CSC320-A

Module 2

Implementing Asynchronous Operations & Cryptography

1 Models & Controllers

Even though each application provide different functionality, it is simple to develop all applications if there is a standard application architecture that all applications can be applied to. The architecture will determine the steps that a developer can follow to be able to implement all applications regardless of size and complexity. In this module, we will implement the models and controllers portion of Model-View-Controller (MVC) application architecture.

1.1 Implementing a Data Model

A model represent one or more data-oriented classes that you can use to store data that commonly represents an unique entity that can then be edited or displayed in views and processed by controllers. There should be some persistence features that to save and restore data models from persistent storage such as files and databases. We have already demonstrated previously how to persist a data model to an XML file. In this chapter, we will demonstrate data models persisted in a database.

Example of a model class representing a single entity

```
public class Account {
    public int Id { get; set; }
    public string Name { get; set; }
    public decimal Balance { get; set; }
}
```

Example of a model class representing a set of entities

```
public class AccountList : List<Account> {
    public void Save(string path) { ... }
    public static AccountList Load(string path) { ... }
}
```

If you wish to persist data models to a database, there is no need to actually create the data model in code as since .NET version 3.5. Microsoft has added data access and model generation technologies like *ADO.NET Entity Data Model* and *LINQ to SQL Classes*. In this chapter, you will use LINQ to SQL Classes to generate a data model to store and access data in a SQL Server database.

We will first create the database for and generate a data model for *SymBank.Banking* module. Run *SQL Server Management Studio to* execute *SymBank.sql* script provided to create a database that has two tables **Accounts** and **Transactions** and a set of stored procedures. Once the database is ready, you can add a *Data Connection* to the database using the *Server Explorer* window in *Visual Studio*.

Data Connection properties

```
Server Name     :   .
Authentication  :   Windows Authentication
Database        :   SymBank
```

Add a **Models** folder to **SymBank.Banking** project. To generate a data model, first add a *LINQ to SQL Classes* template named **SymBank.dbml** in the folder. Then drag and drop all tables in *SymBank* database to the left pane and drag and drop all the stored procedures on the right pane. LINQ to SQL will generate one class for each table in your database to store individual entities and also generate an additional data context class to store and access the entities in the database as well as for calling the stored procedures. You can now compile the project to compile the data model.

While the data model does provide all the functionality that we need in order to store and access data in the database, you should not perform data operations directly in your application as each data operation may be still a bit complex or tedious to perform. You should implement controllers to help you perform the data operations on the data model.

1.2 Implementing a Controller

Controllers provide services that views utilize to perform tedious or complex tasks. Even though it is not compulsory that controllers require interfaces, you do get extra functionality and also more security if you abstract access to controllers. We will first declare interfaces that represent all the functionality that each controller can provide to the application. Add **Services** folder to declare the following interfaces and add a **Controllers** folder to create a class to implement the interfaces.

Interface for account services : Services\IAccountController.cs

```
public interface IAccountController : IService {
    void Add(Account item);
    Account GetAccount(int code);
    List<Account> GetAccountList();
    List<Account> GetAccountsForName(string name);
}
```

Interface for transaction services: Services\ITransactionController.cs

```
public interface ITransactionController : IService {
    int Debit(int source, decimal amount);
    int Credit(int source, decimal amount);
    int Transfer(int source, int target, decimal amount);
}
```

Implementing the controller: Controllers\BankingController.cs

```
public class BankingController : BaseService,
    IAccountController, ITransactionController {
            :
}
```

When performing transactions, the controller needs to record the user name. This can be retrieved using the **IAuthorization** service which we can obtain by using property injection. The service can also be use to perform authorization security checks before performing data operations. Declare a property for the service marked with **Inject** attribute. We can call the **Inject** extension method in **ServiceRepository** from the constructor. Of course you can manually fetch the service without using dependency injection.

Using dependency injection

```
[Inject]public IAuthorization Authorization { get; set; }

public BankingController() { this.Inject(); }
```

Manually retrieving the service

```
public IAuthorization Authorization { get; set; }

public BankingController() {
    Authorization = ServiceRepository.Get<IAuthorization>();
}
```

To use the data model, create an instance of a data-context class generated by LINQ to SQL. It has one property for each table; Accounts and Transactions. Each table implements IQueryable<T> and IEnumerable<T> which means that you can apply all available LINQ operators and **foreach** to access entities in the table. The data-context exposes stored procedures as methods that you can call to execute the procedures in the database.

Implementing IAccountController contract

```
public void Add(Account item) {
        var dc = new SymBankDataContext();
        dc.AccountAdd(item.Code,
            item.Type, item.Name,
            item.ZipCode, Authorization.UserName,
            DateTime.Now, item.Balance);
}

public Account GetAccount(int code) {
        var dc = new SymBankDataContext();
        return dc.Accounts.SingleOrDefault(a => a.Code == code);
}

public List<Account> GetAccountList() {
        var dc = new SymBankDataContext();
        return dc.Accounts.ToList();
}

public List<Account> GetAccountsForName(string name) {
    name = name.ToLower();
    var dc = new SymBankDataContext();
    var query = from account in dc.Accounts
        where account.Name.ToLower().Contains(name)
        orderby account.Name
        select account;
    return query.ToList();
}
```

Implementing ITransactionController contract

```
public int Debit(int source, decimal amount) {
    int? transactionCode = null;
    var dc = new SymBankDataContext();
    dc.AccountDebit(source, amount, Authorization.UserName,
        DateTime.Now, ref transactionCode);
    return (int)transactionCode;
}

public int Credit(int source, decimal amount) {
    int? transactionCode = null;
    var dc = new SymBankDataContext();
    dc.AccountCredit(source, amount, Authorization.UserName,
        DateTime.Now, ref transactionCode);
    return (int)transactionCode;
}

public int Transfer(int source, int target, decimal amount) {
        int? transactionCode = null;
        var dc = new SymBankDataContext();
        dc.AccountTransfer(source, target, amount, Authorization.UserName,
            DateTime.Now, ref transactionCode);
        return (int)transactionCode;
}
```

The controller is now completed. You can use the controller in any kind of application to easily store and access data from the *SymBank* database. In the **BankingModule** class, we can pre-create the controller and register it with our **ServiceRepository** so that multiple views can access and use it. Alternatively you can use **ServiceAttribute** to mark the controller to allow **ServiceRepository** to help you create and register it.

Preparing controllers in code: BankingModule.cs

```
public override void Init() {
    var controller = new BankingController();
    controller.Add<IAccountController>();
    controller.Add<ITransactionController>();
                :
}
```

Automation with ServiceAttribute: BankingController.cs

```
[Service(typeof(IAccountController))]
[Service(typeof(ITransactionController))]
public class BankingController : BaseService,
    IAccountController, ITransactionController {
                :
}
```

2 Task Parallel Library

2.1 Data Parallelism

Multi-threading does not really improve performance if you only have one processor. It just allows you to fully utilize the processor. When a thread is waiting for something it is not using CPU time, than that time can be utilized by another thread. However, if you do have multiple processors performance can be improved by distributing threads across all the processors. One thread can only run on one processor so you definitely need to create multiple threads to fully utilize all available processors. However it has always been very difficult to program a single operation to use multiple threads but in .NET 4.0, Microsoft has implement the Task Parallel Library (TPL) to simplify this. The following is a method that runs completely on one thread.

Operation that runs on one thread: Parallel1\Program.cs

```
static List<string> list1 = new List<string>();
static int ScanDirectory1(string path) {
    try { var files = Directory.GetFiles(path);
        foreach (string file in files) list1.Add(file);
        var directories = Directory.GetDirectories(path);
        int count = files.Length;
        foreach (var directory in directories)
            count += ScanDirectory1(directory);
        return count;
    } catch (Exception) { return 0; }
}
```

The above method scans a directory and all sub-directories on one thread to collect a list of files and return the number of files collected. It would be faster if we can use a separate thread to scan through each sub-directory and these threads is distributed across multiple processors. This can be easily done using TPL by replacing a **foreach** with **Parallel.ForEach** method instead as shown below.

Operation that runs across multiple threads: Parallel1\Program.cs

```
static List<string> list2 = new List<string>();
static int ScanDirectory2(string path) {
    try {   var files = Directory.GetFiles(path);
        foreach (string file in files) list2.Add(file);
        var directories = Directory.GetDirectories(path);
        int count = files.Length;
        Parallel.ForEach(directories, directory =>
            count += ScanDirectory2(directory));
        return count;
    }   catch (Exception) { return 0; }
}
```

We will now measure the time taken to run both methods using **Stopwatch**. The time is consistent for the first method regardless of how many processors you have while the second method will scale automatically to the number of available processors.

Timing synchronous and asynchronous operations

```
static void Main() {
    var watch1 = Stopwatch.StartNew();
    int count1 = ScanDirectory1("C:\\Windows");
    watch1.Stop();
    var watch2 = Stopwatch.StartNew();
    int count2 = ScanDirectory2("C:\\Windows");
    watch2.Stop();
    Console.WriteLine(watch1.ElapsedMilliseconds);
    Console.WriteLine(watch2.ElapsedMilliseconds);
    Console.WriteLine(count1);
    Console.WriteLine(count2);
    Console.WriteLine(list1.Count);
    Console.WriteLine(list2.Count);
}
```

2.2 Concurrency

While the parallel operation is definitely faster, the results are actually incorrect. This is due to concurrency problems that occur when we try to update the same collection or field from multiple threads. All of the collection types in **System.Collections** and **System.Collections.Generic** are not thread-safe. Adding items into the collection by multiple threads at the same time can corrupt the internal structure of the collection. Results can range from inaccurary of items added or even crashing the program. You can synchronize non thread-safe operations by using **lock**.

Synchronize updating of a shared collection

```
foreach (string file in files) lock(list2) list2.Add(file);
```

Alternatively you use a thread-safe collection from **System.Collections.Concurrent**. Methods in these collections will perform internal synchronization. They will be slower but the results will be accurate.

Replace List<T> with ConcurrentBag<T>

```
static ConcurrentBag<string> list2 = new ConcurrentBag<string>();
```

While the collection is now accurate, the file **count** is not. You can use **lock** keyword to synchronize updating of the counter. However since updating counters is common in multi-threading operations, you can can use a special **Interlocked** class to update counters instead.

Synchronize counter updates with Interlocked class

```
Parallel.ForEach(directories, directory =>
    Interlocked.Add(ref count, ScanDirectory2(directory)));
```

2.3 Code Parallelism

While **Parallel.ForEach** is a asynchronous version of **foreach**, you can also run other statements asychronously by using **Parallel.Invoke**. It can accept a parameter array of **Action** delegates to run code. Multiple threads will be used to run the delegates on multiple processors.

Performing operations synchronously: Parallel2\Program.cs

```
static void Main() {
    Console.WriteLine("Task 1"); Thread.Sleep(4000);
    Console.WriteLine("Task 2"); Thread.Sleep(4000);
    Console.WriteLine("Task 3"); Thread.Sleep(4000);
    Console.WriteLine("Finish");
}
```

Performing operations asynchronously

```
Parallel.Invoke(
    () => { Console.WriteLine("Task 1"); Thread.Sleep(4000); },
    () => { Console.WriteLine("Task 2"); Thread.Sleep(4000); },
    () => { Console.WriteLine("Task 3"); Thread.Sleep(4000); }
);
Console.WriteLine("Finish");
```

Since each parameter of the parameter array is just an **Action** delegate, any complex tasks can implemented as separate methods and pass as individual parameters or as an array.

Task code implemented in separate methods

```
static void Task1() { Console.WriteLine("Task 1"); Thread.Sleep(4000); }
static void Task2() { Console.WriteLine("Task 2"); Thread.Sleep(4000); }
static void Task3() { Console.WriteLine("Task 3"); Thread.Sleep(4000); }
```

Pass as individual parameters

```
Parallel.Invoke(Task1, Task2, Task3);
```

Pass in an array

```
Action[] tasks = { Task1, Task2, Task3 };
Parallel.Invoke(tasks);
```

2.4 Creating Tasks

Parallel class create **Task** objects to allocate threads to run code. You can also use this class directly. You can call **Start** to begin running each task and call **Wait** to wait for the task to be completed. There is also a generic version of **Task** that allows execution of tasks that has a result.

Creating and using Task objects: Tasks1

```
Task task1 = new Task(() => { Console.WriteLine("Task 1"); Thread.Sleep(4000); });
Task task2 = new Task(() => { Console.WriteLine("Task 2"); Thread.Sleep(4000); });
Task task3 = new Task(() => { Console.WriteLine("Task 3"); Thread.Sleep(4000); });
task1.Start(); task2.Start(); task3.Start();
task1.Wait(); task2.Wait(); task3.Wait();
Console.WriteLine("Finish");
```

Additional Task features

```
Task.WaitAll(task1,task2,task3);                // wait for multiple tasks to complete
Task.WaitAny(task1,task2,task3);                // wait for one of the tasks to complete
Task tasks1 = Task.WhenAll(task1,task2,task3);  // combining WaitAll tasks into one
Task tasks2 = Task.WhenAny(task1,task2,task3);  // combining WaitAny tasks into one
tasks1.Wait();                                  // wait for multiple tasks to complete
tasks2.Wait();                                  // wait for one of the tasks to complete
```

Returning results from a Task

```
Task<int> task4 = new Task<int>(() => {
    Console.WriteLine("Task 4"); Thread.Sleep(4000); return 123; });
task4.Start(); task4.Wait();
Console.WriteLine(task4.Result);
```

2.5 Task Cancellation

Aborting a thread can be dangerous as you have no idea what the thread was doing at that time. It could hang the system or leave it in an unstable state. It will be better if the task can choose the best location where the thread is not performing any critical operation and thus is safe to be aborted. TPL provides a developer controlled method to cancel tasks. Create a **CancellationTokenSource** where the **CancellationToken** can be obtained. Call **Cancel** method on the source to request for cancellation.

Obtaining a CancellationToken: Tasks2

```
var cs = new CancellationTokenSource();
var ct = cs.Token;
```

Using CancellationSource to cancel:

```
var task1 = new Task(() => {
    while (true) {
        var input = Console.ReadKey(true);
        if (input.Key == ConsoleKey.Escape) {
            cs.Cancel();
            break;
        }}});
```

You can associate a **CancellationToken** with any **Task** you create. You can check the **IsCancellationRequested** property at a safe place in the code to determine if there is a cancel request or throw an exception with the **ThrowIfCancellationRequested** method to stop the current task if cancellation has been requested. To determine if an exception that stopped a task was due to a cancellation check the task's **IsCanceled** property when an exception has been caught.

Stop task if cancellation requested

```
var task2 = new Task(() => {
    while (true) {
        Thread.Sleep(2000);
        Console.WriteLine("Task still running...");
        if (ct.IsCancellationRequested) {
            Console.WriteLine("Task cancelled.");
            break;
        }
    }}, ct);// associated cancellation token with Task

task2.Start();
task1.Start();
task2.Wait();
```

Generating exception on cancel

```
while (true) {
    Thread.Sleep(2000);
    Console.WriteLine("Task still running...");
    ct.ThrowIfCancellationRequested();
}
```

Determining whether task has been cancelled

```
try {
    task2.Start();
    task1.Start();
    task2.Wait();
    Console.WriteLine("Task completed.");
}
catch (Exception) {
    if (task2.IsCanceled) Console.Write("Task was cancelled.");
    else Console.WriteLine("Error occurred in task.");
}
```

2.6 Asynchronous Methods

You can convert any method to run asynchronously. Normally asynchronous method names often end with the word **Async** to differentiate it from a synchronous method. This is not compulsory but common and useful naming convention. Let us declare and implement an asynchronous version of **IAccountController**. Asynchronous methods must always return **Task** or **Task<T>**.

Asynchronous version of IAccountController: Services\IAsyncAccountController.cs

```
public interface IAsyncAccountController : IService {
    Task AddAsync(Account item);
    Task<Account> GetAccountAsync(int code);
    Task<List<Account>> GetAccountListAsync();
    Task<List<Account>> GetAccountsForNameAsync(string name);
}
```

Implementing asynchronous service: Controllers\BankingController.cs

```
[Service(typeof(IAccountController))]
[Service(typeof(ITransactionController))]
[Service(typeof(IAsyncAccountController))]
public class BankingController :
    BaseService,
    IAccountController,
    IAsyncAccountController,
    ITransactionController {
}
```

An asynchronous method creates a **Task** to pass in a delegate to the code to execute. The method must then start the task and return the task. The following shows how to implement the asynchronous methods declared in **IAsyncAccountController**.

Implementing asynchronous methods

```
public Task AddAsync(Account item) {
    var task = new Task(() => {
        var dc = new SymBankDataContext();
        dc.AccountAdd(item.Code,
            item.Type, item.Name,
            item.ZipCode, Authorization.UserName,
            DateTime.Now, item.Balance);
    }); task.Start(); return task;
}

public Task<Account> GetAccountAsync(int code) {
    var task = new Task<Account>(() => {
        var dc = new SymBankDataContext();
        return dc.Accounts.SingleOrDefault(a => a.Code == code);
    }); task.Start(); return task;
}

public Task<List<Account>> GetAccountListAsync() {
    var task = new Task<List<Account>>(() => {
        var dc = new SymBankDataContext();
        return dc.Accounts.ToList();
    }); task.Start(); return task;
}

public Task<List<Account>> GetAccountsForNameAsync(string name) {
    var task = new Task<List<Account>>(() => {
        name = name.ToLower();
        var dc = new SymBankDataContext();
        var query = from account in dc.Accounts
                    where account.Name.ToLower().Contains(name)
                    orderby account.Name
                    select account;
        return query.ToList();
    }); task.Start(); return task;
}
```

Following the above examples you should be able to create an asynchronous version of **ITransactionController** as well and implement it in **BankingController**. You can also convert a synchronous method that depends on other asynchronous methods to be asynchronous using **async** and **await** keywords. This is commonly used in a UI as UI is commonly single-threaded and the UI thread runs synchronously.

Making synchronous method execute asynchronously

```
static async Task<int> DebitNewAccount(Account item, decimal amount) {
    var account = ServiceRepository.Get<IAsyncAccountController>();
    var transaction = ServiceRepository.Get<IAsyncTransactionController>();
    await account.AddAsync(item);
    var transactionCode = await transaction.DebitAsync(item.Code, amount);
    return transactionCode;
}
```

The **async** and **await** keywords goes together. If you use **await** keyword to wait for a asynchronous method to complete, the method itself has to be marked **async**. You can intermix synchronous code with asynchronous method calls in the same method. As the synchronous code runs on the caller thread, this method is safe to be called from the UI thread.

3 Cryptography

3.1 Hashing Algorithm

Hashing algorithm is used for generating digital signatures or password encryption. It is a one-way algorithm so it would be difficult to discover the original value that was hashed. Common algorithms include **MD5**, **SHA1** and **SHA256** to generate the hash code or a digital signature. The larger the **HashSize** the stronger the encryption but the operation will be usually slower and the hashed result will be longer. Hashing the same value gives you the same result. If two hashed passwords are the same you will know the password is correct even though you do not know the password. However, make sure to always use strong passwords.

Using hashing algorithm to generate cipher text: Hashing1.cs

```
HashAlgorithm ha = MD5.Create(); // SHA1.Create();
byte [] password = Encoding.UTF8.GetBytes("p@ssw0rd");
byte [] hashedPassword = ha.ComputeHash(password);
Console.WriteLine(BitConverter.ToString(hashedPassword));
Console.WriteLine(Convert.ToBase64String(hashedPassword));
```

3.2 Symmetric Algorithm

To encrypt data that can be decrypted, use a symmetric algorithm or an asymmetric algorithm. Difference is symmetric algorithm uses the same key for encryption and decryption. In the following example, we use a symmetric algorithm to encrypt data to be written to a file. You need to first generate a key and an initialization vector and save this information so you can use them to decrypt whatever is encrypted. As long as you remember the key and the IV, you can decrypt the data back from the file.

Using a symmetric algorithm: Symmetric1.cs

```
SymmetricAlgorithm sa = Aes.Create();
if (!File.Exists("MyKey.bin")) {
       File.WriteAllBytes("MyKey.bin", sa.Key);
       File.WriteAllBytes("KeyIV.bin", sa.IV );
} else {
    sa.Key = File.ReadAllBytes("MyKey.bin");
    sa.IV  = File.ReadAllBytes("KeyIV.bin");
}
```

Use **CryptoStream** to help you encrypt or decrypt data as it is written or read from a stream. You can get an encryptor or decryptor from the algorithm and pass it to the **CryptoStream** to use.

Creating a file and an encryption stream

```
FileStream stream = new
       FileStream("secure.bin",FileMode.Create,FileAccess.Write);
CryptoStream cstream = new CryptoStream(stream,
      sa.CreateEncryptor(),CryptoStreamMode.Write);
```

Writing text to encryption stream

```
StreamWriter writer = new StreamWriter(cstream);
writer.WriteLine("This will be encrypted!");
writer.Close();
```

Opening a file and creating a decryption stream

```
stream = new FileStream("secure.bin",FileMode.Open, FileAccess.Read);
cstream = new CryptoStream(stream,sa.CreateDecryptor(),
    CryptoStreamMode.Read);
```

Reading from decryption stream

```
StreamReader reader = new StreamReader(cstream);
Console.WriteLine(reader.ReadLine()); reader.Close();
```

3.3 Asymmetric Algorithm

An asymmetric algorithm use a pair of keys, a public and private key where the public key can be used to encrypt the data but only the private key can be used to decrypt the data. The following example shows how to use the algorithm for general data encryption. We will simplify the process and make it reusable by implementing it as a class. You will be able to use it for all your applications and services. In our example, we use the RSA cryptographic algorithm that supports up to 1024-bit encryption by default. Our constructor creates a cryptographic object that uses the RSA algorithm and assigns it to a field so that we can use it from all methods in the class.

Helper class to assist in encryption requirements: Symbion\CryptoHelper.cs

```
using System;
using System.Text;
using System.IO;
using System.Security.Cryptography;

public class CryptoHelper {
    private RSACryptoServiceProvider crypt;
    public CryptoHelper() { crypt = new RSACryptoServiceProvider(); }
}
```

Initially we will provide two methods that will save and load parameters used in the encryption algorithm as XML documents. Note that the **Save** method allows you to determine if private parameters are saved as well which is needed for decryption.

Saving cryptographic parameters

```
public void Save(string path, bool savePrivateKey) {
    File.WriteAllText(path, crypt.ToXmlString(savePrivateKey));
}
```

Loading cryptographic parameters

```
public void Load(string path) {
    crypt.FromXmlString(File.ReadAllText(path));
}
```

We will add methods to encrypt and decrypt text strings. Since encryption algorithms only work with binary data, use the **Encoding** class to convert between text strings and binary data. We will use UTF-8 as the encoding and decoding text format.

Methods to encrypt and decrypt text strings

```
public byte [] Encrypt(string text) {
    byte [] encrypted = Encoding.UTF8.GetBytes(text);
    return crypt.Encrypt(encrypted, false);
}
public string Decrypt(byte [] data) {
    byte [] decrypted = crypt.Decrypt(data, false);
    return Encoding.UTF8.GetString(decrypted);
}
```

Store cryptographic parameters to XML documents: Asymmetric1.cs

```
CryptoHelper ch = new CryptoHelper();
if (!File.Exists("MyKeys.xml")) {
    ch.Save("MyKeys.xml", true);
    ch.Save("PubKey.xml", false);
}
ch.Load("PubKey.xml");  // can only be used to encrypt
byte [] encrypted = ch.Encrypt("Phillip Madden");
ch.Load("MyKeys.xml");  // can be used to encrypt or decrypt
Console.WriteLine(ch.Decrypt(encrypted));
```

Assymetric algorithm is useful for network communication since the public key can be made available to all clients but only the server has the private key. Clients can then securely sent information to the server by encrypting it using the public key. However this can only facilitate a one-way secure communication. For two-way communication, the client can encrypt a symmetric key to send to the server. Only the server can then decrypt the key. Once both sides have the symmetric key, they can use the same key to encrypt and decrypt the messages they send between each other.