***CONSTRUIR UNA APLICACIÓN SERVERLESS API INCLUYENDO TESTEO***

* Crear Nuevo Proyecto en Vstudio : AWS Serverless Application with Tests (.NET Core — C#)
* Nombre : **SampleAPI**
* En Select BluePrint : **ASP. NET Core Web API**

**Archivos creados:**

**aws-lambda-tools-defaults.json**

* Este archivo describe la configuración predeterminada de tu cuenta de AWS con respecto a tu aplicación, como la región, el usuario predeterminado, el entorno de tiempo de ejecución (etiquetado como "marco"), cualquier plantilla sin servidor que describa la configuración de la aplicación, etc.

**LambdaEntryPoint.cs**

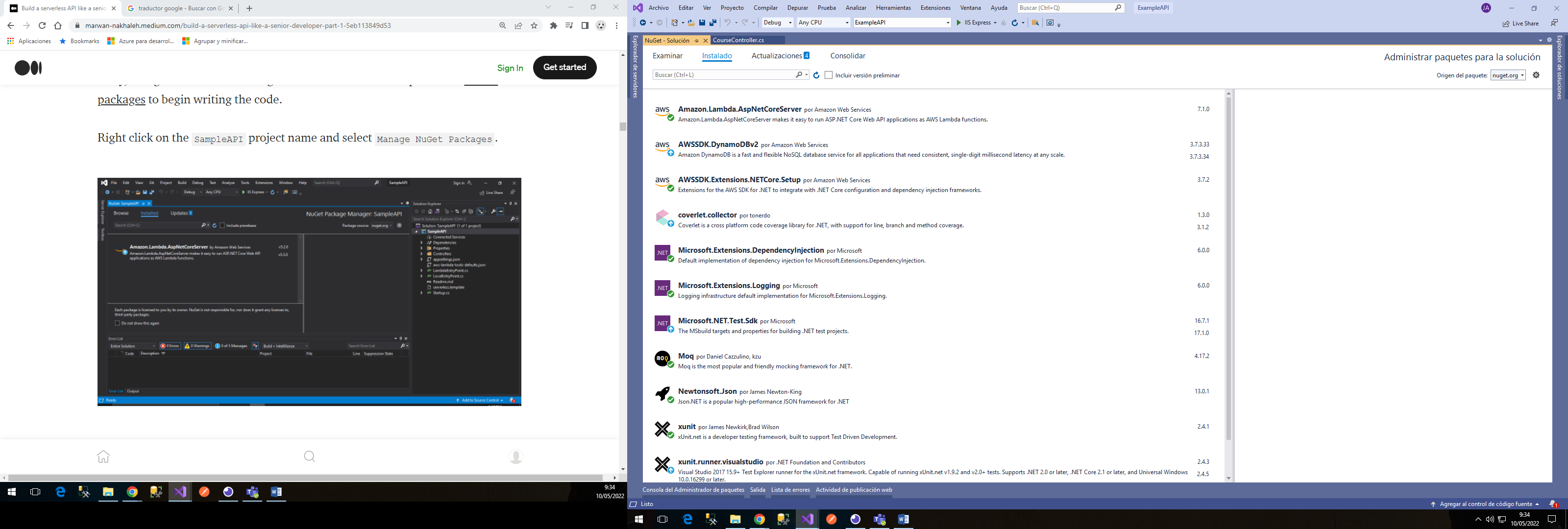
* El código de este archivo es lo que se ejecuta cuando un proyecto corre dentro de AWS Lambda. El método UseStartup permite la ejecución de la clase Startup.cs

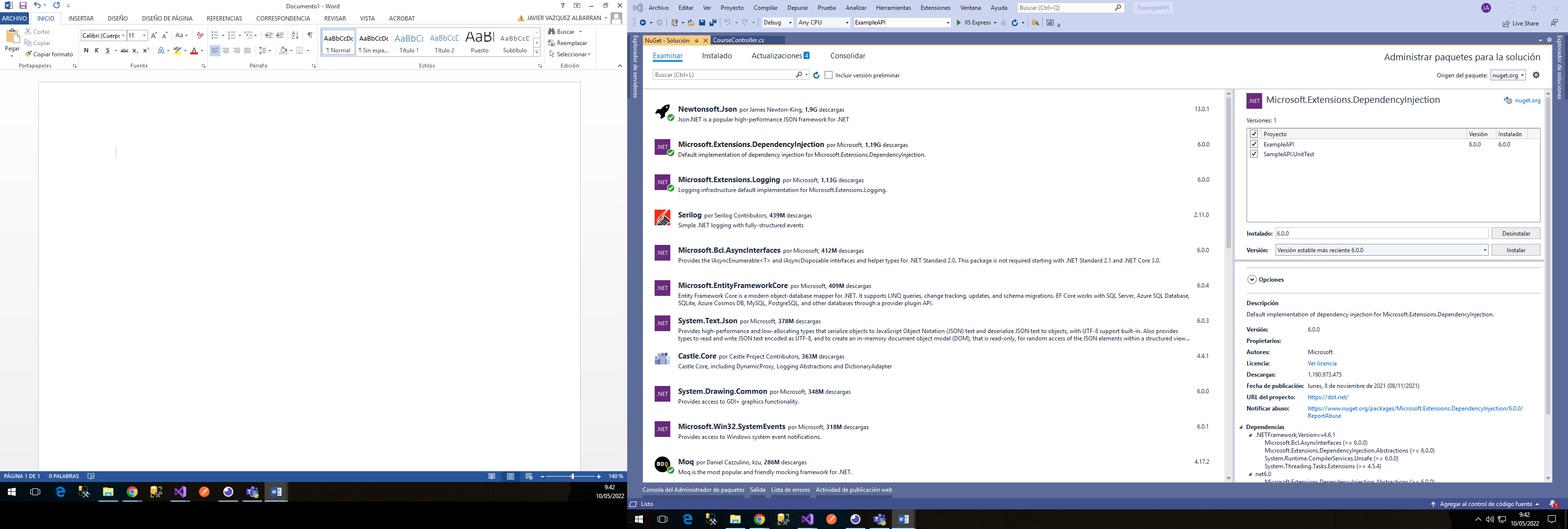
**LocalEntryPoint.cs**

* El código en este archivo es lo que se ejecuta cuando un proyecto se ejecuta en tu máquina local mientras estás probando el desarrollo.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

* Instalar Nuget





Una captura de pantalla de una computadora

Descripción generada automáticamente

* Nueva Carpeta : SampleAPI/**Domain**
* Nueva clase : /Domain/**Video.cs**

public class Video

{

[JsonProperty("title")]

public string? Title { get; set; }

[JsonProperty("description")]

public string? Description { get; set; }

[JsonProperty("subsection")]

public string? SubsectionName { get; set; }

[JsonProperty("course")]

public string? CourseName { get; set; }

public Video(string title, string subsection, string description, string course)

{

if (string.IsNullOrEmpty(title))

{

throw new ArgumentException("Video needs a title");

}

if (string.IsNullOrEmpty(subsection))

{

throw new ArgumentException("Video needs an subsection name");

}

if (string.IsNullOrEmpty(description))

{

throw new ArgumentException("Video needs a description");

}

if (string.IsNullOrEmpty(course))

{

throw new ArgumentException("Video needs a course");

}

Title = title;

SubsectionName = subsection;

Description = description;

CourseName = course;

}

}

* Nueva clase : /Domain/**Course.cs**

public class Course

{

[JsonProperty("title")]

public string? Title { get; }

[JsonProperty("instructor")]

public string? Instructor { get; }

[JsonProperty("category")]

public string? Category { get; }

[JsonProperty("description")]

public string Description { get; }

[JsonProperty("videos")]

public List<Video>? Videos { get; }

public Course(string title, string instructor, string category, string description, List<Video> videos)

{

if (string.IsNullOrEmpty(title))

{

throw new ArgumentException("Course needs a name");

}

if (string.IsNullOrEmpty(instructor))

{

throw new ArgumentException("Course needs an instructor");

}

if (string.IsNullOrEmpty(category))

{

throw new ArgumentException("Course needs a category");

}

if (string.IsNullOrEmpty(description))

{

throw new ArgumentException("Course needs a description");

}

Title = title;

Instructor = instructor;

Category = category;

Description = description;

Videos = videos;

}

}

Ahora vamos a incluir pruebas de test unitario dentro de nuestro proyecto. Se trabaja con **XUnit** ya que es mucho más flexible que NUnit.

* Renombrar la clase ValuesControllerTests.cs a QueryUnitTests.cs
* Renombrar el método TestGet() a TestGetCoursesQuery()

que tendrá el siguiente aspecto de momento:

using Microsoft.VisualStudio.TestTools.UnitTesting;

using System;

namespace SampleAPI.Tests

{

[TestClass]

public class QueryUnitTests

{

[TestMethod]

public void TestGetCoursesQuery()

{

}

}

}

Ahora vamos a crear la capa Query para poder testear

* Sobre el proyecto **SampleAPI** agregar nueva carpeta: **Data**, en la que crearemos una capa repository
* En la carpeta **Data** creamos dos Nuevas Carpetas: **Queries y Repositories.** Esto hará que permitamos que el almacenamiento de datos sea un área separada del resto del código. Esto hará que nuestro código tenga en cuenta cualquier método de almacenamiento de datos, ya sea que obtengamos datos directamente de una base de datos o de una API o de donde sea.

Primero crearemos nuestro viewmodel

* Creamos una Nueva Carpeta : **SampleAPI**/**Models** y en ella una nueva clase : **CourseModel.cs**

public class CourseModel

{

public List<Course> coursesList { get; }

public CourseModel(List<Course> course\_list)

{

coursesList = course\_list;

}

}

* Creamos en la carpeta **SampleAPI/Data**/**Queries** una nueva clase abstracta: **BaseQuery.cs**

public abstract class BaseQuery

{

public abstract CourseModel GetCourseInformation();

}

y una clase : **Query.cs**

public class Query : BaseQuery

{

public override CourseModel GetCourseInformation()

{

List<Course> courseList = new List<Course>();

Course course = new Course("Baby's First Course", "Marwan Nakhaleh", "APIs", "this is a course", new List<Video>());

courseList.Add(course);

return new CourseModel(courseList);

}

}

* Ahora vamos a realizar el testeo.

Modificar clase **SampleAPI.Tests**/**QueryUnitTests.cs**

using Microsoft.VisualStudio.TestTools.UnitTesting;

using SampleAPI.Data.Queries;

using SampleAPI.Models;

using System;

using Xunit;

namespace SampleAPI.UnitTest

{

[TestClass]

public class QueryUnitTests

{

[Fact]

public void TestGetCoursesQuery()

{

Query queryService = new Query();

CourseModel allCourseData = queryService.GetCourseInformation();

Xunit.Assert.Equal("Baby's First Course",

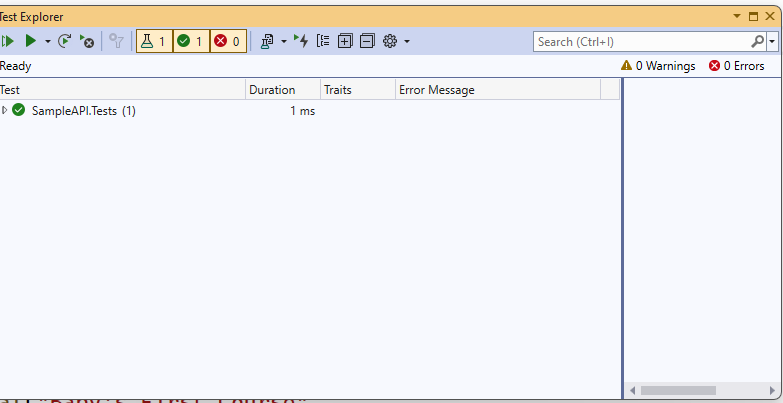
allCourseData.coursesList.First(course => course.Title == "Baby's First Course").Title);

}

}

}

Dentro de esta clase con el botón dcho: ***Ejecutar Pruebas***



Vemos que la prueba se ejecuta correctamente sin errores. (*Lo único que hemos hecho ha sido testear que CourseModel.GetCourseInformation devuelve el curso que estamos testeando*)

Ahora vamos a hacer un testeo pero desde datos reales:

* En la carpeta **SampleAPI/Data/Repositories** añadimos una clase abstracta : **BaseRepository.cs**

public abstract class BaseRepository

{

private List<Course> \_courses { get; }

public BaseRepository()

{

\_courses = new List<Course>();

}

public abstract List<Course> GetCourses();

}

* Creamos una *Nueva Carpeta* en SampleAPI.Tests : **Mocks**
* En esta carpeta creamos una *Nueva Clase* : **MockRepository.cs**

public class MockRepository : BaseRepository

{

public override List<Course> GetCourses()

{

List<Course> \_courses = new List<Course>();

Course course1 = new Course("Baby's First Course", "Marwan Nakhaleh", "APIs", "this is a course", new List<Video>());

Course course2 = new Course("Yet Another Course", "Marwan Nakhaleh", "APIs", "this is another course", new List<Video>());

\_courses.Add(course1);

\_courses.Add(course2);

return \_courses;

}

}

Necesitamos ahora que el constructor pueda aceptar un parámetro *BaseRepository*.

* Modificamos la clase **SampleAPI/Data/Queries**/**Query.cs**

public class Query : BaseQuery

{

**private readonly BaseRepository \_repository;**

**public Query(BaseRepository repository)**

**{**

**\_repository = repository;**

**}**

public override CourseModel GetCourseInformation()

{

List<Course> courseList = new List<Course>();

Course course = new Course("Baby's First Course", "Marwan Nakhaleh", "APIs", "this is a course", new List<Video>());

courseList.Add(course);

return new CourseModel(courseList);

}

}

* Modificamos la clase **SampleAPI.Test/QueryUnitTests.cs**

using Microsoft.VisualStudio.TestTools.UnitTesting;

using SampleAPI.Data.Queries;

using SampleAPI.Data.Repositories;

using SampleAPI.Models;

using SampleAPI.Tests.Mock;

using System;

using Xunit;

namespace SampleAPI.UnitTest

{

[TestClass]

public class QueryUnitTests

{

private BaseRepository \_dataRepository;

public QueryUnitTests()

{

\_dataRepository = new MockRepository();

}

[Fact]

public void TestGetCoursesQuery()

{

Query queryService = new Query(\_dataRepository);

CourseModel allCourseData = queryService.GetCourseInformation();

Xunit.Assert.Equal("Baby's First Course",

allCourseData.coursesList.First(course => course.Title == "Baby's First Course").Title);

}

}

}

* Volvemos a ejecutar el *Run Test* y vemos que ahora tenemos un repositorio óptimo accesible a nuestra capa query

A screenshot of a computer

Description automatically generated

Ahora necesitamos modificar nuestro **SampleAPI/Data/Queries/Query.cs** para que devuelva datos del repositorio en lugar de simplemente inventarlos.

* Modificamos el método **GetCourseInformation**

public override CourseModel GetCourseInformation()

{

List<Course> courseList = new List<Course>();

Course course = new Course("Baby's First Course", "Marwan Nakhaleh", "APIs", "this is a course", new List<Video>());

courseList.Add(course);

return new CourseModel**(\_repository.GetCourses()**);

}

* Y volvemos a ejecutar el test

A screenshot of a computer

Description automatically generated

Ahora vamos a crear el *controlador* que devuelva datos para el cliente consumiendo la API

* Renombrar el controlador de **SampleAPI/Controllers/ValuesController.cs** a **CourseController.cs**
* Modificar [Route("api/[controller]")] por **[Route("api/v1/[controller]")]**

namespace SampleAPI.Controllers

{

[Route("api/v1/[controller]")]

public class CourseController : ControllerBase

{

private readonly BaseQuery \_queryService;

public CourseController(BaseQuery queryService)

{

\_queryService = queryService;

}

[HttpGet]

public IEnumerable<Course> Get()

{

return \_queryService.GetCourseInformation().coursesList;

}

}

}

\*\* agregamos logs a las capas de consulta y repositorio

* Modificamos la clase **/Data/Repositories**/**BaseRepository.cs**

public abstract class BaseRepository

{

protected List<Course> \_courses { get; }

protected ILogger \_logger;

public BaseRepository(ILoggerFactory logger)

{

\_courses = new List<Course>();

\_logger = logger.CreateLogger("BaseRepository");

}

public abstract List<Course> GetCourses();

}

* Añadimos nueva clase en la carpeta /Data/Repositories : **Repository.cs**

Nota: sealed hace que la clase esté sellada y no pueda heredarse

public sealed class Repository : BaseRepository

{

public override List<Course> GetCourses()

{

\_logger.LogInformation("returning course information");

return \_courses;

//return null;

}

public Repository(ILoggerFactory logger) : base(logger)

{

\_logger = logger.CreateLogger("Repository");

}

}

* Modificamos la clase Data/Queries/**BaseQuery.cs**

public abstract class BaseQuery

{

protected ILogger \_logger;

protected BaseRepository \_repository;

public BaseQuery(BaseRepository repository, ILoggerFactory logger)

{

\_repository = repository;

\_logger = logger.CreateLogger("BaseQuery");

}

public abstract CourseModel GetCourseInformation();

}

* Actualizamos la clase **SampleAPI.Data.Queries/Query**

public class Query : BaseQuery

{

public Query(BaseRepository repository, ILoggerFactory logger) : base(repository, logger)

{

\_repository = repository;

\_logger = logger.CreateLogger("Query");

}

public override CourseModel GetCourseInformation()

{

\_logger.LogInformation("retrieving course data for client");

return new CourseModel(\_repository.GetCourses());

}

}

* Modificar **SampleAPI**/**Startup.cs**

public void ConfigureServices(IServiceCollection services)

{

services.AddControllers();

**services.AddAWSService<IAmazonDynamoDB>();**

**services.AddScoped<BaseRepository, Repository>();**

**services.AddScoped<BaseQuery, Query>();**

}

* Modificar **CourseController.cs**

namespace SampleAPI.Controllers

{

[Route("api/v1/[controller]")]

public class CourseController : ControllerBase

{

private readonly BaseQuery \_queryService;

private readonly ILogger \_logger;

public CourseController(BaseQuery queryService, ILoggerFactory logger)

{

\_logger = logger.CreateLogger("CourseController");

\_queryService = queryService;

}

[HttpGet]

public IEnumerable<Course> Get()

{

\_logger.LogInformation("starting GET /api/v1/courses");

return \_queryService.GetCourseInformation().coursesList;

}

}

}

* **Ctrl + F5 para ejecutar en local**

Welcome to running ASP.NET Core on AWS Lambda

Probar en Imnsomnia : <https://localhost:44331/api/v1/course>

Responderá con el 200 OK

A screenshot of a computer

Description automatically generated

* Como paso final vamos a agregar logs a las capas de consulta y repositorio y limpiar un poco la configuración (Esto ya está pasar al siguiente paso)
* Modificamos **BaseRepository.cs**

public abstract class BaseRepository

{

protected List<Course> \_courses { get; }

protected ILogger \_logger;

public BaseRepository(ILoggerFactory logger)

{

\_courses = new List<Course>();

\_logger = logger.CreateLogger("BaseRepository");

}

public abstract List<Course> GetCourses();

}

* De igual manera con **Repository.cs**

public sealed class Repository : BaseRepository

{

//sealed (sellada) hace que ninguna clase pueda heredar del repository

public override List<Course> GetCourses()

{

\_logger.LogInformation("returning course information");

return \_courses;

//return null;

}

public Repository(ILoggerFactory logger) : base(logger)

{

\_logger = logger.CreateLogger("Repository");

}

}

* Ahora **BaseQuery.cs**

public abstract class BaseQuery

{

protected ILogger \_logger;

protected BaseRepository \_repository;

public BaseQuery(BaseRepository repository, ILoggerFactory logger)

{

\_repository = repository;

\_logger = logger.CreateLogger("BaseQuery");

}

public abstract CourseModel GetCourseInformation();

}

* Finalmente **Query.cs**

public class Query : BaseQuery

{

public Query(BaseRepository repository, ILoggerFactory logger) : base(repository, logger)

{

\_repository = repository;

\_logger = logger.CreateLogger("Query");

}

public override CourseModel GetCourseInformation()

{

\_logger.LogInformation("retrieving course data for client");

return new CourseModel(\_repository.GetCourses());

}

}

Ctrl + f5 de nuevo

Ahora para poder pasar el testeo con datos reales haremos:

* **SampleAPI.Tests/Mocks/MockRepository.cs**

public class MockRepository : Data.Repositories.BaseRepository

{

public MockRepository(ILoggerFactory logger) : base(logger)

{

\_logger = logger.CreateLogger<MockRepository>();

}

public override List<Course> GetCourses()

{

List<Course> \_courses = new List<Course>();

Course course1 = new Course("Baby's First Course", "Marwan Nakhaleh", "APIs", "this is a course", new List<Video>());

Course course2 = new Course("Yet Another Course", "Marwan Nakhaleh", "APIs", "this is another course", new List<Video>());

\_courses.Add(course1);

\_courses.Add(course2);

return \_courses;

}

}

* Finalmente **: SampleAPI.Tests/QueryUnitTests.cs**

[TestClass]

public class QueryUnitTests

{

private BaseRepository \_dataRepository;

private ILoggerFactory \_logger;

public QueryUnitTests()

{

//Un objeto [ILoggerFactory](https://learn.microsoft.com/es-es/dotnet/api/microsoft.extensions.logging.iloggerfactory?view=net-8.0) que se usa para crear una instancia de [NullLogger](https://learn.microsoft.com/es-es/dotnet/api/microsoft.extensions.logging.abstractions.nulllogger?view=net-8.0) que no registra nada.

\_logger = new NullLoggerFactory();

\_dataRepository = new MockRepository(\_logger);

}

[Fact]

public void TestGetCoursesQuery()

{

Query queryService = new Query(\_dataRepository, \_logger);

CourseModel allCourseData = queryService.GetCourseInformation();

Xunit.Assert.Equal("Baby's First Course", allCourseData.coursesList.FirstOrDefault(course => course.Title == "Baby's First Course").Title);

}

}

Volver a probar el testeo

A screenshot of a computer

Description automatically generated

FIN PARTE PRIMERA

SEGUNDA PARTE: Vamos a utilizar una *DynamoDB* situada en nuestro local para enlazarla con la *ApiGateWay* y poder poblar de datos nuestra aplicación.

Para poder utilizar este tutorial necesitamos instalarnos las siguientes herramientas:

* Docker Desktop
* DynamoDB local JAR executable
* AWS SAM

DynamoDB: <https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/DynamoDBLocal.DownloadingAndRunning.html>

AWS SAM Cli : <https://docs.aws.amazon.com/es_es/serverless-application-model/latest/developerguide/install-sam-cli.html>

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

* Comenzamos arrancando Docker Desktop
* Abrimos una terminal de comandos navegando al directorio donde hayamos instalado DynamoDB local.JAR, es decir, en DynamoDBLocal\_lib
* Arrancamos la base de datos local de DynamoDB ejecutando el siguiente comando:

docker run -p 8000:8000 amazon/dynamodb-local -jar DynamoDBLocal.jar -inMemory –sharedDb

Explicación del uso:

* Si utilizas la opción -sharedDb, DynamoDB crea un único archivo de base de datos denominado *shared-local-instance.db*. Todos los programas que se conectan con DynamoDB obtienen acceso a este archivo. Si eliminas el archivo, perderás todos los datos que hayas guardado en él.
* Si omites -sharedDb, el archivo de base de datos recibirá el nombre *myaccesskeyid\_region.db*, con el ID de clave de acceso de AWS y la región de AWS que aparecen en la configuración de la aplicación. Si eliminas el archivo, perderás todos los datos que hayas guardado en él.
* Si usas la opción -inMemory, DynamoDB no escribe ningún archivo de base de datos. En lugar de ello, todos los datos se escriben en la memoria y ninguno de ellos se guarda cuando cierra DynamoDB.
* Si utilizas la opción -optimizeDbBeforeStartup, también debes especificar el parámetro -dbPath para que DynamoDB pueda encontrar el archivo de base de datos.
* Los SDK de AWS para DynamoDB requieren que se especifiquen en la configuración de la aplicación un valor de clave de acceso y un valor de región de AWS. A no ser que utilices la opción -sharedDb o -inMemory, DynamoDB usará estos valores para asignar el nombre al archivo de base de datos local. Estos valores no tienen que ser valores de AWS válidos para la ejecución local. Sin embargo, tal vez te convenga utilizar valores válidos para que pueda ejecutar el código en la nube más adelante cambiando simplemente el punto de enlace que estés utilizando.
* El puerto 8000:8000 representa el puerto al que queremos enviar la instancia

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

* Ahora abrimos un terminal separado y pasamos el siguiente comando:

aws dynamodb list-tables --endpoint-url [http://localhost:8000](http://localhost:8000/)

Esto nos permite confirmar que la comunicación con dynamodb es correcta y nos enseñará un conjunto vacio de tablas :

TableNames[]

* Ahora en el mismo terminal, creamos una tabla Course con el atributo de clave de partición

aws dynamodb create-table --table-name Course --attribute-definitions AttributeName=title,AttributeType=S --key-schema AttributeName=title,KeyType=HASH --billing-mode PAY\_PER\_REQUEST --endpoint-url [http://localhost:8000](http://localhost:8000/)

* Ahora vamos a poblar esta tabla Course con datos, para ello creamos un fichero YAML con datos y copiarlo en una carpeta especifica

***/curso\_aws/DynamoDBLocal/batch\_put.yaml***

Fichero batch\_put.yaml:

RequestItems:

Course:

- PutRequest:

Item:

title:

S: 'Intro to Serverless APIs'

instructor:

S: 'Marwan Nakhaleh'

description:

S: 'Learn how to build a rest API running on Lambda served through API Gateway.'

category:

S: 'API'

- PutRequest:

Item:

title:

S: 'Intro to Angular'

instructor:

S: 'Marwan Nakhaleh'

description:

S: 'Learn the key concepts of Javascript front-end framework development by example with Angular 12.'

category:

S: 'frontend'

ReturnConsumedCapacity: INDEXES

ReturnItemCollectionMetrics: NONE

* En la terminal de comandos navegar a la carpeta donde tenemos el fichero yaml e introducir el siguiente comando:

aws dynamodb batch-write-item --cli-input-yaml file://batch\_put.yaml --endpoint-url [http://localhost:8000](http://localhost:8000/)

* En la misma carpeta ***/curso\_aws/DynamoDBLocal*** crear un nuevo fichero llamado ***expression\_criteria.json*** con el siguiente contenido:

{  
":v1": {"S": "Intro to Serverless APIs"}  
}

* En la terminal de comandos

aws dynamodb query --table-name Course --key-condition-expression "title = :v1" --expression-attribute-values file://expression\_criteria.json --return-consumed-capacity TOTAL --endpoint-url [http://localhost:8000](http://localhost:8000/)

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

nota: Si falla esto es porque necesita la vesion v2 de AWS Cli. Instalarla y en Agregar y Quitar Programas de Windows desinstalar la version v1 para que trabaje solo con esta y Volver a ejecutar el comando

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

recibiremos algo parecido a esto:

Graphical user interface, application

Description automatically generated

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Ahora ya podemos configurar nuestro código en el proyecto:

* Fichero appsettings.Development.json

{

"AWS": {

"Region": "us-east-1",

"DynamoDB": {

"ServiceURL": "http://localhost:8000",

"TableName": "Course"

}

}

}

* Crear una nueva carpeta en SampleAPI llamada **Configurations**
* Crear nueva clase /Configurations/**ConfigurationKeys.cs**

namespace ExampleAPI.Configurations

{

public class ConfigurationKeys

{

public static string DynamoDBServiceURLKey = "AWS:DynamoDB:ServiceURL";

public static string DynamoDBTableNameKey = "AWS:DynamoDB:TableName";

}

}

* Crear nueva clase /Configurations/**DbConfiguration.cs**

public class DbConfiguration

{

//Cuando extraigamos los valores para la URL de DynamoDB y el nombre de la tabla de los archivos

//appsettings.<environment>.json, se almacenarán en estas variables.

public string ServiceURL { get; set; }

public string TableName { get; set; }

}

* Creamos en **Startup.cs** un nuevo método ~~static~~ llamado **GetDbConfiguration** que devolverá un objeto de tipo **DbConfiguration**. Este método obtendrá la configuración actual de la base de datos para el entorno e inyectará un nuevo objeto *DbConfiguration* en nuestra aplicación en tiempo de ejecución.

private ~~static~~ DbConfiguration GetDbConfiguration()

{

DbConfiguration dbConf = new DbConfiguration()

{

ServiceURL = Configuration.GetValue<string>(ConfigurationKeys.DynamoDBServiceURLKey),

TableName = Configuration.GetValue<string>(ConfigurationKeys.DynamoDBTableNameKey)

};

return dbConf;

}

* Ahora también en **Startup.cs** actualizamos **ConfigureServices.**

public void ConfigureServices(IServiceCollection services)

{

services.AddControllers();

services.AddAWSService<IAmazonDynamoDB>();

**services.AddSingleton(GetDbConfiguration());**

services.AddScoped<BaseRepository, Repository>();

services.AddScoped<BaseQuery, Query>();

}

* Ahora modificamos SampleAPI/Data/Repositories/**Repository.cs** para hacer consultas a nuestra base de datos actual

public sealed class Repository : BaseRepository

{

private readonly string \_dbServiceUrl;

private readonly string \_dbTableName;

public Repository(DbConfiguration dbConf, ILoggerFactory logger) : base(logger)

{

\_dbServiceUrl = dbConf.ServiceURL;

\_dbTableName = dbConf.TableName;

\_logger = logger.CreateLogger("Repository");

}

public override List<Course> GetCourses()

{

\_logger.LogInformation("returning course information");

return \_courses;

}

}

NOTA : Puede que no recoja bien las rutas de dbConfig.ServiceURL y dbConfig.TableName desde el fichero aws lambda defaults.json, si ocurre esto darselo directamente así

public Repository(DbConfiguration dbConf, ILoggerFactory logger) : base(logger)

{

//\_dbServiceUrl = dbConf.ServiceURL;

\_dbServiceUrl = "http://localhost:8000";

//\_dbTableName = dbConf.TableName;

\_dbTableName = "Course";

\_logger = logger.CreateLogger("Repository");

}

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

* Ahora añadimos un método privado en **Repository.cs** activando el cliente database

private AmazonDynamoDBClient GetDbClient()

{

if (!string.IsNullOrEmpty(\_dbServiceUrl))

{

AmazonDynamoDBClient client;

AmazonDynamoDBConfig ddbConfig = new AmazonDynamoDBConfig()

{

ServiceURL = \_dbServiceUrl

};

client = new AmazonDynamoDBClient(ddbConfig);

return client;

}

else

{

throw new Exception("No database URL has been provided!");

}

}

* Ahora actualizamos **GetCourses()** para que use este nuevo método

public override List<Course> GetCourses()

{

try

{

AmazonDynamoDBClient client = GetDbClient();

ScanResponse result = ScanAllItems(client);

AddAllCoursesToCourseList(result);

}

catch (Exception e)

{

\_logger.LogError(e.Message);

}

return \_courses;

}

* Ahora vamos a crear un método que simplifique el rotorno de todos los objetos desde nuestra tabla de Course. Creamos un nuevo método private en **Repository.cs** llamado **ScanAllItems()** que devolverá un objeto *ScanResponse* y que recibirá como parámetro un objeto *AmazonDynamoDBClient*

private ScanResponse ScanAllItems(AmazonDynamoDBClient client)

{

ScanRequest request = new ScanRequest

{

TableName = \_dbTableName,

};

try

{

var response = client.ScanAsync(request);

ScanResponse result = response.Result;

return result;

}

catch (Exception e)

{

\_logger.LogError("Unable to scan items: " + e.Message);

return null;

}

}

* Ahora añadimos los cursos recibidos a la lista de cursos. Creamos un nuevo método privado void llamado **AddCourseToCourseList** pasándole como parámetro un objeto *ScanResponse*

private void AddAllCoursesToCourseList(ScanResponse response)

{

//ScanResponse tendrá un atributo item desde el cual

//podemos recuperar informacion de cada registro

//Se puede acceder a cada atributo por el nombre del atributo

//asi como por su valor de ipo de datos primitivo, por ejemplo,

//item['title'].S obtenemos el atributo title y luego la cadena

//almacenada en el valor

if (response != null)

{

foreach (Dictionary<string, AttributeValue> item in response.Items)

{

Course course = new Course(item["title"].S, item["category"].S, item["instructor"].S, item["description"].S, new List<Video>());

\_courses.Add(course);

}

}

else

{

throw new Exception("No retrieved courses to add.");

}

}

Ctrl + F5

Copiar la URL y llevarla a Imnsomnia añadiendo /api/v1/course

Esto nos devolverá toda la lista de los cursos de la base de datos de dynamoDB Local