* Course Overview
  + Lightweight, high performance ORM
* Intro
* What is Dapper
  + .NET micro ORM
  + Object Relational Mapper(ORM)
    - Mapping between database and .NET objects
  + ORM vs Micro ORM
    - ORM(entity framework)
      * Does mapping
      * Does sql generation
    - Micro ORM
      * Fast mapping
      * Complete control of sql
  + Features of a Micro ORM
    - Lightweight
    - Fast
    - simple/easy to use
* Key Features of Dapper
  + Query and map
  + Query parameters are parameterized meaning they are susceptible to sql injection attacks
  + Performance
  + Simplified API
    - Query method: maps strongly type objects
    - Query method: maps to dynamic objects
    - Execute method: for commands that don’t return results
  + Works with any database
* Show Me Some Code!
  + //entity framework
    - var contacts = context.Contacts.Where(c => c.Id == id);
    - var contacts = context.Contacts.Where(c => c.Id == id)
    - .Select(c => new {
    - FirstName = c.FirstName, LastName = c.LastName
    - })
    - Linq gets converted to sql behind the scene
  + //dapper
    - var contacts = connection.Query<Contact>(
    - “SELECT Id, FirstName, LastName FROM Contacts WHERE Id = @Id”, new { id });
    - Complete control of sql
    - Parameters are parameterized to prevent sql injections
* Database Setup
  + Store procedure example
    - CREATE procedure [dbo].[GetContact]
    - @Id int
    - AS
    - BEGIN
    - SELECT [Id]
    - , [FirstName]
    - , [LastName]
    - , [Company]
    - , [Title]
    - , [Email]
    - FROM [dbo].[Contacts]
    - WHERE Id = @Id;
    - SELECT
    - Id,
    - ContactId,
    - AddressType,
    - StreetAddress,
    - City,
    - StateId,
    - PostalCode
    - FROM [dbo].[Addresses]
    - WHERE ContactID = @Id;
    - END
  + After tables and stored procedures are created, run the Scip.PostDeployment-seed-data.sql
  + Publish the database
  + Use SQL manager studio or azure data studio
  + Establish connection to the database using Windows Authentication and the database server name
* Summary
* Overview
* Installing Dapper
  + Add new project -> class library to hold data layer
  + Model fields must match the column name in the database
  + Install nuget package dapper
  + Add test runner class, .NET core console app
    - Add reference to data layer
* Repository Skeleton
  + Create the definition for the IContactRepository
  + Implement the methods in the ContactRepository
    - public class ContactRepository : IContactRepository
    - {
    - private IDbConnection db;
    - public ContactRepository(string connString)
    - {
    - this.db = new SqlConnection(connString);
    - }
    - …
    - }
  + Add nuget package ‘Microsoft.Extensions.Configuration’ to the Runner project
  + Add nuget package ‘Microsoft.Extensions.Configuration.Json’ to the runner project
    - In the runner program.cs file
    - private static void initialize()
    - {
    - var builder = new ConfigurationBuilder()
    - .SetBasePath(Directory.GetCurrentDirectory())
    - .AddJsonFile(“appsettings.json”, optional: true, reloadOnChange: true);
    - config = builder.Build();
    - }
    - private static IContactRepository CreateRepository()
    - {
    - return new ContactRepository(config.GetConnectionString(“DefaultConnection”));
    - }
  + Add an appsetting.json file to the runner project
    - {
    - “ConnectionStrings”: {
    - “DefaultConnection”: “server=(localdb)\\MSSQLLocalDB;database=ContactsDB;Trusted\_Connection=Yes;”
    - }
    - }
    - Right click on the json file and click properties
      * The field ‘Copy to Output’ choose ‘Copy if newer’
* Basic List Query and List Sup…
  + In the runner program.cs
    - static void Get\_all\_should\_return\_6\_results()
    - {
    - //arrange
    - var repository = CreateRepository();
    - //act
    - var contacts = repository.GetAll();
    - //assert
    - Console.WriteLine($”Count: {contacts.Count}”);
    - Debug.Assert(contacts.Count == 6);
    - contacts.Output();
    - }
  + Triple A pattern for unit test(arrange, act, assert)
  + Add yamldotnet nuget package to the runner project
    - public static class Extensions
    - {
    - public static void Output(this object item)
    - {
    - var serializer = new SerializerBuilder().Build();
    - var yaml = serializer.Serialize(item);
    - Console.WriteLine(yaml);
    - }
    - }
  + In the ContactRepository.cs file
    - public List<Contact> GetAll()
    - {
    - return this.db.Query<Contact>(“SELECT \* FROM Contacts”).ToList();
    - }
* Mapping Objects That Don’t …
  + Occasionaly c# classes don’t match column names precisely
  + In the ContactRepository.cs file
    - public List<Contact> GetAll()
    - {
    - return this.db.Query<Contact>(“SELECT FirstName Fname, LastName, Email, … FROM Contacts”).ToList();
    - }
  + Use aliases in the sql statement when c# classes don’t match column names precisely
* Inserting Data
  + In the program.cs file
    - static int Insert\_should\_assign\_identity\_to\_new\_entity()
    - {
    - //arrange
    - IContactRepository repository = CreateRepository();
    - var contact = new Contact
    - {
    - FirstName = “Joe”,
    - LastName = “Blow”,
    - Email = [joe.blow@gmail.com](mailto:joe.blow@gmail.com),
    - Company = “Microsoft”,
    - Title = “Developer”
    - };
    - //act
    - Repository.Add(contact);
    - //assert
    - Debug.Assert(contact.Id != 0);
    - Console.WriteLine(“\*\*\* Contact Inserted \*\*\*”);
    - Console.WriteLine($”New ID: {contact.Id}”);
    - return contact.Id;
    - }
  + In the ContactRepository.cs file
    - public Contact Add(Contact contact)
    - {
    - var sql = “INSERT INTO Contacts (FirstName, LastName, Email, Company, Title) VALUES (@FirstName, @LastName, @Email, @Company, @Title);” + “SELECT CAST(SCOPE\_IDENTITY() as int)”;
    - var id = this.db.Query<int>(sql, contact).Single();
    - contact.Id = id;
    - return contact;
    - }
* Retrieving a Single Item with a …
  + In program.cs
    - static void Find\_should\_retrieve\_existing\_entity(int id)
    - {
    - //arrange
    - IcontactRepository repository = CreateRepository();
    - //act
    - var contact = repository.Find(id);
    - //assert
    - Console.WriteLine(“\*\*\* Get Contact \*\*\*”);
    - contact.Output();
    - Debug.Assert(contact.FirstName == “Joe”);
    - Debug.Assert(contact.LastName == “Blow’);
    - }
  + In ContactRepository.cs
    - public contact Find(int id)
    - {
    - return this.db.Query<Contact>(“SELECT \* FROM Contacts WHERE Id = @Id”, new { id }).SingleOrDefault();
    - }
  + The sql will be sent in as parameterized query to prevent sql injection attacks
    - Send id anonymous type
    - Can be more explicit and do new { Id = id}
    - But since they have the same name, there is no need to
* Updating Data
  + In program.cs
    - static void Modify\_should\_update\_existing\_entity(int id)
    - {
    - //arrange
    - IContactRepository repository = CreateRepository();
    - //act
    - var contact = repository.Find(id);
    - contact.FirstName = “Bob”;
    - repository.Update(contact);
    - //create a new repository for verification purposes
    - IContactRepository repository2 = CreateRepository();
    - var modifiedContact = repository2.Find(id);
    - //assert
    - Console.WriteLine(“\*\*\* Contact Modified \*\*\*”);
    - modifiedContact.Output();
    - Debug.Assert(modifiedContact.FirstName = “Bob”);
  + In ContactRepository.cs
    - public Contact Update(Contact contact)
    - {
    - var sql =
    - “UPDATE Contacts “ +
    - “SET FirstName = @FirstName, “ +
    - “ LastName = @LastName, “ +
    - “ Email = @Email, “ +
    - “ Company = @Company, “ +
    - “ Title = @Title “ +
    - “WHERE Id = @Id”;
    - this.db.Execute(sql, contact);
    - return contact;
    - }
* Deleting Data
  + In program.cs
    - static void Delete\_should\_remove\_entity(int id)
    - {
    - //arrange
    - IContactRepository repository = CreateRepository();
    - //act
    - repository.Remove(id);
    - //create a new repository for verification purposes
    - IContactRepository repository2 = CreateRepository();
    - var deletedEntity = repository2.Find(id);
    - //assert
    - Debug.Assert(deletedEntity == null);
    - Console.WriteLine(“\*\*\* Contact Deleted \*\*\*”);
    - }
  + In ContactRepository
    - public void Remove(int id)
    - {
    - this.db.Execute(“DELETE FROM Contacts WHERE Id = @Id”, new { id });
    - }
* Implementing Dapper.Contrib
  + Dapper.contrib is a popular add on library for enhancing dapper
    - Provides method for you to write less code for the basic CRUD operations
    - Do this without having to type out all the sql explicitly
  + Install dapper.contrib to the DataLayer project
  + Add a new repository to show dapper contrib
  + In ContactRepository.Contrib
    - public Contact Add(Contact contact)
    - {
    - var id = this.db.Insert(contact);
    - contact.Id = (int)id;
    - return contact;
    - }
    - public Contact Find(int id)
    - {
    - return this.db.Get<Contact>(id);
    - }
    - public List<Contact> GetAll()
    - {
    - return this.db.GetAll<Contact>().ToList();
    - }
    - public void Remove(int id)
    - {
    - this.db.Delete(new Contact { Id = id });
    - }
    - public Contact Update(Contact contact)
    - {
    - this.db.Update(contact);
    - return contact;
    - }
  + If you standard sql without mapping of column names and customizations, dapper contrib is a good tool
  + Dapper contrib tries to put every property it see in the class into the sql statement
  + Use attributes on the class properties to tell dapper
    - In this case there are computed fields not from the database
    - Use the attribute [Computed], tell dabber contrib to ignore it
    - Can also use [Write(false)] to tell dabber contrib to ignore it
    - public class Contact
    - {
    - ….
    - [Computed]
    - public bool IsNew => this.Id == default(int);
    - [Write(false)]
    - public List<Address> Addresses { get; } = new List<Address>();
    - }
  + Dapper contrib may not be best first for more complex sql
* Retrieving Complex Parent-C…
  + Add Contact GetFullContact(int id) to the repository interface
  + Implement the method in ContactRepository
    - public Contact GetFullContact(int id)
    - {
    - var sql =
    - “SELECT \* FROM Contacts WHERE Id = @Id; “ +
    - “SELECT \* FROM Addresses WHERE ContactId = @Id”;
    - using (var multipleResults = this.db.QueryMultiple(sql, new { Id = id }))
    - {
    - var contact = multipleResults.Read<Contact>().SingleOrDefault();
    - var addresses = multipleResults.Read<Address>().ToList();
    - if(contact != null && addresses != null)
    - {
    - contact.Addresses.AddRange(addresses);
    - }
    - return contact;
    - }
    - }
  + QueryMultiple
    - Expect multiple result sets
  + Use Read<type> to extract result set from the result sets
* Create-Update-Delete Operat…
  + Add void Save(Contact contact) to the repository interface
  + Implement the save in ContactRepository
    - public void Save(Contact contact)
    - {
    - if(contact.IsNew)
    - {
    - this.Add(contact);
    - }
    - else
    - {
    - this.Update(contact);
    - }
    - foreach (var addr in contact.Addresses.Where(a => !a.IsDeleted))
    - {
    - addr.ContactId = contact.Id;
    - if (addr.IsNew)
    - {
    - this.Add(addr);
    - }
    - else
    - {
    - this.Update(addr);
    - }
    - }
    - foreach (var addr in contact.Addresses.Where(a => a.IsDeleted))
    - {
    - this.db.Execute(“DELETE FROM Addresses WHERE Id = @Id”, new { addr.Id } );
    - }
    - }
  + Wrap save in TranscationScope() incase things go wrong
    - public void Save(Contact contact)
    - {
    - using (var txScope = new TransactionScope())
    - {
    - if(contact.IsNew)
    - ….
    - txScope.Complete();
    - }
    - }
  + Depending the version of dot net core you are using
    - You may get a ‘Enlisting in Ambient transactions is not supported.’
    - Use more update version of sql client library
    - Add nuget package ‘System.Data.SqlClient’
  + In c# 8 feature we can ommit the block
    - using var txScope = new TransactionScope();
    - if( contact.isNew)
    - ….
* Summary
* Overview
* Stored Procedures and Dyna…
  + Create new repository for accessing database with store procedures
    - public class ContactRepositorySP : IContactRepository {}
  + implement the methods in ContactRepositorySP
    - public Contact GetFullContact(int id)
    - {
    - using (var multipleResults = this.db.QueryMultiple(“GetContact”, new { Id = id }, commandType: CommandType.StoredProcedure))
    - {
    - var contact = multipleResults.Read<Contact>().SingleOrDefault();
    - var addresses = multipleResults.Read<Address>().ToList();
    - if(contact != null && addresses != null)
    - {
    - contact.Addresses.AddRange(addresses);
    - }
    - return contact;
    - }
    - }
    - public Contact Find(int id)
    - {
    - return this.db.Query<Contact>(“GetContact”, new { Id = id }, commandType: CommandType.StoredProcedure).SingleOrDefafult();
    - }
  + If you have a stored procedure that has multiple sql queries
    - If you use .Query<Type> it will only use the first queries and ignore the other
    - If you use .QueryMultiple() it will use all the queries
  + Implement save using stored procedure and dynamic parameters
    - public void Save(Contact contact)
    - {
    - using var txScope = new TransactionScope();
    - var parameters = new DynamicParameters();
    - parameters.Add(“@Id”, value: contact.Id, dbType: DbType.Int32, direction: ParameterDirection.InputOutput);
    - parameters.Add(“@FirstName”, contact.FirstName);
    - parameters.Add(“@LastName”, contact.LastName);
    - parameters.Add(“@Company”, contact.Company);
    - parameters.Add(“@Title”, contact.Title);
    - parameters.Add(“@Email”, contact.Email);
    - this.db.Execute(“SaveContact”, parameters, commandType: CommandType.StoredProcedure);
    - contact.Id = parameters.Get<int>(“@Id”);
    - foreach (var addr in contact.Addresses.Where(a => !a.IsDeleted))
    - {
    - addr.ContactId = contact.Id;
    - var addParams = new DynamicParameters( new
    - {
    - ContactId = addr.ContactId,
    - AddressType = addr.AddressType,
    - StreetAddress = addr.StreetAddress,
    - City = addr.City,
    - StateId = addr.StateId,
    - PostalCode = addr.PostalCode
    - });
    - addrParams.Add(“@Id”, addr.Id, DbType.Int32, ParameterDirection.InputOutput);
    - this.db.Execute(“SaveAddress”, addrParams, commandType: CommandType.StoreProcedure);
    - addr.Id = addrParams.Get<int>(“@Id”);
    - }
    - foreach (var addr in contact.Addresses.Where(a => a.IsDeleted)
    - {
    - this.db.Execute(“DeleteAddress”, new { Id = addr.Id }, commandType: CommandType.StoreProcedure);
    - }
    - txScope.Complete();
    - }
  + Dynamic parameters
    - Common way to specify parameters when you need to send parameters into stored procedures
    - Anonymous type could also be used
    - But dynamic parameters make it explicit
    - And helps if you need to change name if parameter names don’t and property names don’t match precisely
  + Implement the Remove method
    - public void Remove(int id)
    - {
    - this.db.Execute(“DeleteContact”, new { Id = id }, commandType: CommandType.StoreProcedure);
    - }
* List Support for In Operator
  + Ability to use array as values for an ‘in’ operator in a where clause
  + Create new repository
    - public class ContactRepositoryEx {}
  + Example method
    - public List<Contact> GetContactsById(params int[] ids)
    - {
    - return this.db.Query<Contact>(“SELECT \* FROM Contacts WHERE ID IN @Ids”, new { Ids = ids }).ToList();
    - }
  + ‘Params’ keyword
    - Specify a method parameter that takes a variable number of arguments
  + In the program.cs file
    - static void List\_support\_should\_produce\_correct\_results()
    - {
    - //arrange
    - var repository = CreateRepositoryEx();
    - //act
    - var contracts = repository.GetContactsById(1, 2, 3);
    - //assert
    - Debug.Assert(contacts.Count == 3)
    - contacts.Output();
    - }
* Dynamic Capabilities
  + example
    - public List<dynamic> GetDynamicContactsById(params int[] ids)
    - {
    - return this.db.Query(“SELECT \* FROM Contacts WHERE ID IN @Ids”, new { Ids = ids }.ToList();
    - }
  + dynamic objects expose members such as properties and methods at run time instead of compile time
    - enables you to create objects to work with structures that do not match a static type or format
* Bulk Insert
  + If we need several items we need to insert into our database, dapper enables us to send in a array
    - public int BulkInsertContacts(List<Contact> contacts)
    - {
    - var sql =
    - “INSERT INTO Contacts(FirstName, LastName, Email, Company, Title) VALUES (@FirstName, @LastName, @Email, @Company, @Title); “ +
    - “SELECT CASE(SCOPE\_IDENTITY() as int)”;
    - return this.db.Execute(sql, contacts);
    - }
  + In program.cs
    - static void Bulk\_insert\_should\_insert\_4\_rows()
    - {
    - //arrange
    - var repository = CreateRepositoryEx();
    - var contacts = new List<Contact>
    - {
    - new Contact { FirstName = “Charles”, LastName = “Barkley” },
    - new Contact { FirstName = “Scottie”, LastName = “Pippen” },
    - new Contact { FirstName = “Tim”, LastName = “Duncan” },
    - new Contact { FirstName = “Patrick”, LastName = “Ewing” }
    - };
    - //act
    - var rowsAffected = repository.BulkInsertContacts(contacts);
    - //assert
    - Console.WriteLine($”Rows inserted: {rowsAffected}”);
    - Debug.Assert(rowsAffected == 4);
    - }
  + Not a performance optimization, more a syntax optimization
* Literal Replacements
  + Only supported for boolean and numeric
    - If they were support for strings, it would be subject to sql inject
  + In ContactRepositoryEx
    - public List<Address> GetAddressesByState(int stateId)
    - {
    - return this.db.Query<Address>(“SELECT \* FROM Addresses WHERE StateId = {=stateId}”, new { stateId}).ToList();
    - }
    - Syntax {=stateId} is need for literal replacements
* Multi Mapping
  + One to one or one to many relationship to object and you want to load early load objects with a single query
  + Example query
    - SELECT \* FROM Contacts AS C
    - INNER JOIN Addresses AS A
    - ON A.ContactId = C.Id;
  + In ContactRepositoryEx
    - public List<Contact> GetAllContactWithAddresses()
    - {
    - var sql = “SELECT \* FROM Contacts AS C INNER JOIN Addresses AS A ON A.ContactId = C.Id”;
    - var contactDict = new Dictionary<int, Contact>();
    - var contacts = this.db.Query<Contact, Address, Contact>(sql, (contact, address) =>
    - {
    - if(!contact.Dict.TryGetValue(contact.Id, out var currentContact))
    - {
    - currentContact = contact;
    - contactDict.Add(currentContact.Id, currentContact);
    - }
    - currentContact.Addressess.Add(address);
    - return currentContact;
    - });
    - return contacts.ToList();
    - }
  + Query<Contact, Address, Contact>
    - First Contact is the first object your are mapping to
    - Address is the child object
    - Second Contact is the return type
* Other Databases: MySQL
  + Dapper can be used for any database
  + Example
    - public class ContactRepsoitoryMySql
    - {
    - …
    - public contactRepositoryMySql(string connString)
    - {
    - this.db = new MySqlConnection(connString);
    - }
    - …
    - }
  + In appsettings.json
    - {
    - “ConnectionStrings” : {
    - “DefaultConnection: “server …”,
    - “MySqlConnection”: “server=….”
    - }
    - }
  + Dapper works with provider for specific database you are using
* Async Capabilities
  + Dapper supports async versions of all the primary methods
    - public async Task<List<Contact> GetAllAsync()
    - {
    - var contacts = await this.db.QueryAsync<Contact>(“SELECT \* FROM Contacts”);
    - return contacts.ToList();
    - }
* Summary