viewing purposes and other operations such as insert, update and delete are not done. For example the view data in a read-only grid.  
  
The **AsNoTracking()** extension method returns a new query and the returned entities will not be cached by the context (DbContext or Object Context). This means that the Entity Framework does not perform any additional processing or storage of the entities that are returned by the query. Please note that we cannot update these entities without attaching to the context.

using System;

using System.Collections.Generic;

using System.Linq;

using System.Text;

using System.Threading.Tasks;

using LotteryApp.Models;

using Microsoft.EntityFrameworkCore;

namespace LotteryApp.Repository.Sql

{

/// <summary>

/// Class used to code the database interactions for the entity Customers.

/// </summary>

public class SqlCustomerRepository : ICustomerRepository

{

private readonly LotteryContext \_db;

public SqlCustomerRepository(LotteryContext db)

{

\_db = db;

}

/// <summary>

/// Returns all customers within the entity Customers.

/// a task represents an asynchronous operation that can return a value,

/// in this case GetAsync() is returning a list of customers,

/// TASK is actually the return of GetAsync and is an object than contains a list of Customers as an IEunemerable.

/// </summary>

public async Task<IEnumerable<Customer>> GetAsync()

{

return await \_db.Customers // this is entity

.AsNoTracking() // this is read only so disables change tracking, i.e. checking if data updated

.ToListAsync(); // creates a list from the queriable db customers

}

/// <summary>

/// Returns a Customer object based on the Customer Model containg

/// the customer details from the entity Customers matching the given ID.

/// </summary>

public async Task<Customer> GetAsync(Guid id)

{

return await \_db.Customers // Customers is the database Entity

.AsNoTracking() // this is read only so disables change tracking, i.e. checking if data updated

.FirstOrDefaultAsync(x => x.CustID == id); // searches for a customer with the specific ID and returns if exists ?????

}

/// <summary>

/// Returns a Customer object based on the Customer Model containg

/// the customer details from the entity Customers matching the given email.

/// </summary>

public async Task<Customer> GetEmailAsync(string email)

{

return await \_db.Customers // Customers is the database Entity

.AsNoTracking() // this is read only so disables change tracking, i.e. checking if data updated

.FirstOrDefaultAsync(x => x.Email == email); // searches for a customer with the specific email and returns if exists.

}

/// <summary>

/// Returns all customers with a data field matching the start of the given string. RETURNS A LIST

/// </summary>

public async Task<IEnumerable<Customer>> GetAsync(string value) // NOT Utilised at the moment \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

{

string[] parameters = value.Split(' ');

return await \_db.Customers

.Where(x =>

parameters.Any(y =>

x.CustName.StartsWith(y) ||

x.Email.StartsWith(y) ||

x.Phone.StartsWith(y) ))

.OrderByDescending(x =>

parameters.Count(y =>

x.CustName.StartsWith(y) ||

x.Email.StartsWith(y) ||

x.Phone.StartsWith(y) ))

.AsNoTracking()

.ToListAsync();

}

public async Task<Customer> UpsertAsync(Customer customer) // update or insert customer

{

var current = await \_db.Customers.FirstOrDefaultAsync(x => x.CustID == customer.CustID);

if (null == current)

{

\_db.Customers.Add(customer);

}

else

{

\_db.Entry(current).CurrentValues.SetValues(customer);

}

await \_db.SaveChangesAsync();

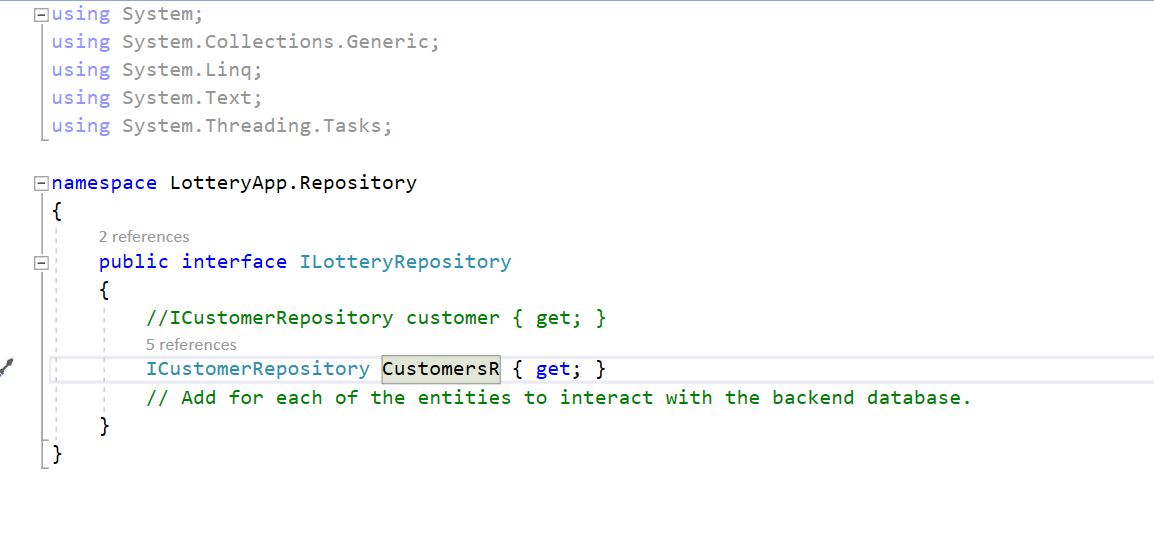
return customer;

}

}

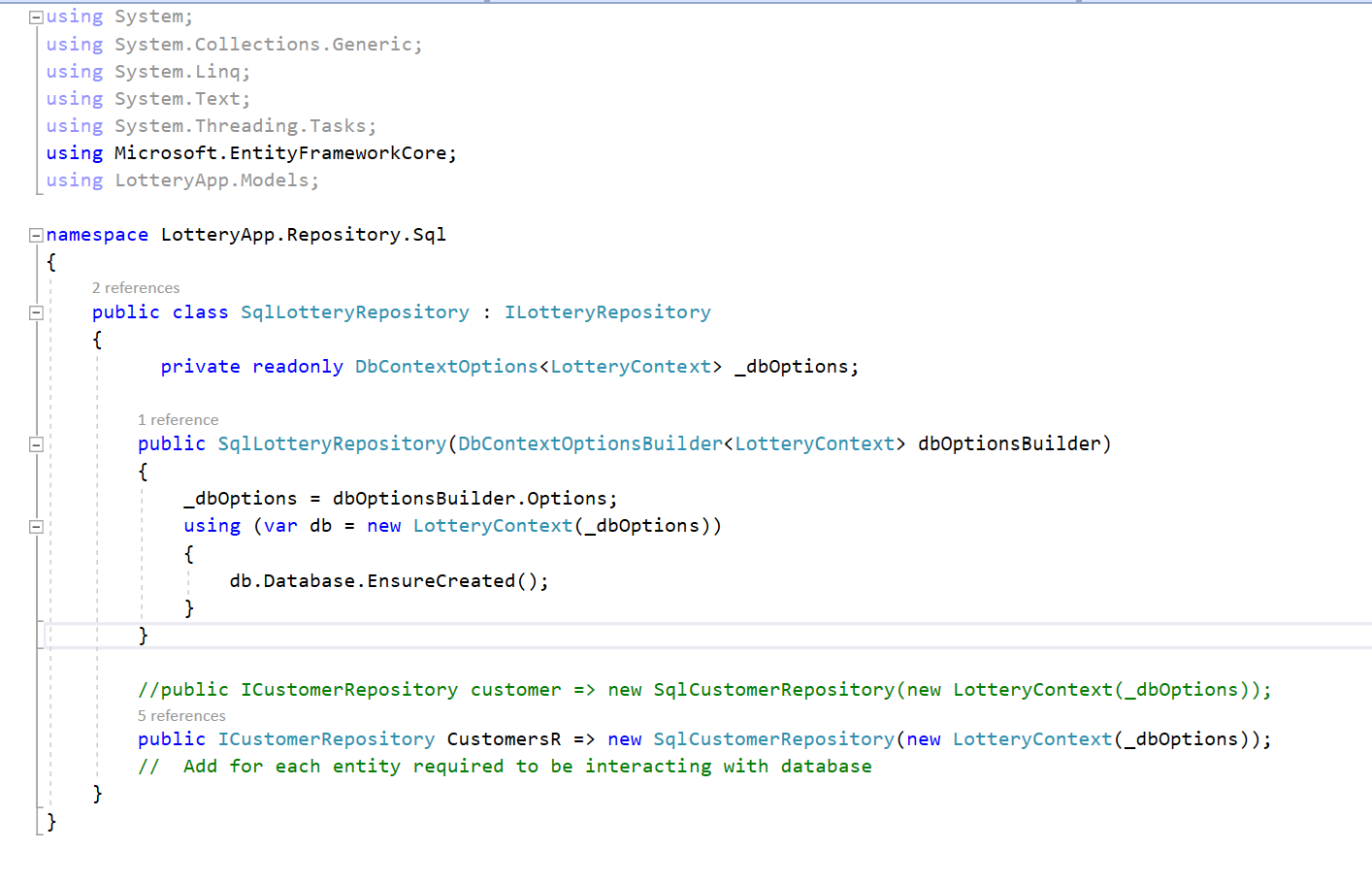
}

ILotteryRepository.cs, defines methods for interacting with the back end database, these methods are implemented in SqlCustomerRepository.cs



The implementation of this in SqlLotteryRepository.cs is as follows:

Configures database connections for the context, creates the instance of the SqlCustomerRepository which must implement from ICustomerRepository, this is the SqlCustomerRepository has the commands and LINQ queries to interact with backend database within confines of model.



**How do LotteryContext, SqlLotteryRepository and SqlCustomerRepository (or any other SqlEntityRepository) work together.**

LotteryContext implements DbContext and has properties of type DbSets.

It is the class file which has definitions for defining an instance of the DbContext (connection to database) and DbSets used to query and save changes to each entity for the overall application. It is sent options for the database connection from the application at startup through SqlLotteryRepository.

SqlLotteryRepository (creates instances of LotteryContext and each SqlEntityRepository ) –

In the constructor uses the prebuilt framework DbContextOptionsBuilder<LotteryContext> to create an instance of the LotteryContext named \_dbOptions, this constructor is sent the database options (from the APP code) to allow for the connection to the required database.

It also creates as a property an instance of each of the entity repositories sending an instance of the LotteryContext , therefore passing the connection and the dbSets within LotteryContext.

It also creates as a property an instance of each of the SqlEntityRepository, sending the dbContext connection and dbSet connection for that entity to allow for the asynchronous operations coded within each SqlEntityRepository.

i.e. properties for each EntityRepository coded as follows  
 ICustomerRepository Customers => new SqlCustomerRepository(new LotteryContext(\_dbOptions)

SqlCustomerRepository (as all SqlEntityRepositories)

Instances of these created in the SqlLotteryRepository (App repository), within the constructor sets the database connection for any instance. Also codes the Tasks for any instance of that entity for whatever operations for that entitiy may be required (i.e. CRUD).

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

Models are created and so these must now interact with the views to allow for the user input and interactions.

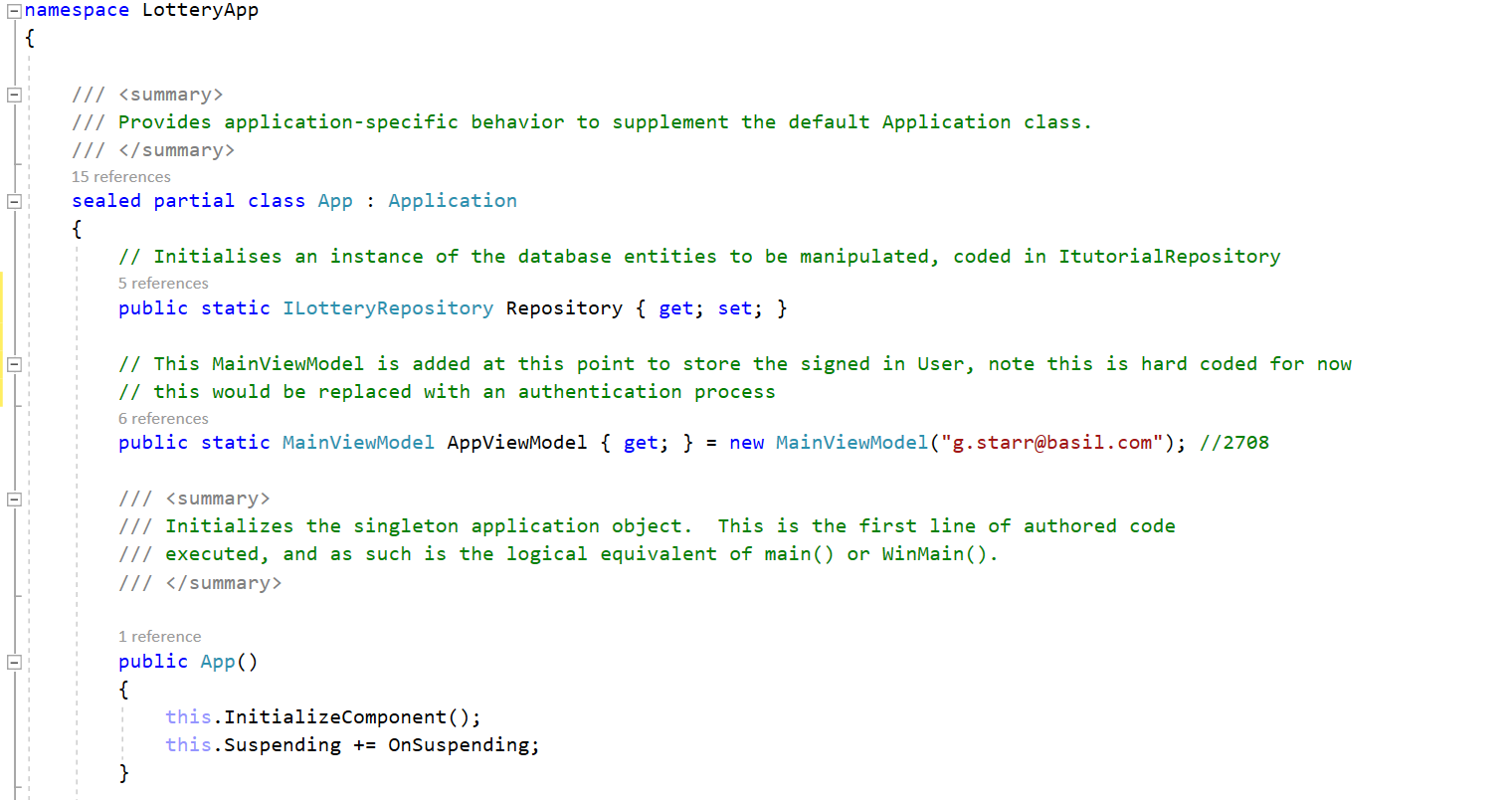
App Start

The starting point for the App is App.xaml.cs:

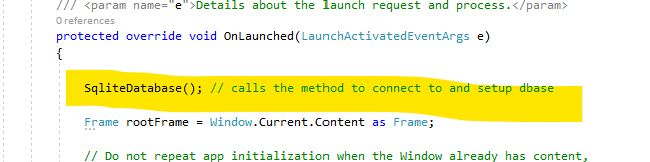
The repository object is set up on launch of the application. The repository creates the Dbsets for the customers to be manipulated. After this the App will launch the first page of the application which is MainPage.xaml.

App.xaml.cs

Initialises an instance of the ILotteryRepository coded in the repository folder, this repository is available throughout the app. The sqlLotteryRepository codes/sets up all of the methods which will access the database I.e. find a Customer entity with a particular ID. So the App.xaml.cs, creates the Repository with the connection to the database (for now this is an sqlite database). Also creates an instance of the MainViewModel which will take place of an authentication logon for now.

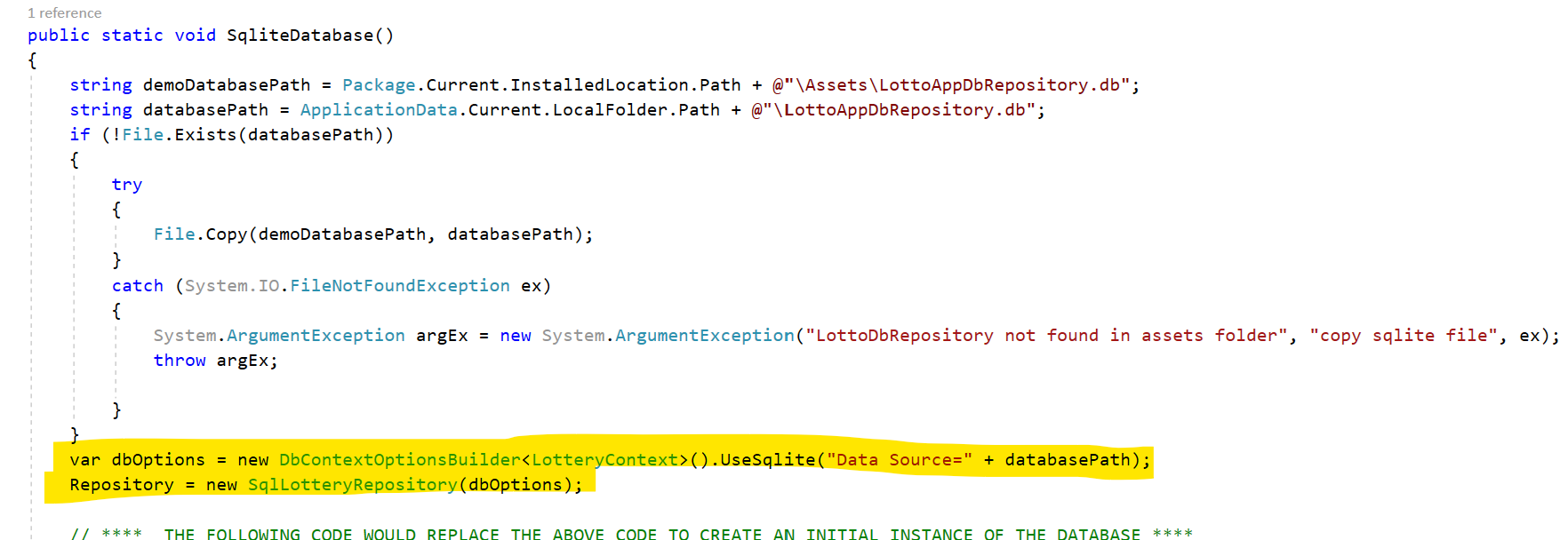


Within the OnLaunhed event for the app within App.xaml.cs the SqliteDatabase() method is called which instansiates the Repository object with the options for the SqliteDatabase connection. Note: This tutorial shows a SqliteDatabase connection for example purposes and easiness of running on any platform for now. The data for Lottery Tickets would require to be hosted on a central server through another SQL manager with perhaps a local storage option for the app.



This assumes there is already a demo SqliteDatabase as part of the installed package, please see the separate tutorial on how to set this up. This part would change depending on the database manager to be used.

The Repository is created as an instance of SqlTutorialRepository and sends the Sqlite Options to the data context



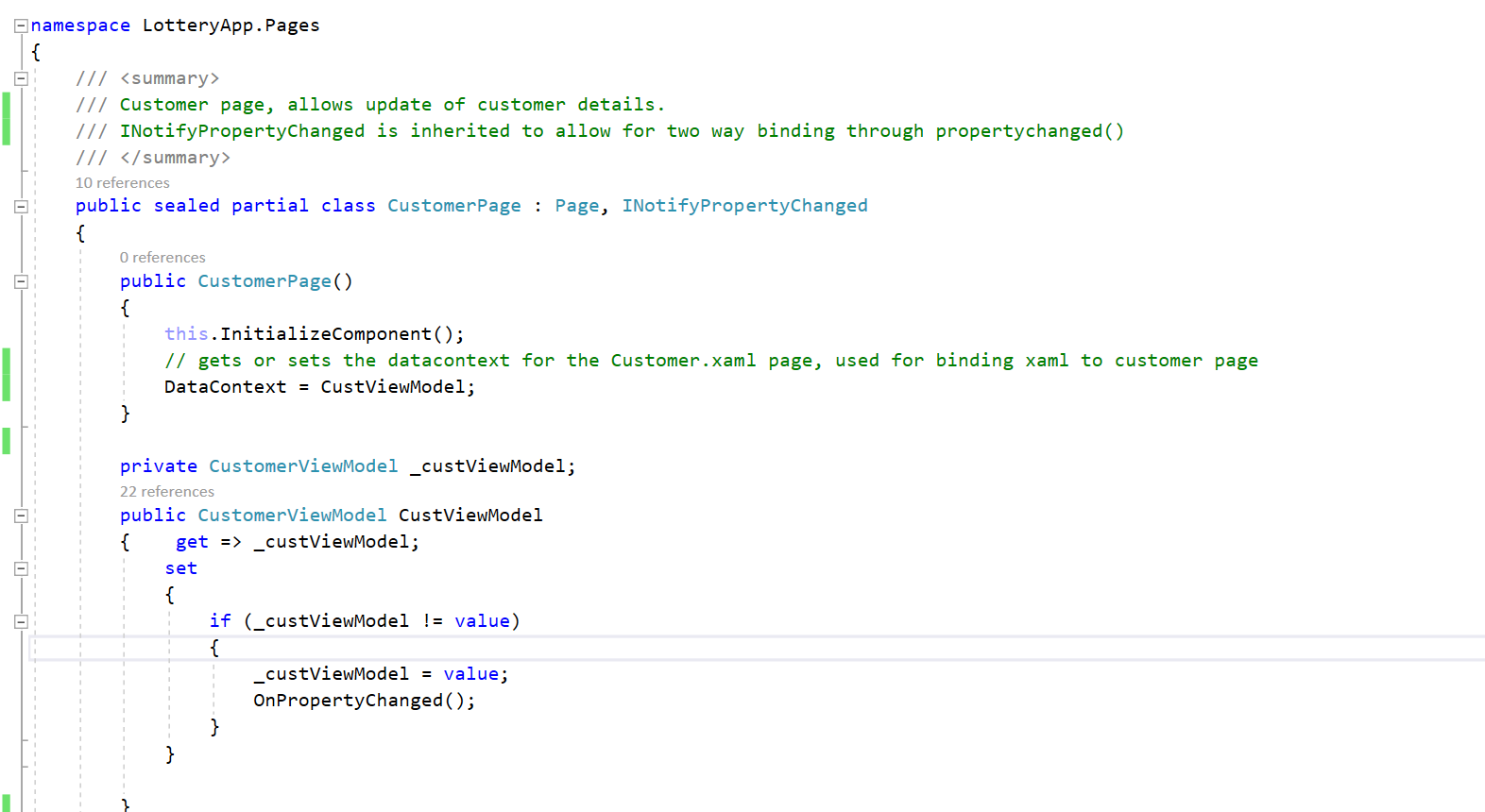
So assuming the Sqlite database is within the correct folders this would set up the database connection for use throughout the application.

Customer.xaml, Customer.xaml.cs, CustomerViewModel.cs

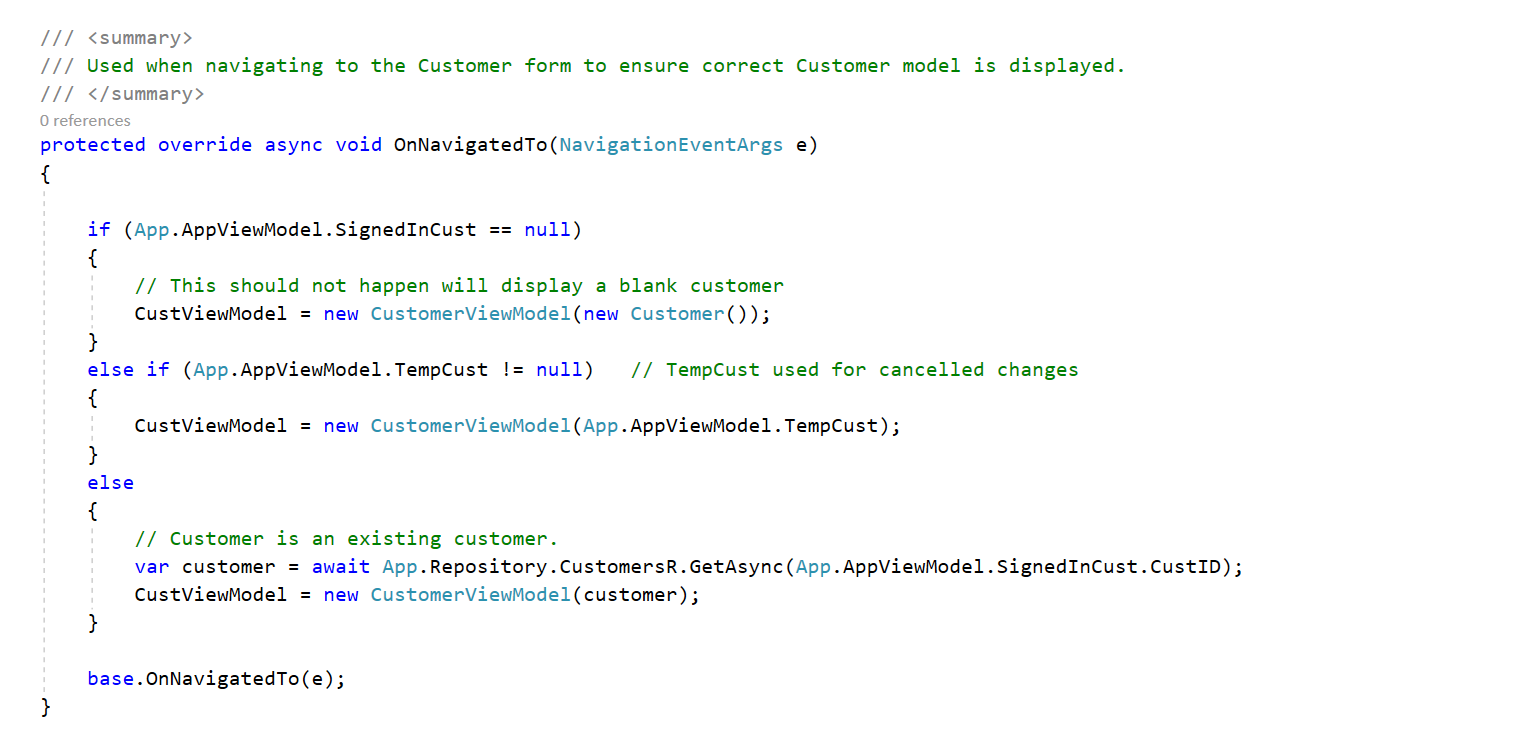
ON navigation to the Customer page the signed in customer would be retrieved from the database on search of the email address and the customer details would be displayed on screen. The user can then amend the details if they wish.

To do this in the MVVC architecture a View is set up Customer.xaml with text boxes for the customer fields which are bound to the properties within the instance of the ViewModel CustomerViewModel.cs (this is the DataContext). The ViewModel calls the code within the sqlCustomerRepository to retrieve and set any data as required. The Customer.xaml sets the binding for the textboxes and formats the page. The Customer.xaml.cs controls how it is all run, sets up the ViewModel, controls the navigation on exit of any transaction.

**Customer.xaml.cs**The first part sets the DataContext to the property CustViewModel (note CustViewModel is a property of the this page).



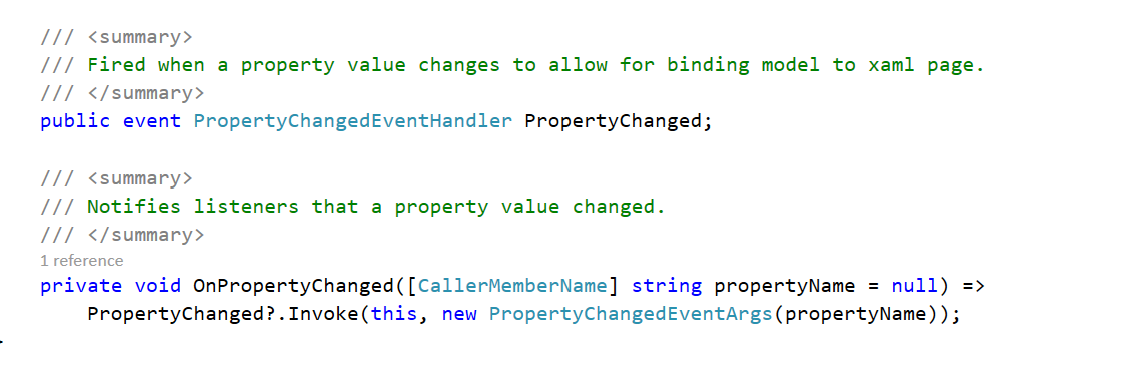
OnNavigatedTo is overridden from the main page to set up the customerViewModel with the instance of the correct CustomerModel. Note TempCust is a property within the MainViewModel to used to store temporary changes to the customer if a save is not made on exit of the customer page.



This is the code to display a dialogue box on exit of the screen to check the user action if the Customer page is exited with changes not saved or aborted.



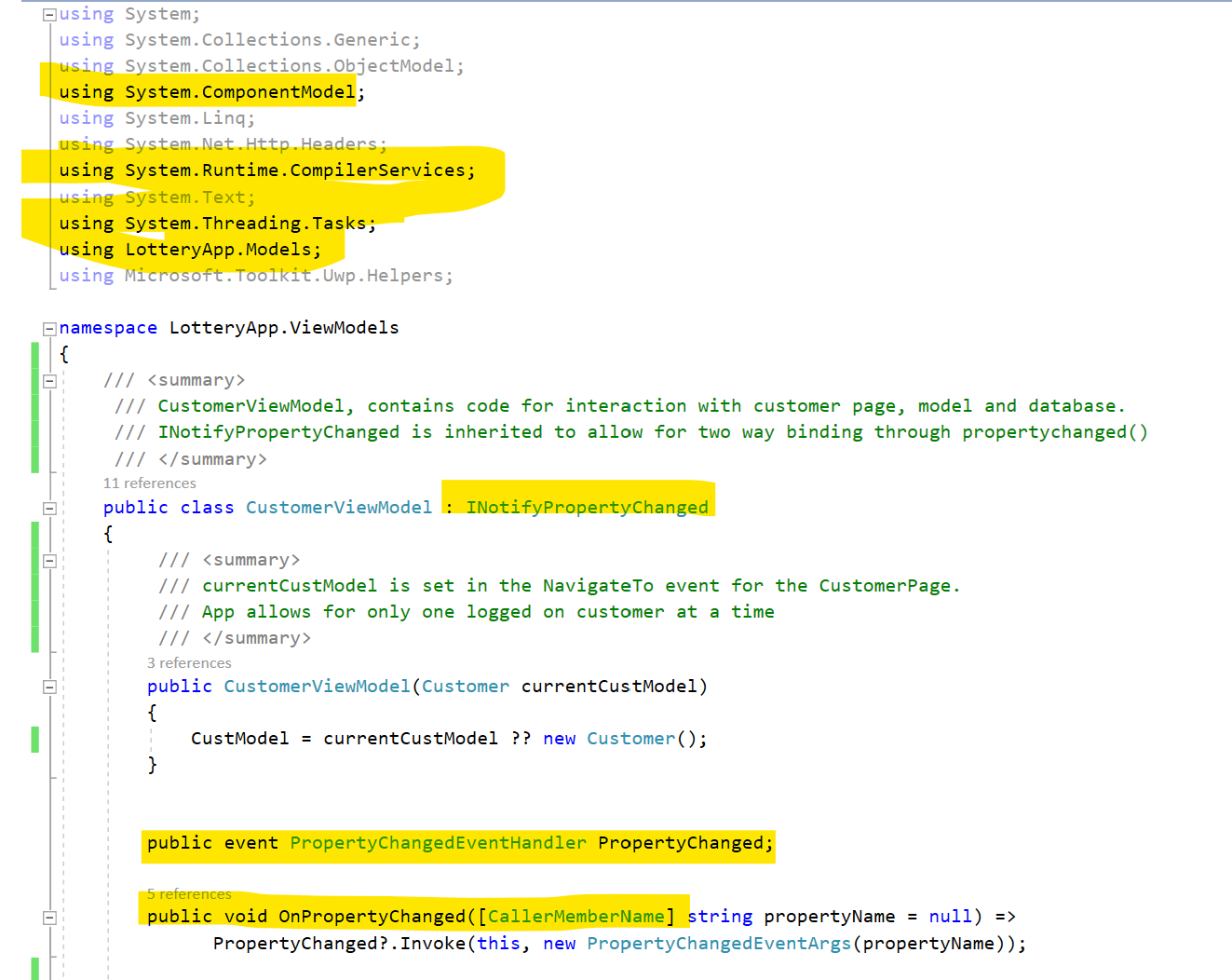
This ensures the two way binding between the ViewModel and the customer.xaml takes place.

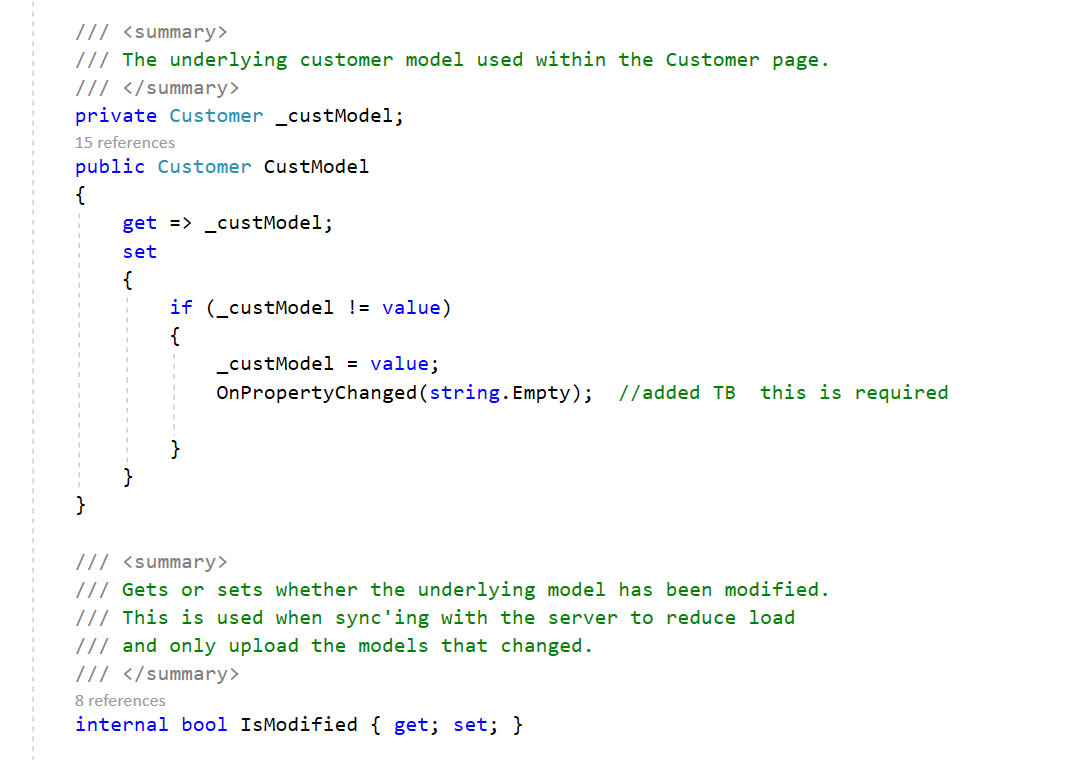


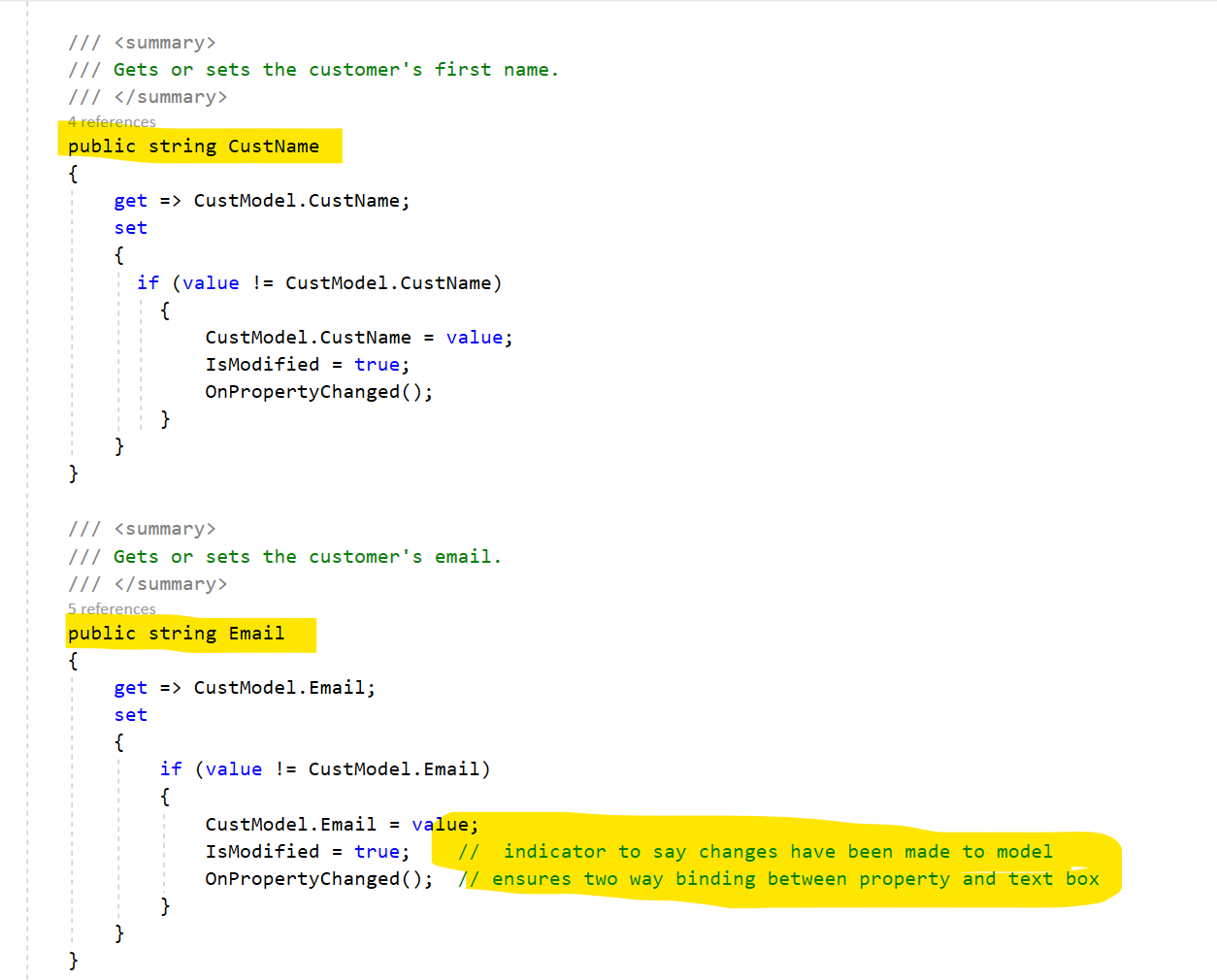
NOTE: if the save cancel dialogue box is initiated the code runs the SaveChangesDialogue.xaml (displayed below).

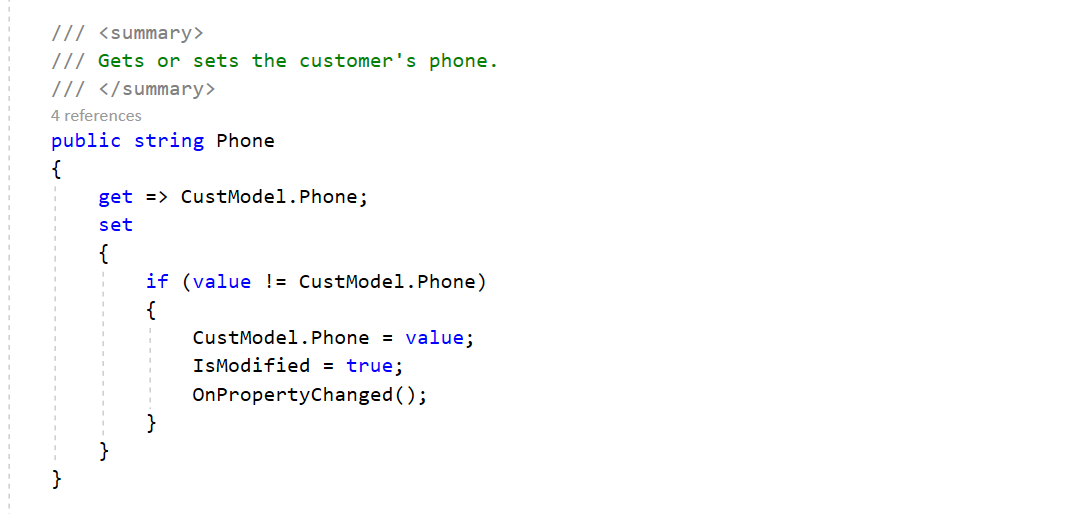
CustomerViewModel.cs

Note, this implements INotifyPropertyChanged to notify automatically if one of the textboxes in Customer.xaml which is bound to the Name, Email or Phone property is updated.

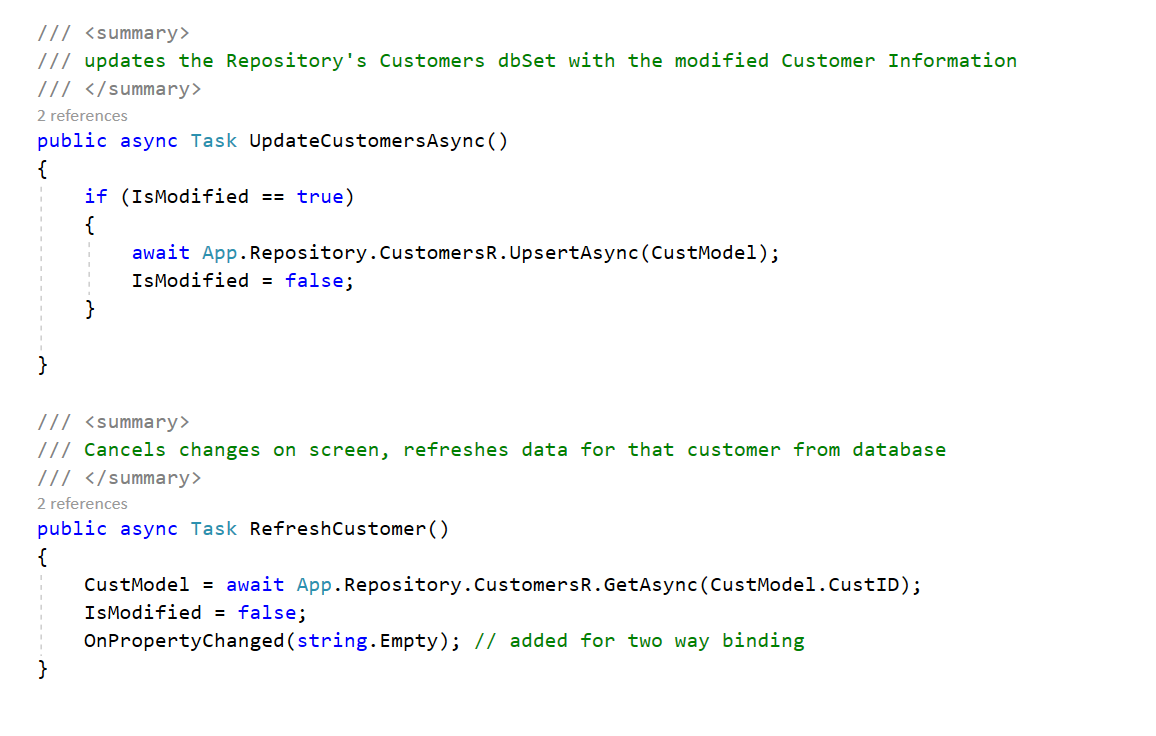








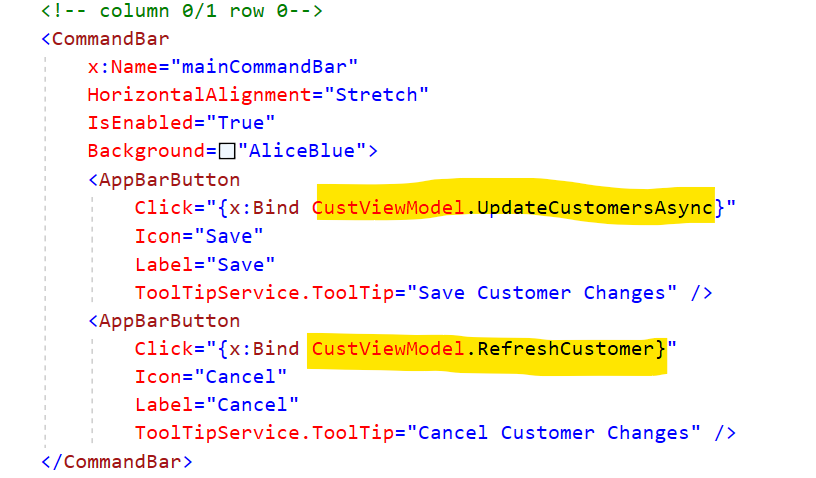
Tasks which are bound to the buttons for save, delete and allow for the repository to be updated from the Model.



In Customer.xaml

We have already added the datacontext in customer.xaml.cs

Bind the command buttons in the app bar to the methods in the CutomerViewModel, note some of these are unnecessary but are helpful if you wanted to implement later.

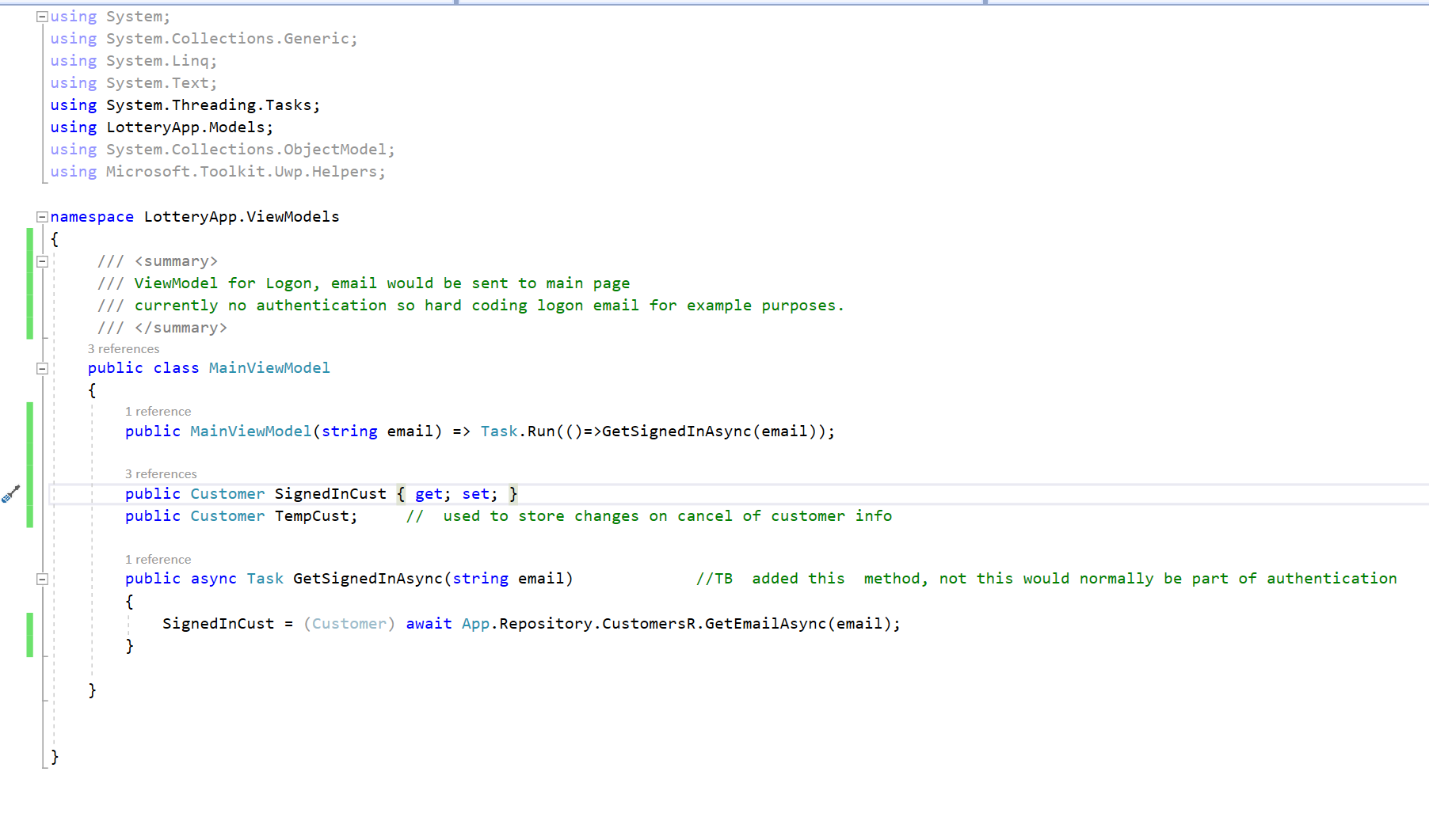


Now add the binding on the text boxes to the properties in the CustomerViewModel, note the PropertyChanged which triggers the changes being adopted by the model, this is from the INotifyPropertyChanged interface implemented within CustomerViewModel. (Note: validation for phone number was commented out)



MainViewModel.cs (In ViewModels):

Used to store logged in customer and Temporary changes to customer page.



SaveChangesDialog.xaml (In Views):



