using System;

using System.Collections.Generic;

using System.IO;

using System.Threading.Tasks;

using System.Xml.Serialization;

using Newtonsoft.Json;

using Serilog;

namespace Serialization

{

public class Program

{

private static ILogger \_logger = null;

public static async Task Main(string[] args)

{

\_logger = ConfigureLogger();

// "" strings you need to escape some characters with \

// @"" strings you don't

var xmlFilePath = @"C:\revature\persons.xml";

var jsonFilePath = @"C:\revature\persons.json";

//var data = GetInitialData();

//var data = DeserializeXmlFromFile(xmlFilePath);

//List<Person> data = DeserializeJsonFromFileAsync(jsonFilePath).Result; // synchronously waits

List<Person> data = await DeserializeJsonFromFileAsync(jsonFilePath);

ModifyData(data);

//SerializeXmlToFile(xmlFilePath, data);

//SerializeJsonToFileAsync(jsonFilePath, data).Wait(); // synchronously waiting for the Task to complete

await SerializeJsonToFileAsync(jsonFilePath, data);

}

public static ILogger ConfigureLogger()

{

var logger = new LoggerConfiguration()

.WriteTo.Console()

.WriteTo.File(@"C:\revature\logs.log")

.MinimumLevel.Warning()

.CreateLogger();

return logger;

}

public static async Task SerializeJsonToFileAsync(string jsonFilePath, List<Person> data)

{

// we will do this with JSON.NET aka Newtonsoft Json

// we use NuGet to get these dependencies

string json = JsonConvert.SerializeObject(data);

// exceptions should be handled here, ignored for sake of time

await File.WriteAllTextAsync(jsonFilePath, json);

// switching from sync to async:

// 1. call the async version of whatever method is going to access network/disk/other slow thing.

// 2. await the task returned by that method

// 3. add the async modifier to your method

// 4. make your method return a Task

// 5. add "Async" suffix to the name of your method.

// (6. repeat from step 1 on up to any callers of your method)

}

public static async Task<List<Person>> DeserializeJsonFromFileAsync(string jsonFilePath)

{

// Serilog supports "structured logging"

\_logger.Information("Loading JSON from file {file}", jsonFilePath);

string json;

try

{

json = await File.ReadAllTextAsync(jsonFilePath);

}

catch (IOException ex)

{

\_logger.Error(ex, "Exception while trying to read file {file}", jsonFilePath);

return null;

}

var data = JsonConvert.DeserializeObject<List<Person>>(json);

return data;

}

public static void ModifyData(List<Person> data)

{

var person = data[0];

person.Id += 10;

}

public static List<Person> DeserializeXmlFromFile(string xmlFilePath)

{

// XmlSerializer serialization can be configured on the serializer object

var serializer = new XmlSerializer(typeof(List<Person>));

FileStream fileStream = null;

try

{

fileStream = new FileStream(xmlFilePath, FileMode.Open);

return (List<Person>)serializer.Deserialize(fileStream);

}

catch (IOException ex)

{

Console.WriteLine($"Error while opening {xmlFilePath} for writing: {ex.Message}");

}

catch (InvalidOperationException ex)

{

Console.WriteLine($"Error while serializing: {ex.Message}");

}

finally // finally block always runs whether, no-exception, handled-exception, or unhandled-exception

{

// this "do something if not null"

//if (fileStream != null)

//{

// fileStream.Dispose();

//}

fileStream?.Dispose(); // is exact same as commented-out code above

// null-conditional operator

}

return null;

}

public static void SerializeXmlToFile(string xmlFilePath, List<Person> data)

{

// XmlSerializer was made pre-generics and has not been updated

var serializer = new XmlSerializer(typeof(List<Person>));

// "using statement" replaces a try-finally-dispose on an disposable object.

try

{

//using (fileStream = new FileStream(xmlFilePath, FileMode.Create))

//{

// serializer.Serialize(fileStream, data);

// // at the end of the using block, the object is disposed automatically (regardless

// // of any unhandled exceptions)

//}

// from c# 8, we have "using declaration" - instead of having to indent a whole block,

// the implicit dispose will happen at the end of the current block

using var fileStream = new FileStream(xmlFilePath, FileMode.Create);

// to make this async, i would load the file contents asynchronously into

// a memorystream, then give the serializer that memorystream

// xmlserializer doesnt directly have async support

serializer.Serialize(fileStream, data);

}

catch (IOException ex)

{

Console.WriteLine($"Error while opening {xmlFilePath} for writing: {ex.Message}");

}

catch (InvalidOperationException ex)

{

Console.WriteLine($"Error while serializing: {ex.Message}");

}

}

public static List<Person> GetInitialData()

{

return new List<Person>

{

new Person

{

Id = 1,

Name = "Billy",

Address = new Address

{

Street = "123 Main St",

City = "Dallas",

State = "TX"

}

},

new Person

{

Id = 2,

Name = "Sam"

}

};

}

}

}