C#

Daily Exercises

Version 2.0 ITC

Introduction

1. This document contains all of the exercises you will complete during the C# / ASP.NET Web API track of the backend academy.

2. You will need access to Visual Studio 2022 Community, but since it only runs on a Windows machine we have provided you virtual machines (VMs) that you will use to do the labs. These VMs are just “Windows desktops in the cloud”.

DAY 1
Wednesday, 17-July

Day 1 Exercises

EXERCISE 1

Create a `HelloMe` console application that prompts the user for their name and then echos back to the console the message "Hello *name*!". For example, if the user entered Natalie as their name, the program would respond "Hello Natalie!"

Don't forget to create Git repositories and push to GitHub or GitLab (whichever you are using).

EXERCISE 2

Create a `FormatTheData` console application. It should experiment with formatting options found at https://docs.microsoft.com/en-us/dotnet/standard/base-types/formatting-types

Here is some data to start with.

```
string name = "Zachary";
DateTime bday = new DateTime(1990, 5, 20);
decimal pay = 69759.25;

Console.WriteLine(
   "V1- {0} was born on {1:dd-MM-yyyy} and earns {2:C}",
   name, bday, pay);

Console.WriteLine(
   $"V2- {name} was born on {bday:dd-MM-yyyy} and earns {pay:C}");
```

OUTPUT (as run in the USA)

```
V1- Zachary was born on 20-05-1990 and earns $69,759.25
V2- Zachary was born on 20-05-1990 and earns $69,759.25
```

EXERCISE 3

Create a `ReadTheData` console application. When you read data using `Console.ReadLine()`, it returns the data as a string. When you are prompting the user for numbers, you must use a conversion function to convert the string to a numer. There are many options and often it is a matter of preference, including:

```
Convert.ToInt32()  Convert.ToSingle()  Convert.ToDouble()
Convert.ToDecimal()

int.Parse()   float.Parse()   double.Parse()   decimal.Parse()
```

In this application, prompt the user for two integers and then display the sum, difference, product, and quotient (+, -, x, /) of the two numbers.

Then, prompt the user for two doubles and repeat that process.

EXERCISE 4

Create a `MortgageCalculator` console application. It will accept input from a user regarding a loan and then display the expected monthly payment.

The formula for calculating the monthly payment is:

$$Pmt = \frac{A \times MIR}{1-(1+MIR)^{-NP}}$$

Variable	Description
A	Amount borrowed
MIR	Monthly interest rate (yearly rate / 1200)
NP	Number of monthly payments (loan length X 12)
Pmt	Estimated payment

To format the payment as money, you can use either of the techniques below:

```
Console.WriteLine("Your estimated payment is {0:C}", payment);
Console.WriteLine($"Your estimated payment is {payment:C}");
```

TRANSCRIPT OF PROGRAM

```
How much are you borrowing? 53000.00
What is your interest rate? 7.625
How long is your loan (in years)? 15

Your estimated payment is $495.09
```

EXERCISE 5

Create a console application named `PriceQuoter`.

Users will enter a product code and quantity. Your application will look up the price of the item, apply any discounts, and display the price of the order.

	--------------------PER UNIT PRICE---------------------		
Product Code	Quantity 1-24	Quantity 25-50	Quantity 51+
BG-127	$18.99	$17.00	$14.49
WRTR-28	$125.00	$113.75	$99.99
GUAC-8	$8.99	$8.99	$7.49

If a person orders 250 or more units (a "large" order), the order price will be discounted an additional 15%.

Display the product code, the item count quantity, the per-unit price, and the total order price. If there is a discount, note that too and display the large-order discounted price.

Run your program several times and test it with different inputs.

<u>EXERCISE 6</u>

Create a console application named `AddessDecypher`.

Define a variable called encoded “Address” that contains the value:

```
     Betty Smallwood|3329 Duchess|Erath, Texas
```

Note: A pipe character is used to split 3 fields - name, address, and city(comma) state.

Using string methods, extract the 4 individual fields and display them. For example:

```
Name: Betty Smallwood
Address: 3329 Duchess
City: Erath
State: Texas
```

DAY 2
Thursday, 18-July

Day 2 Exercises

EXERCISE 1

Create a `NameAndAge` console application. Prompt the user for their name and how old they will be on 31-December.

Then create a method named GetBirthYear(). Pass it the person's age on 31-December. Within the method, calculate and return their birth year.

Then display a message resembling "Natalie was born in 1984" (but use the name entered and the birth year calculated!)

Once this works, add a loop that repeats this code 3 times.

Once that works, modify the loop so that it prompts the user "Do you want to enter another (yes/no)?" If the user responds yes, Yes, YES, etc (regardless of case), then iterate again.

EXERCISE 2

Create a `ClassPlay` console application. It will create a simple class, create objects from it, and then use those objects.

Create a class named SportsPlayer that represents someone who plays a sport.

- Add at least 5 sensible properties, such as player name, sport, years experience playing, etc.
- Add 2 sensible methods, one of them should print out the information about the sport player

Create three instances of SportsPlayer. Set all of the fields and call all of the methods on each.

Did you use a constructor as one of your methods? If not, add a constructor and create objects using it.

EXERCISE 3

Create a `ProcessTestScores` console application. In it:

- Call a GetTestScores() method that reads and returns an array of 6 test scores
- Call a GetHighestScore() method. Pass it the test score array and have it return the highest test score.
- Call a GetAverageScore() method. Pass it the test score array and have it return the average test score.
- Call a GetLowestScore() method. Pass it the test score array and have it return the lowest test score.
- Display the highest, average, and lowest test scores

EXERCISE 4

Create a `FavoriteMovies` console application. You will read in a list of movies (prompt the user to continue as long as they want). Store the movies in a `List<>` of strings.

Sort the list.

Provide the user with the ability to search the list for a movie.

- They can so a “Partial Search” where they enter a word or phrase and the algorithm displays all matches
- They can do an “Exact Search” where enter a movie name and it returns whether the movie was found.

Do a case insensitive search in both cases. Repeat the searches as long as the user wants to search.

EXERCISE 5

Create a console application named `ManagingFamily`. In this application, you will create a class named Person. It will contain 3 properties (name, age, and gender), a constructor, and a method named Display() that displays the person.

Allow the user to enter as many “people” as they want. Prompt for name, age, and gender. Then create a Person object. Add that person to a List<Person>.

Provide a menu where people can choose to:

- Display all people
- Display aall people of a selected gender
- Display all people between a specified age range.

EXERCISE 6 (Challenge)

Create a console application named `TimeMath`.

Allow the user to enter a time in the format hh:mm (ex. 10:35)   Don't worry about AM or PM for this exercise.

Then perform a time calculation by adding a number of minutes entered by the user to the time (ex: 30).

Display the new time (ex: 11:05)

DAY 3 and 4
Friday, 19-July and
Monday, 22-July

Day 3 and 4 Exercises

EXERCISE 1

Create a `PetStore` console application that manages the inventory for a pet store.

Part 1

Create a base class called `InventoryItem` with the following properties: `Id`, `Name`, `Description`, `Price`, and `Quantity`.

Create derived classes for *specific* item types depending on the store type chosen. Here's an example for the pet store:

- `FoodItem` with properties: `Brand`, `FoodType` (enum Dry or Wet), and `AnimalType` (Dog, Cat, etc.).
- `AccessoryItem` with properties: `Size` and `Color`.
- `CageItem` with properties: `Dimensions` and `Material`.
- `AquariumItem` with properties: `Capacity` and `Shape`.
- `ToyItem` with properties: `Material` and `RecommendedAge`.

Part 2

Create a list of base class objects using `List<InventoryItem>` to store the inventory of your store.

You can manually load 15 or 20 items into the array. Or you create a data file and load it using a `StreamReader` (https://learn.microsoft.com/en-us/dotnet/api/system.io.streamreader?view=net-8.0)

Format the file something like this:

```
1,Food,Dog Food,High-quality dry dog food,15.49,50,BrandA,Dry,Dog
2,Accessory,Cat Collar,Cat collar with bell,5.99,20,Small,Blue
3,Cage,Parrot Cage,Large parrot cage with toys,120.00,5,24x24x36,Wire
4,Aquarium,50-gallon Fish Tank,50-gallon fish tank with
stand,200.00,3,50,Rectangle
5,Toy,Hedgehog Wheel,Exercise wheel for
hedgehogs,12.49,8,Plastic,6months+
```

Part 3

Allow your user to use a menu to look up items, view all items, or add items to the inventory.

The menu will provide options such as:
1- Show all items   (showing only the name, type (derived class name), and the id)

2- Show an item's details (allow the user to enter an ID and display all of the properties of that item)

Part 4

If time permits, give them the option to purchase the item. Reduce the quantity of the item by 1 when this happens, and display the amount due.

Add additional features that you think are interesting if you have time.

DAY 5
Tuesday, 23-July

Day 5 Exercises

<u>EXERCISE 1</u>

Create a `LINQandNumbers` console application. In it, create an array with 50 random numbers in it from 1 to 10000. Use `Math.Random()` to fill the array.

Write LINQ queries to find answers to the following problems. Encapsulate each LINQ query and the displaying of results in its own method.

1. List the numbers in ascending order
2. List the numbers under 100 in descending order
3. List the even numbers in original order
4. Find the minimum number, the maximum number, and the sum of all the numbers

<u>EXERCISE 2</u>

Create a `LINQandStrings` console application. In it, create a List<string> that contains at least 20 fruits and vegetables.

Ex: `"Apple", "Banana", "Strawberry", "Cherry", "Mango", "Blueberry", …`

Write LINQ queries to find answers to the following problems. Encapsulate each LINQ query and the displaying of results in its own method.

1. List all strings that start with a B or b.
2. List all strings that contain the word "betty"
3. List strings that start with the letters A-M
4. How many strings start with the letters N-Z
5. What is the longest string in the list

<u>EXERCISE 3</u>

Create a `LINQandObjects` console application. Define a class called Person that has the following properties:

- Name
- Gender
- Age
- Hometown

Now, create a List<Person> that contains at least 20 people.

Write LINQ queries to find answers to the following problems. Encapsulate each LINQ query and the displaying of results in its own method.

1. List the males under 25
2. Provide a list of all the distinct hometowns (do not duplicate any hometowns) in ascending order
3. Provide a list of people sorted by hometown, and within hometown by name
4. What is the average age of all women and the average age of all men
5. Provide a list of hometowns and how many people are from that hometown (Challenge!)

DAY 6
Wednesday, 24-July

Day 6 Exercises

EXERCISE 1

Create a console applicatio named `Delegate_Math`. In this application, you will experiment with the syntax and use of delegates.

Start by creating a class named `BasicMath` that has methods named `Add`, `Subtract`, `Multiply`, and `Divide`. Each accepts two `double` parameters and returns the result of the named operation.

```
public double Add(double a, double b)
{
    return a + b;
}
```

In the namespace scope above the `Program` class, create a delegate type named `MathOperation` that can be used to invoke any of the four math functions (`Add`, `Subtract`, etc.). That is, it can be used to call a function with two double parameters that returns a double.

In the `Main` method, create an instance of the delegate and use it to call the `Add` method. Display the results.

```
BasicMath math = new BasicMath();
MathOperation mathOp = new MathOperation(math.Add);
// etc.
```

Repeat using a delegate to call the other three math functions.

EXERCISE 2

Create a console applicatio named `Delegate_Logger`. In this application, you will again experiment with the syntax and use of delegates.

Start by creating a class named `Logger` that has methods named `Info()`, `Warning()`, and `Error()`. Each accepts one `string` parameter (message) and displays a message similar to:

```
Console.WriteLine("[INFO] " + message);
```

The difference between each method's implementation is the message in square brackets (ex: [INFO], [WARNING], and [ERROR])

In the namespace scope above the `Program` class, create a delegate type named `LoggingOperation` that can be used to invoke any of the three log functions (`Info`, `Warning`, etc.).

In the `Main` method, create an instance of the delegate and use it to call the `Info()` method. Repeat using a delegate to call the other two loggingfunctions.

Once this works, create a `LoggingOperation` delegate that references an anonymous method that displays a message using the format:

```
Console.WriteLine("[ALERT] " + message);
```

Test your new logging method.

EXERCISE 3

Create a console application named `PaymentProcessingApp`. It will demonstrate using a delegate in an Object-Oriented (OO) scenario. We will define a PaymentProcessor class that can process different types of payments (e.g., Mastercard, Visa, etc). Then, we will use a delegate to handle different payment methods dynamically.

Step 1: Define the various payment processing methods

Define a class named `PaymentMethods`. In it, define three methods named `ProcessMastercardPayment`, `ProcessVisaPayment`, and `ProcessDiscoverPayment` Each is passed a string containing the account number (yes, this is simplified) and an amount of the payment. Each returns a bool indicating success.

The methods themselves can simply each what type of payment is being processed, the account, and the amount. Use a random number to return false in 10% of the time, otherwise return true.

Step 2: Define the delegate named `PaymentHandler` that matches the signature of the "process payment" methods above.

Define a delegate that will represent the method signature for processing payments.

Step 3: Create the `PaymentProcessor` class

Create a class named `PaymentProcessor` that will use the delegate to process payments dynamically. It contains one method named `ProcessPayment` that calls the correct process payment method based on a delegate.

```
bool ProcessPayment(PaymentHandler paymentHandler, string
accountNumber, double amount)
{
   return paymentHandler(amount);
}
```

Step 4: Use the Delegate in the `Main` method

Within the `Main` method, use the `PaymentProcessor` class and delegate to process payments.

```
PaymentMethod paymentMethods = new PaymentMethods();

PaymentProcessor paymentProcessor = new PaymentProcessor();

PaymentHandler mastercardHandler =
   new PaymentHandler(paymentMethods.ProcessMastercardPayment);

bool isOk = paymentProcessor.ProcessPayment(creditCardHandler,
   "1234-1111-2222-1234", 100.00);

if(! isOkay)
  ConsolWriteLine("[ALERT] Payment processing failed");
```

Test each of your payment methods.

DAY 7
Thursday, 25-July

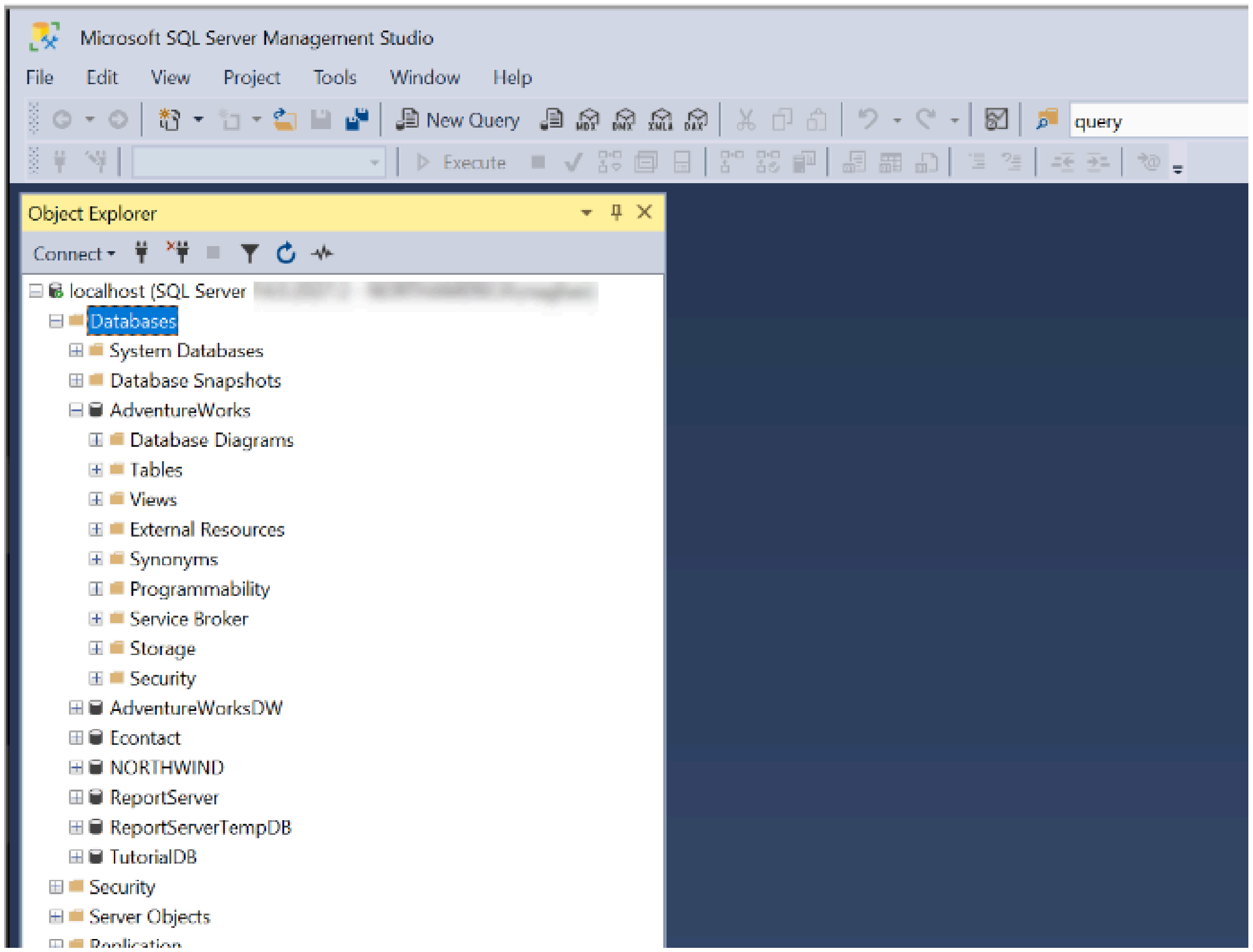
Day 7 Exercises

LOGGING iN TO SQL SERVER

Your AWS Workspace has SQL Server installed on it. To connect, launch SQL Server Management Studio (SSMS).

1. The server name should be okay and is your computer name
2. Choose SQL Authentication
3. Use the user name "sa" (no quotes)
4. Use the password "Password@123" (no quotes)
5. You may need to check the "Trust Server certificate" checkbox

When you successfully log in, you will see the Object Explorer window on the left. For example:



We have pre-installed 2 databases: Northwind and pubs.

What's in there?? In the Object Explorer, expand the database node to view the list of databases.

Expand the database node (+) that contains the tables you want to view.

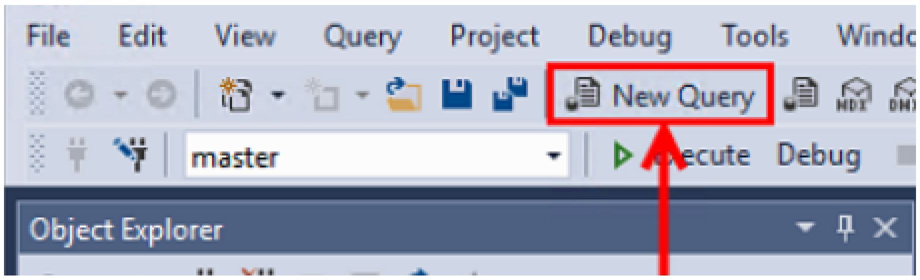
Expand the "Tables" node to see a list of all the tables in that database.

Right-click on a specific table to view its properties, data, or to perform other tasks related to that table.

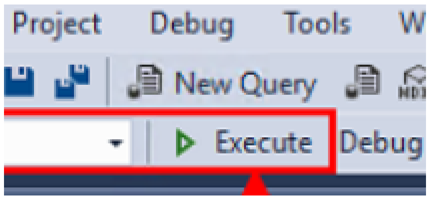
<u>EXERCISE 1</u>

Take a few minutes to find a quick video on running queries in SSMS. Or, charge ahead and see if you can figure it out from the hints below.

If you click on the database you want to query in the explorer window, you can write a query that runs against that database.



After entering your query, you can execute that query.



Results appear in a window below the query.

Start by running the simple query:

```
SELECT * from Suppliers;
```

When you figure that out, take a few minutes to complete the tasks below.

- How many tables are in the northwind database?
- What is the name of the table that lists the categories of products sold?
- What is the name of the table that lists the products sold?

- What is the name of the table that lists the names of the suppliers of products?

Take a few minutes to return queries for the following problem statements. Any properties mentioned below may not be the correct column name. Look things up before you write the query.

- List all category data
- List category id and category name of all categories sorted by id
- List the id, product name, and unit price of all products
- List the id, product name, and unit price of all products that cost less that $10.00
- Do a join between categories and products so that you can list the id, product name, unit price, and category name of all products

Challenge: List the product id, product name, unit price, category name, and supplier name of all products that cost between %5.00 and $20.00

EXERCISE 2

Create a console applicatio named `NorthwindGrocery_ConsoleApp`. In this application, you will provide the user with a menu that allows them to query and display data from the northwind database.

Your C# program should present the user will the following menu choices:

1 - List all categories
2 - List products in a specific category
3 - List products in a price range
4 - List all suppliers
5 - List products from a specific category
6 - Quit

For option 2, you will also have to prompt the user for the category ***name***
For option 3, you will have to prompt the user for minimum and maximum prices
For option 5, you will have to prompt the user for the supplier ***id***

EXERCISE 3 (Challenge)

Note: This project will likely extend into tomorrow.

Create a C# program called `Dealership`. It will require you to create a new database as well as a C# application to manage the cars in a used car dealership.

Your database should be named Dealership. It will have two tables: Cars and Sales.

The Cars table represents the inventory of used cars we have on the lot and ones we have sold in the past. Its columns are:

- id (auto-increment number)
- inventory_number
- vehicle_identification_number
- make
- mode
- year
- odometer_reading
- color
- price
- status (available, sold)

The Sales table keeps track who bought which car. Its columns are:

- id (auto-increment number)
- inventory_number (from Cars)
- sales_date
- customer name
- customer phone
- payment method
- payment amount

Load sample data into the tables.

Your C# program should present the user will the following menu choices:

1 - List all available cars
2 - List available cars with less than a specific odometer reading
3 - List available cars with a specific make and model
4 - List available cars between a specific price range
5 - Sell a car
6 - Change a car's price
7 - Delete a car from inventory
8 - Quit

For option 2, prompt for the maximum odometer reading acceptable
For option 3, prompt for the make and model
For option 4, prompt for the price range
For option 5, prompt for the inventory_number of the car being sold, display the car data, then prompt for the data needd for the Sales table and add it to the table, then mark the car as sold in the Cars table (see bonus below)
For option 6, prompt for the inventory_number of the car being sold, display the car data, then prompt for the new price and update the Cars table
For option 7, prompt for the inventory_number of the car being sold, display the car data, then confirm the user really wants to delete, the delete the car from the Cars table

As a bonus for the Car sales option, create a file named with the inventory number followed by the word recipt (ex: Inv123_receipt or DK15C9_receipt) for each sale that contains a minimally acceptable formatted receipt that can be emailed to the customer.

DAY 8
Friday, 26-July

Day 8 Exercises

THIS IS A “CATCH UP” DAY.

We expect you will be working on the Dealership lab today.

You can also use it to finish watching any videos you need to finish or review. And finish up any labs that are still outstanding or that you want to experiment with some more.

DAY 9
Monday, 29-July

Day 9 Exercises

EXERCISE 1

Create the ASP.NET Core Web API project `CityInfo` that you saw in the course. Follow along in the course to configure it and then run the application and experiment with the endpoints that were automatically generated.

EXERCISE 2

Create an ASP.NET Core Web API project named `EmployeesInformation` that will provide GET endpoints for employee resources. Use the knowledge gained from exercise 1 to configure it. This time, however, you will generate new endpoints. Refer to the videos on resources and routing to complete this exercise.

Part 1
Begin by creating a new class named `EmployeesController` that inherits from `ControllerBase`. It will serve as your Employees API controller.

Next, create a `Models` folder with an `Employee` class in it. Within it, define properties for `ID`, `Name`, `JobTitle`, and `Salary`. `Name` should default to the empty string. `JobTitle` can be null. `Salary` should default to 0.

Now, create a class named `EmployeesDataStore` that manages a hard-coded list of employees. Define a property named `Employees` that returns a list of 4-5 employees.

Finally, go back to your `EmployeesController` and create two methods:

- `GetEmployees()` returns a `JsonResult` containing the list of employees from EmployeesDataStore. The route for this endpoint should be /`api/employees`. It will use the `EmployeesDataStore` to fetch and return the list of employees.
- `GetEmployee(int id)` returns a `JsonResult` containing the one employee from EmployeesDataStore with a matching id. The route for this endpoint should be /`api/employees/`*`id`* (where id is a number matching an employee ID). It will use the `EmployeesDataStore` to find and return the employee being requested.

Complete the tasks needed to configure routing and test the endpoint in the browser.

Part 2
Create a database in SQL Server named `EmployeesDatabase`. In it, define an `Employee` table containing columns for `ID`, `Name`, `JobTitle`, and `Salary`.
Seed the database with 8-10 employees.

Then, modify `EmployeesDataStore` to load the list from this database (rather than have a hard-coded list of employees)

Part 4 (Optional)
Add a DELETE endpoint for employees that is passed an employee id. Its purpose is to delete the matching employee in the database. However, your controller should not work directly with the database.

This means you will need to add a new method to `EmployeesDataStore` named `DeleteEmployee(int id)` that will be passed an employee id and will delete that employee record from the database.

Part 5 (Optional)
Add a POST endpoint for employees that is passed information fort a new employee. Its purpose is to add the employee to the database. However, your controller should not work directly with the database.

This means you will need to add a new method to `EmployeesDataStore` named `InsertEmployee(string name, string jobTitle, decimal salary)` that will add that employee record to the database. The id should be an autoincrement field.

NOTE: We will not work on a lab with Entity Framework until tomorrow.

DAY 10
Tuesday, 30-July

Day 10 Exercises

<u>EXERCISE 1</u>

In this exercise, you will create an ASP.NET Core Web API project named `LocalGym` that interacts with a database using Entity Framework Core. The theme of the project will be a gym where members can book sessions with trainers.

Part 1

You will need to create the following Entity Framework entities:

- Member
  - MemberId (Primary Key)
  - FirstName
  - LastName
  - Email
  - JoinDate
- Trainer
  - TrainerId(Primary Key)
  - FirstName
  - LastName
  - Speciality
  - FeePer30Minutes
  - HireDate
- Session
  - SessionId(Primary Key)
  - CustomerId (Foreign Key)
  - TrainerId (Foreign Key)
  - SessionDate
  - Duration

Implement the following relationships:

- A Member can have multiple training sessions.
- A Trainer can conduct multiple training sessions.

Next, you will need to create a DbContext class with properties:

- Members
- Trainers
- Sessions

Part 2
Now, write code to create the initial database migration and create/seed the tables. Refer to the video on Entity Framework Core for help with all of these steps.

Part 3
You should create a `LocalGymRepository` class that implements the Repository pattern. You will minimally have the following methods:

- GetMembersAsync()
- GetMemberAsync(int id);
- GetTrainersAsync()
- GetTrainerAsync(int id);
- GetSessionsForTrainerAsync(int id)
- GetSessionsForMemberAsync(int id);

Challenge: Once you have this complete, implement methods to create a member and update a member.

Part 4
Now we will switch over to creating the Web API controllers. Let's start by creating a `MembersController` and a `TrainersController`. Each should have endpoints that "get all" resources (ex: get all members or get all trainers) and get a specific resource (ex: get a member by id, get a trainer by id). The endpoints should be:

/api/members
/api/members/id
/api/trainers
/api/trainers/id

The "Using Entity Framework Core in Your Controllers" covers how to do this while working with the repository..

Complete all tasks so that you can test your 4 API endpoints.

Part 5
You will test your endpoints using the Postman tool. It is a very popular tool for testing APIs and it is important to understand how to use it. You can download it from here: https://www.postman.com/downloads/

Once you install and launch Postman, use it to test the GET endpoints at:

/api/members
/api/members/id
/api/trainers
/api/trainers/id

Fix any bugs!

Part 6
Now go back to your `TrainersController`. Implement the POST and PUT endpoints to create and update trainers..

When you complete them, test in Postman.

What's Next?
If you have any time left, you can implement POST and PUT endmoints for members. Test!

Then, you should implement a `SessionsController` that lets you get all sessions, get all sessions for a specific member, all sessions for a specific trainer, and all sessions for a specific member/trainer combination. Test!

Finally, if there's STILL more time left, implement a POST endpoint to create a new session. Test!

DAY 11
Wednesday, 31-July

Day 11 Exercises

EXERCISE 1

In this exercise, you will continue working in your `LocalGym` project from yesterday.

Part 1
Secure your API by implementing token-based security. You will essentially be repeating the steps found in:

- Demo: Creating a Token
- Demo: Requiring and Validating a Token
- Demo: Using Information from the Token in your Controller

Part 2
Document your API endpoints using Swagger.

Part 3
Study the section on testing your API using .html files. Follow the process described in:

- Demo: Testing with .http Files

Unfortunately, we do not have the ability to test deployment to Azure. But you will be working with the cloud in the last academy (DevOps) of your internship.

DAY 12
Thursday, 1-August

Day 12 Exercises

EXERCISE 1

In this exercise, you will continue working in your `LocalGym` project from yesterday.

Using the information you learned in today's debugging course, implement a Global Error Handler. You do NOT need to define custom errors. Instead, replace the path in `app.UseExceptionHandler()` with an action similar to that shown in the video "Global Error Handling Using the Built-In Middleware"

EXERCISE 2

Complete the interactive lab "StarChart Web API using ASP.NET Core". The link is in the channel.

NOTE: This is your last assignment before the C# capstone. Tomorrow is a "catch up" day. There is no new learning. So if today's exercises carry over into tomorrow, there is no issue.

DAY 13
Friday, 2-August

Day 13 Exercises

There are no exercises today. It is a “Catch Up” day to allow you to finish labs and review any content you feel you would like to understand better.

The link to the capstone will be available today also. So once you have completed your “learning phase”, feel free to start work on the capstone,