Министерство науки и высшего образования РФ

Пензенский государственный университет

Кафедра «Вычислительная техника»

**ОТЧЕТ**

по лабораторной работе №4

по дисциплине «Программные технологии проектирования программного обеспечения вычислительных средств»

на тему «Создание простого Web-приложения»

Выполнили: студенты группы 22ВВП1

Беляев Д. И.

Демин М. С.

Приняли:

Патунин Д. В.

Деев М. В.

Пенза 2025

**Цель работы**

Создание простого Web-приложения

**Задание**

1. Реализовать класс DataBase, который будет отвечать за взаимодействие с базой данных. В качестве базы данных использовать SQLite. Все запросы SQL должны храниться в виде константных строк с заданным шаблоном. При формировании запроса использовать форматирование и интерполяцию.

2. Реализовать следующие запросы:

• Запрос CREATE TABLE IF NOT EXISTS;

• CREATE Index к поле ID таблицы Products;

• Запрос Select к таблице Products;

• Запрос Insert к таблице Products. При вставке данных в таблицу необходимо использовать связывание переменных;

• Запрос Update к таблице Products;

• Запрос Delete к таблице Products;

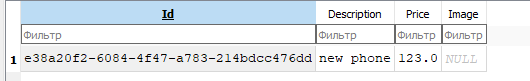
3. Написать тесты для осуществления модульного тестирования пункта №2. Для каждого запросы должны быть написаны тесты.

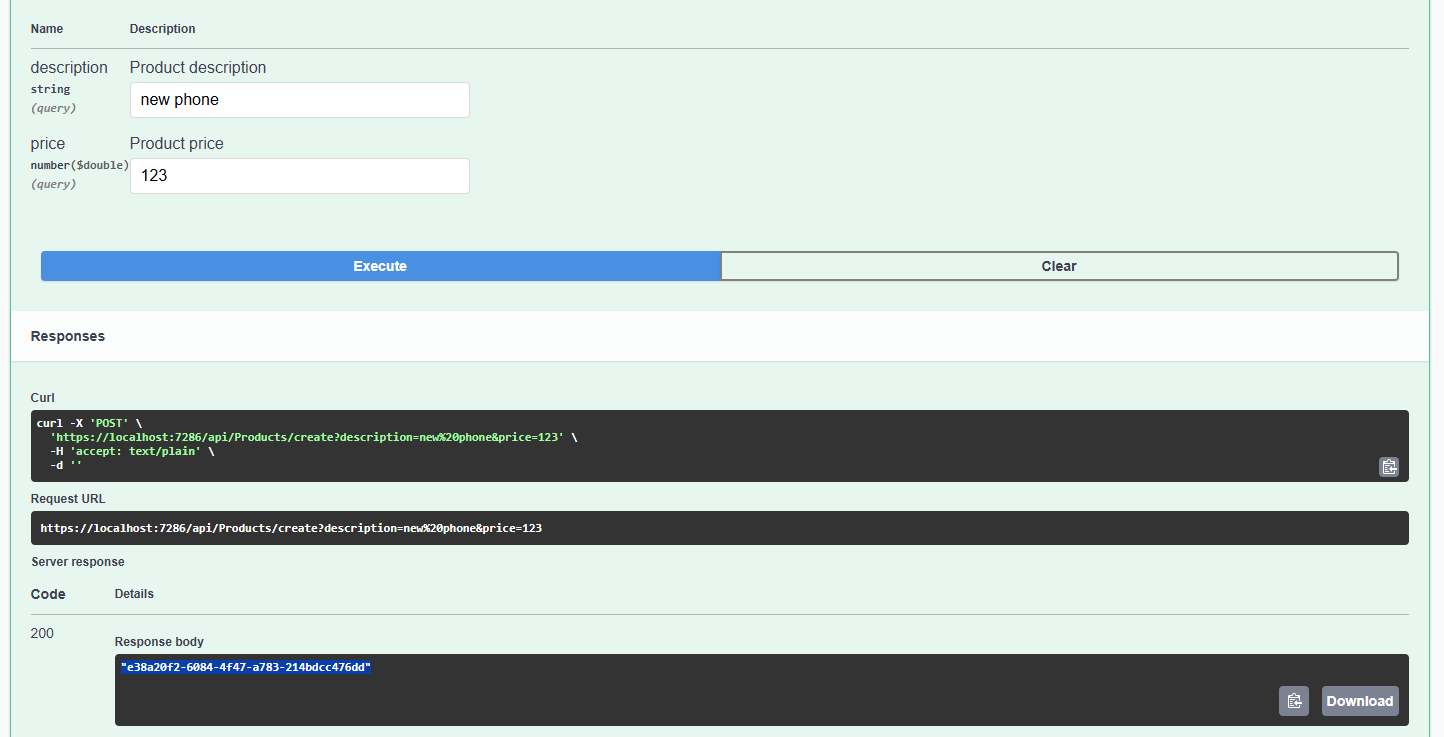
4. С помощью внедрения зависимостей внедрить класс DataBase в класс ProductService. DataBase должен использоваться вместо текущего места хранения продуктов (файла или оперативной памяти).

5. Написать тесты для осуществления интеграционного тестирования.

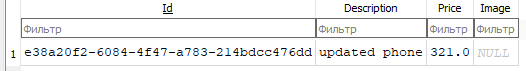
**Ход работы**

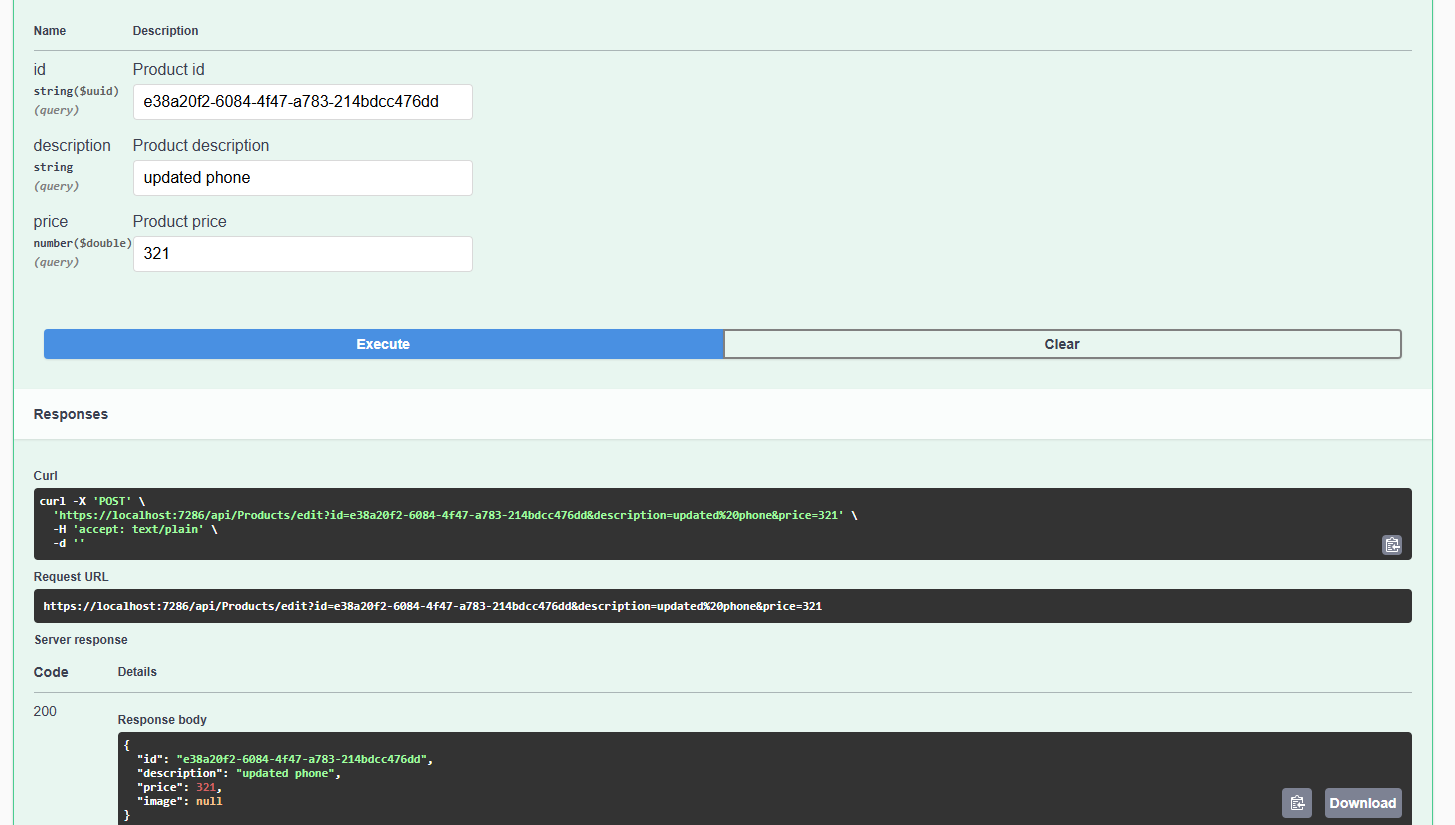
Добавление данных в таблицу



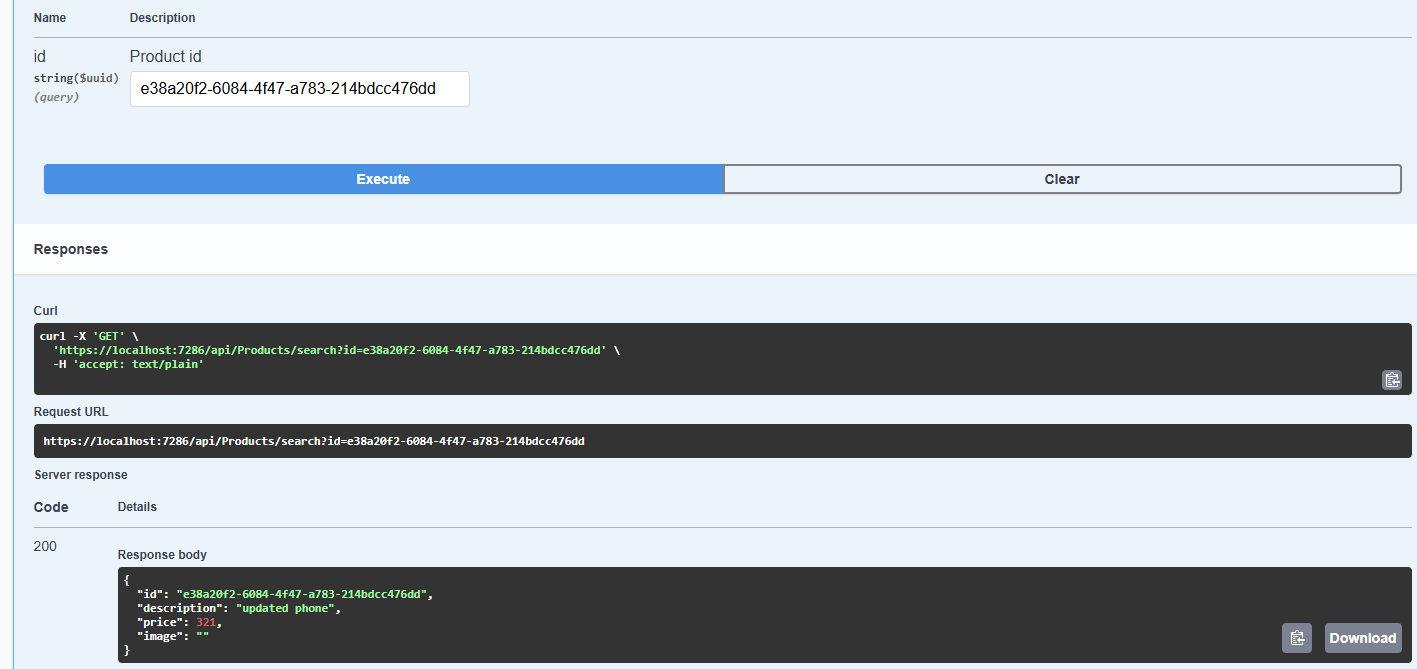


Обновление данных в таблице

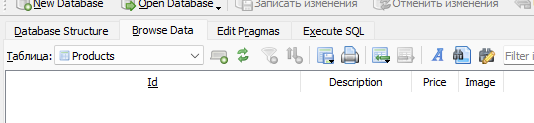


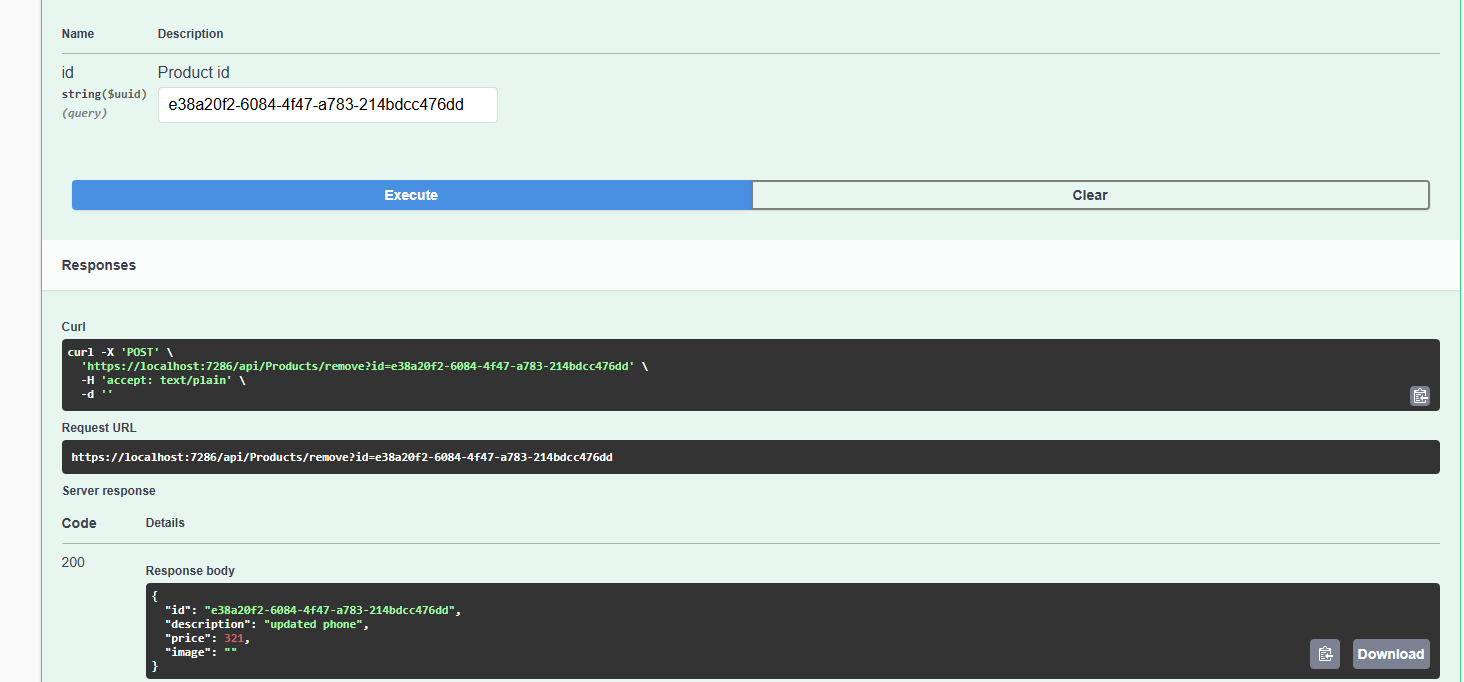


Поиск данных в таблице



Удаление данных из таблицы





**Листинг**

{

"Logging": {

"LogLevel": {

"Default": "Information",

"Microsoft.AspNetCore": "Warning"

}

},

"AllowedHosts": "\*",

"DataBaseFilePath": "Resources/database.txt",

"ConnectionStrings": {

"DefaultConnection": "Data Source=shopapp.db"

}

}

using ShopApp.Core.Data;

using ShopApp.Core.Models;

using ShopApp.Core.Services;

using ShopApp.WebApi.Data;

using ShopApp.WebApi.Services;

WebApplicationBuilder builder = WebApplication.CreateBuilder(args);

string connectionString = builder.Configuration.GetConnectionString("DefaultConnection")

?? "Data Source=shopapp.db";

builder.Services.AddSingleton<IDataBase>(new SqliteDataBase(connectionString));

builder.Services.AddSingleton<IProductsService<Product>, ProductsService>();

builder.Services.AddControllers();

builder.Services.AddEndpointsApiExplorer();

builder.Services.AddSwaggerGen(options =>

{

string basePath = AppContext.BaseDirectory;

string xmlPath = Path.Combine(basePath, "ShopApp.WebApi.xml");

options.IncludeXmlComments(xmlPath);

});

WebApplication app = builder.Build();

// Configure the HTTP request pipeline.

if (app.Environment.IsDevelopment())

{

\_ = app.UseSwagger();

\_ = app.UseSwaggerUI();

}

app.UseHttpsRedirection();

app.UseAuthorization();

app.MapControllers();

app.MapControllerRoute(

name: "default",

pattern: "{controller=Home}/{action=Index}");

app.Run();

using ShopApp.Core.Data;

using ShopApp.Core.Models;

using ShopApp.Core.Services;

namespace ShopApp.WebApi.Services

{

/// <summary>

/// Provides operations for managing products using a database.

/// </summary>

public class ProductsService : IProductsService<Product>, IDisposable

{

private readonly IDataBase \_dataBase;

/// <summary>

/// Initializes a new instance of the <see cref="ProductsService"/> class.

/// </summary>

/// <param name="dataBase">The database instance to be used for product operations.</param>

public ProductsService(IDataBase dataBase)

{

\_dataBase = dataBase ?? throw new ArgumentNullException(nameof(dataBase));

Task.Run(async () =>

{

await \_dataBase.CreateTableAsync();

await \_dataBase.CreateIndexAsync();

}).Wait();

}

/// <summary>

/// Performs application-defined tasks associated with freeing, releasing, or resetting unmanaged resources.

/// </summary>

public void Dispose()

{

GC.SuppressFinalize(this);

}

/// <summary>

/// Adds a new product.

/// </summary>

/// <param name="product">The product to add.</param>

/// <returns>A task that represents the asynchronous add operation. The task result contains true if successful; otherwise, false.</returns>

public async Task<bool> Add(Product product)

{

try

{

await \_dataBase.InsertProductAsync(product);

return true;

}

catch (Exception)

{

return false;

}

}

/// <summary>

/// Updates an existing product.

/// </summary>

/// <param name="product">The product with updated data.</param>

/// <returns>A task that represents the asynchronous update operation. The task result contains the updated product if successful; otherwise, null.</returns>

public async Task<Product?> Edit(Product product)

{

try

{

await \_dataBase.UpdateProductAsync(product);

return product;

}

catch (Exception)

{

return null;

}

}

/// <summary>

/// Removes a product by its unique identifier.

/// </summary>

/// <param name="productId">The unique identifier of the product to remove.</param>

/// <returns>A task that represents the asynchronous remove operation. The task result contains the removed product if successful; otherwise, null.</returns>

public async Task<Product?> Remove(Guid productId)

{

try

{

// Get the product from the database before deleting.

Product? product = await \_dataBase.SelectProductByIdAsync(productId);

if (product != null)

{

await \_dataBase.DeleteProductAsync(productId);

}

return product;

}

catch (Exception)

{

return null;

}

}

/// <summary>

/// Searches for a product by its unique identifier.

/// </summary>

/// <param name="productId">The unique identifier of the product.</param>

/// <returns>A task that represents the asynchronous search operation. The task result contains the product if found; otherwise, null.</returns>

public async Task<Product?> Search(Guid productId)

{

try

{

return await \_dataBase.SelectProductByIdAsync(productId);

}

catch (Exception)

{

return null;

}

}

/// <summary>

/// Retrieves all products.

/// </summary>

/// <returns>A task that represents the asynchronous get operation. The task result contains an enumerable of <see cref="Product"/> objects.</returns>

public async Task<IEnumerable<Product>> GetAll()

{

try

{

return await \_dataBase.SelectProductsAsync();

}

catch (Exception)

{

return [];

}

}

}

}

using Microsoft.Data.Sqlite;

using ShopApp.Core.Data;

using ShopApp.Core.Models;

namespace ShopApp.WebApi.Data

{

/// <summary>

/// Provides methods to interact with the SQLite database.

/// </summary>

public class SqliteDataBase : IDataBase

{

private const string SQL\_CREATE\_TABLE\_PRODUCTS = @"

CREATE TABLE IF NOT EXISTS Products (

Id TEXT PRIMARY KEY,

Description TEXT,

Price REAL,

Image TEXT NULL

);";

private const string SQL\_CREATE\_INDEX\_PRODUCTS\_ID = @"

CREATE INDEX IF NOT EXISTS IDX\_PRODUCTS\_ID

ON Products(Id);";

private const string SQL\_SELECT\_ALL\_PRODUCTS = "SELECT Id, Description, Price, Image FROM Products;";

private const string SQL\_SELECT\_PRODUCT\_BY\_ID = "SELECT Id, Description, Price, Image FROM Products WHERE Id = @Id;";

private const string SQL\_INSERT\_PRODUCT = @"

INSERT INTO Products (Id, Description, Price, Image)

VALUES (@Id, @Description, @Price, @Image);";

private const string SQL\_UPDATE\_PRODUCT = @"

UPDATE Products

SET Description = @Description, Price = @Price, Image = @Image

WHERE Id = @Id;";

private const string SQL\_DELETE\_PRODUCT = "DELETE FROM Products WHERE Id = @Id;";

private readonly string \_connectionString;

/// <summary>

/// Initializes a new instance of the <see cref="SqliteDataBase"/> class.

/// </summary>

/// <param name="connectionString">The SQLite connection string.</param>

public SqliteDataBase(string connectionString)

{

\_connectionString = connectionString;

}

/// <summary>

/// Asynchronously creates the Products table if it does not exist.

/// </summary>

public async Task CreateTableAsync()

{

using SqliteConnection connection = new(\_connectionString);

await connection.OpenAsync();

using SqliteCommand cmd = connection.CreateCommand();

cmd.CommandText = SQL\_CREATE\_TABLE\_PRODUCTS;

\_ = await cmd.ExecuteNonQueryAsync();

}

/// <summary>

/// Asynchronously creates an index on the Id field of the Products table.

/// </summary>

public async Task CreateIndexAsync()

{

using SqliteConnection connection = new(\_connectionString);

await connection.OpenAsync();

using SqliteCommand cmd = connection.CreateCommand();

cmd.CommandText = SQL\_CREATE\_INDEX\_PRODUCTS\_ID;

\_ = await cmd.ExecuteNonQueryAsync();

}

/// <summary>

/// Asynchronously retrieves all products from the Products table.

/// </summary>

/// <returns>An enumerable collection of <see cref="Product"/> objects.</returns>

public async Task<IEnumerable<Product>> SelectProductsAsync()

{

List<Product> products = [];

using SqliteConnection connection = new(\_connectionString);

await connection.OpenAsync();

using SqliteCommand cmd = connection.CreateCommand();

cmd.CommandText = SQL\_SELECT\_ALL\_PRODUCTS;

using SqliteDataReader reader = await cmd.ExecuteReaderAsync();

while (await reader.ReadAsync())

{

string? idStr = reader["Id"]?.ToString();

if (string.IsNullOrWhiteSpace(idStr) || !Guid.TryParse(idStr, out Guid parsedId))

{

continue;

}

products.Add(new Product

{

Id = parsedId,

Description = reader["Description"]?.ToString(),

Price = Convert.ToDouble(reader["Price"]),

Image = reader["Image"]?.ToString()

});

}

return products;

}

/// <summary>

/// Asynchronously retrieves a single product by its Id.

/// </summary>

/// <param name="productId">The unique identifier of the product.</param>

/// <returns>A <see cref="Product"/> object if found; otherwise, null.</returns>

public async Task<Product?> SelectProductByIdAsync(Guid productId)

{

using SqliteConnection connection = new(\_connectionString);

await connection.OpenAsync();

using SqliteCommand cmd = connection.CreateCommand();

cmd.CommandText = SQL\_SELECT\_PRODUCT\_BY\_ID;

\_ = cmd.Parameters.AddWithValue("@Id", productId.ToString());

using SqliteDataReader reader = await cmd.ExecuteReaderAsync();

if (await reader.ReadAsync())

{

string? idStr = reader["Id"]?.ToString();

return string.IsNullOrWhiteSpace(idStr) || !Guid.TryParse(idStr, out Guid parsedId)

? null

: new Product

{

Id = parsedId,

Description = reader["Description"]?.ToString(),

Price = Convert.ToDouble(reader["Price"]),

Image = reader["Image"]?.ToString()

};

}

return null;

}

/// <summary>

/// Asynchronously inserts a new product into the Products table.

/// </summary>

/// <param name="product">The <see cref="Product"/> to insert.</param>

public async Task InsertProductAsync(Product product)

{

using SqliteConnection connection = new(\_connectionString);

await connection.OpenAsync();

using SqliteCommand cmd = connection.CreateCommand();

cmd.CommandText = SQL\_INSERT\_PRODUCT;

\_ = cmd.Parameters.AddWithValue("@Id", product.Id.ToString());

\_ = cmd.Parameters.AddWithValue("@Description", product.Description);

\_ = cmd.Parameters.AddWithValue("@Price", product.Price);

\_ = cmd.Parameters.AddWithValue("@Image", product.Image ?? (object)DBNull.Value);

\_ = await cmd.ExecuteNonQueryAsync();

}

/// <summary>

/// Asynchronously updates an existing product in the Products table.

/// </summary>

/// <param name="product">The <see cref="Product"/> with updated data.</param>

public async Task UpdateProductAsync(Product product)

{

using SqliteConnection connection = new(\_connectionString);

await connection.OpenAsync();

using SqliteCommand cmd = connection.CreateCommand();

cmd.CommandText = SQL\_UPDATE\_PRODUCT;

\_ = cmd.Parameters.AddWithValue("@Id", product.Id.ToString());

\_ = cmd.Parameters.AddWithValue("@Description", product.Description);

\_ = cmd.Parameters.AddWithValue("@Price", product.Price);

\_ = cmd.Parameters.AddWithValue("@Image", product.Image ?? (object)DBNull.Value);

\_ = await cmd.ExecuteNonQueryAsync();

}

/// <summary>

/// Asynchronously deletes a product from the Products table by its Id.

/// </summary>

/// <param name="productId">The unique identifier of the product to delete.</param>

public async Task DeleteProductAsync(Guid productId)

{

using SqliteConnection connection = new(\_connectionString);

await connection.OpenAsync();

using SqliteCommand cmd = connection.CreateCommand();

cmd.CommandText = SQL\_DELETE\_PRODUCT;

\_ = cmd.Parameters.AddWithValue("@Id", productId.ToString());

\_ = await cmd.ExecuteNonQueryAsync();

}

}

}

using ShopApp.Core.Models;

namespace ShopApp.Core.Data

{

/// <summary>

/// Defines the methods required for interacting with the database.

/// </summary>

public interface IDataBase

{

/// <summary>

/// Creates the Products table if it does not exist.

/// </summary>

Task CreateTableAsync();

/// <summary>

/// Creates an index on the Id field of the Products table.

/// </summary>

Task CreateIndexAsync();

/// <summary>

/// Retrieves all products from the Products table.

/// </summary>

/// <returns>An enumerable collection of <see cref="Product"/> objects.</returns>

Task<IEnumerable<Product>> SelectProductsAsync();

/// <summary>

/// Retrieves a single product by its Id.

/// </summary>

/// <param name="productId">The unique identifier of the product.</param>

/// <returns>A <see cref="Product"/> object if found; otherwise, null.</returns>

Task<Product?> SelectProductByIdAsync(Guid productId);

/// <summary>

/// Inserts a new product into the Products table.

/// </summary>

/// <param name="product">The <see cref="Product"/> to insert.</param>

Task InsertProductAsync(Product product);

/// <summary>

/// Updates an existing product in the Products table.

/// </summary>

/// <param name="product">The <see cref="Product"/> with updated data.</param>

Task UpdateProductAsync(Product product);

/// <summary>

/// Deletes a product from the Products table by its Id.

/// </summary>

/// <param name="productId">The unique identifier of the product to delete.</param>

Task DeleteProductAsync(Guid productId);

}

}

using Moq;

using ShopApp.Core.Data;

using ShopApp.Core.Models;

using ShopApp.WebApi.Services;

namespace ShopApp.Tests

{

[TestFixture]

public class TestsProductsService

{

private Mock<IDataBase> \_mockDataBase;

private ProductsService \_service;

[SetUp]

public void SetUp()

{

\_mockDataBase = new Mock<IDataBase>();

// For initialization, simulate successful table/index creation.

\_ = \_mockDataBase.Setup(db => db.CreateTableAsync())

.Returns(Task.CompletedTask);

\_ = \_mockDataBase.Setup(db => db.CreateIndexAsync())

.Returns(Task.CompletedTask);

\_service = new ProductsService(\_mockDataBase.Object);

}

[TearDown]

public void TearDown()

{

\_service?.Dispose();

}

[TestCase("Test Product", 19.99, null)]

public async Task Add\_WhenInsertSucceeds\_ReturnsTrue(string description, double price, string? image)

{

// Arrange

Product product = new()

{

Id = Guid.NewGuid(),

Description = description,

Price = price,

Image = image

};

\_ = \_mockDataBase.Setup(db => db.InsertProductAsync(product))

.Returns(Task.CompletedTask);

// Act

bool result = await \_service.Add(product);

// Assert

Assert.That(result, Is.True, "Service.Add should return true when InsertProductAsync succeeds.");

\_mockDataBase.Verify(db => db.InsertProductAsync(It.Is<Product>(p => p.Id == product.Id)), Times.Once);

}

[TestCase("Test Product", 19.99, null)]

public async Task Add\_WhenInsertFails\_ReturnsFalse(string description, double price, string? image)

{

// Arrange

Product product = new()

{

Id = Guid.NewGuid(),

Description = description,

Price = price,

Image = image

};

\_ = \_mockDataBase.Setup(db => db.InsertProductAsync(product))

.ThrowsAsync(new Exception("Insert error"));

// Act

bool result = await \_service.Add(product);

// Assert

Assert.That(result, Is.False, "Service.Add should return false when InsertProductAsync fails.");

}

[TestCase("Test Product", 19.99, null)]

public async Task Edit\_WhenUpdateSucceeds\_ReturnsProduct(string description, double price, string? image)

{

// Arrange

Product product = new()

{

Id = Guid.NewGuid(),

Description = description,

Price = price,

Image = image

};

\_ = \_mockDataBase.Setup(db => db.UpdateProductAsync(product))

.Returns(Task.CompletedTask);

\_ = \_mockDataBase.Setup(db => db.SelectProductByIdAsync(product.Id))

.ReturnsAsync(product);

// Act

Product? result = await \_service.Edit(product);

// Assert

Assert.That(result, Is.Not.Null, "Service.Edit should return the product when update succeeds.");

Assert.That(result!.Id, Is.EqualTo(product.Id));

\_mockDataBase.Verify(db => db.UpdateProductAsync(It.Is<Product>(p => p.Id == product.Id)), Times.Once);

}

[TestCase("Test Product", 19.99, null)]

public async Task Edit\_WhenUpdateFails\_ReturnsNull(string description, double price, string? image)

{

// Arrange

Product product = new()

{

Id = Guid.NewGuid(),

Description = description,

Price = price,

Image = image

};

\_ = \_mockDataBase.Setup(db => db.UpdateProductAsync(product))

.ThrowsAsync(new Exception("Update error"));

// Act

Product? result = await \_service.Edit(product);

// Assert

Assert.That(result, Is.Null, "Service.Edit should return null when update fails.");

}

[TestCase("Test Product", 19.99, null)]

public async Task Remove\_WhenProductExists\_ReturnsProduct(string description, double price, string? image)

{

// Arrange

Product product = new()

{

Id = Guid.NewGuid(),

Description = description,

Price = price,

Image = image

};

\_ = \_mockDataBase.Setup(db => db.SelectProductByIdAsync(product.Id))

.ReturnsAsync(product);

\_ = \_mockDataBase.Setup(db => db.DeleteProductAsync(product.Id))

.Returns(Task.CompletedTask);

// Act

Product? result = await \_service.Remove(product.Id);

// Assert

Assert.That(result, Is.Not.Null, "Service.Remove should return the product when it exists.");

Assert.That(result!.Id, Is.EqualTo(product.Id));

\_mockDataBase.Verify(db => db.DeleteProductAsync(product.Id), Times.Once);

}

[Test]

public async Task Remove\_WhenProductDoesNotExist\_ReturnsNull()

{

// Arrange

Guid productId = Guid.NewGuid();

\_ = \_mockDataBase.Setup(db => db.SelectProductByIdAsync(productId))

.ReturnsAsync((Product?)null);

// Act

Product? result = await \_service.Remove(productId);

// Assert

Assert.That(result, Is.Null, "Service.Remove should return null if the product is not found.");

}

[TestCase("Test Product", 19.99, null)]

public async Task Search\_WhenProductExists\_ReturnsProduct(string description, double price, string? image)

{

// Arrange

Product product = new()

{

Id = Guid.NewGuid(),

Description = description,

Price = price,

Image = image

};

\_ = \_mockDataBase.Setup(db => db.SelectProductByIdAsync(product.Id))

.ReturnsAsync(product);

// Act

Product? result = await \_service.Search(product.Id);

// Assert

Assert.That(result, Is.Not.Null, "Service.Search should return the product if it exists.");

Assert.That(result!.Id, Is.EqualTo(product.Id));

}

[Test]

public async Task Search\_WhenProductDoesNotExist\_ReturnsNull()

{

// Arrange

Guid productId = Guid.NewGuid();

\_ = \_mockDataBase.Setup(db => db.SelectProductByIdAsync(productId))

.ReturnsAsync((Product?)null);

// Act

Product? result = await \_service.Search(productId);

// Assert

Assert.That(result, Is.Null, "Service.Search should return null if the product is not found.");

}

[TestCase(

"Original Description", 10.0, null,

"Updated Description", 15.5, null)]

public async Task GetAll\_ReturnsAllProducts(

string firstDescription, double firstPrice, string? firstImage,

string secondDescription, double secondPrice, string? secondImage)

{

// Arrange

List<Product> products =

[

new Product

{

Id = Guid.NewGuid(),

Description = firstDescription,

Price = firstPrice,

Image = firstImage

},

new Product

{

Id = Guid.NewGuid(),

Description = secondDescription,

Price = secondPrice,

Image = secondImage

}

];

\_ = \_mockDataBase.Setup(db => db.SelectProductsAsync())

.ReturnsAsync(products);

// Act

IEnumerable<Product> result = await \_service.GetAll();

// Assert

Assert.That(result, Is.Not.Null, "Service.GetAll should not return null.");

Assert.That(result.Count(), Is.EqualTo(2), "Service.GetAll should return all products.");

}

}

}

using Microsoft.Extensions.Configuration;

using Microsoft.Extensions.Logging.Abstractions;

using ShopApp.Core.Data;

using ShopApp.Core.Models;

using ShopApp.WebApi.Controllers;

using ShopApp.WebApi.Data;

using ShopApp.WebApi.Services;

namespace ShopApp.Tests

{

[TestFixture]

public class TestsProductsServiceController

{

private string \_tempDbPath;

private IConfiguration \_configuration;

private ProductsService \_service;

private ProductsController \_controller;

[SetUp]

public void SetUp()

{

// Generate a unique temporary database file path.

\_tempDbPath = Path.Combine(Path.GetTempPath(), $"{Guid.NewGuid()}.db");

// Build configuration with the connection string.

Dictionary<string, string?> configDictionary = new()

{

{ "ConnectionStrings:DefaultConnection", $"Data Source={\_tempDbPath}" }

};

\_configuration = new ConfigurationBuilder()

.AddInMemoryCollection(configDictionary)

.Build();

// Instantiate the real ProductsService.

string connectionString = \_configuration.GetConnectionString("DefaultConnection")!;

IDataBase dataBase = new SqliteDataBase(connectionString);

\_service = new ProductsService(dataBase);

// Create the controller with a NullLogger.

\_controller = new ProductsController(NullLogger<ProductsController>.Instance, \_service);

}

[TearDown]

public void TearDown()

{

\_service.Dispose();

// Clean up the temporary database file.

if (File.Exists(\_tempDbPath))

{

const int maxRetries = 10;

const int delayMs = 100;

for (int retry = 0; retry < maxRetries; retry++)

{

try

{

File.Delete(\_tempDbPath);

break;

}

catch (IOException)

{

Thread.Sleep(delayMs);

}

}

}

}

[TestCase("Product A", 10.0)]

[TestCase("Product B", 20.0)]

public async Task CreateProduct\_ReturnsNonEmptyGuid(string description, double price)

{

// Act

Guid productId = await \_controller.CreateProduct(description, price);

// Assert

Assert.That(productId, Is.Not.EqualTo(Guid.Empty), "CreateProduct should return a non-empty Guid.");

// Verify that the product exists.

Product? product = await \_controller.SearchProduct(productId);

Assert.That(product, Is.Not.Null, "Product should be found after creation.");

Assert.That(product!.Description, Is.EqualTo(description), "Product description should match.");

Assert.That(product.Price, Is.EqualTo(price), "Product price should match.");

}

[TestCase("Original Product A", 10.0, "Updated Product A", 15.0)]

public async Task EditProduct\_ReturnsUpdatedProduct(string initialDesc, double initialPrice, string updatedDesc, double updatedPrice)

{

// Arrange

Guid productId = await \_controller.CreateProduct(initialDesc, initialPrice);

Assert.That(productId, Is.Not.EqualTo(Guid.Empty), "Product creation should return a valid Guid.");

// Act

Product? editedProduct = await \_controller.EditProduct(productId, updatedDesc, updatedPrice);

// Assert

Assert.That(editedProduct, Is.Not.Null, "EditProduct should return the updated product.");

Assert.That(editedProduct!.Description, Is.EqualTo(updatedDesc), "Product description should be updated.");

Assert.That(editedProduct.Price, Is.EqualTo(updatedPrice), "Product price should be updated.");

}

[TestCase("Product A", 10.0)]

public async Task RemoveProduct\_ReturnsRemovedProduct(string description, double price)

{

// Arrange

Guid productId = await \_controller.CreateProduct(description, price);

Assert.That(productId, Is.Not.EqualTo(Guid.Empty), "Product creation should return a valid Guid.");

// Act

Product? removedProduct = await \_controller.RemoveProduct(productId);

// Assert

Assert.That(removedProduct, Is.Not.Null, "RemoveProduct should return the removed product.");

Assert.That(removedProduct!.Id, Is.EqualTo(productId), "Removed product should have the correct ID.");

// Verify the product no longer exists.

Product? searchResult = await \_controller.SearchProduct(productId);

Assert.That(searchResult, Is.Null, "Product should not be found after removal.");

}

[TestCase("Product A", 10.0)]

public async Task SearchProduct\_ReturnsProduct(string description, double price)

{

// Arrange

Guid productId = await \_controller.CreateProduct(description, price);

Assert.That(productId, Is.Not.EqualTo(Guid.Empty), "Product creation should return a valid Guid.");

// Act

Product? foundProduct = await \_controller.SearchProduct(productId);

// Assert

Assert.That(foundProduct, Is.Not.Null, "SearchProduct should return the product if it exists.");

Assert.That(foundProduct!.Description, Is.EqualTo(description), "Product description should match.");

Assert.That(foundProduct.Price, Is.EqualTo(price), "Product price should match.");

}

}

}

using Microsoft.Data.Sqlite;

using ShopApp.Core.Data;

using ShopApp.Core.Models;

using ShopApp.WebApi.Data;

namespace ShopApp.Tests

{

[TestFixture]

public class TestsSqliteDataBase

{

private string \_dbFilePath;

private string \_connectionString;

private IDataBase \_dataBase;

[SetUp]

public async Task SetUp()

{

// Create a temporary file for the SQLite database.

\_dbFilePath = Path.Combine(Path.GetTempPath(), $"{Guid.NewGuid()}.db");

\_connectionString = $"Data Source={\_dbFilePath}";

\_dataBase = new SqliteDataBase(\_connectionString);

// Create table and index.

await \_dataBase.CreateTableAsync();

await \_dataBase.CreateIndexAsync();

}

[TearDown]

public void TearDown()

{

// Clean up the temporary database file.

if (File.Exists(\_dbFilePath))

{

const int maxRetries = 10;

const int delayMs = 100;

for (int retry = 0; retry < maxRetries; retry++)

{

try

{

File.Delete(\_dbFilePath);

break;

}

catch (IOException)

{

System.Threading.Thread.Sleep(delayMs);

}

}

}

}

[Test]

public async Task CreateTableAsync\_TableExistsInDatabase()

{

// Act: Query sqlite\_master to check for table existence.

bool tableExists;

using (SqliteConnection connection = new(\_connectionString))

{

await connection.OpenAsync();

using SqliteCommand cmd = connection.CreateCommand();

cmd.CommandText = "SELECT name FROM sqlite\_master WHERE type='table' AND name='Products';";

object? result = await cmd.ExecuteScalarAsync();

tableExists = result != null && result.ToString() == "Products";

}

// Assert

Assert.That(tableExists, Is.True, "The Products table should exist after CreateTableAsync is called.");

}

[Test]

public async Task CreateIndexAsync\_IndexExistsInDatabase()

{

// Act: Query sqlite\_master to check for index existence.

bool indexExists;

using (SqliteConnection connection = new(\_connectionString))

{

await connection.OpenAsync();

using SqliteCommand cmd = connection.CreateCommand();

cmd.CommandText = "SELECT name FROM sqlite\_master WHERE type='index' AND name='IDX\_PRODUCTS\_ID';";

object? result = await cmd.ExecuteScalarAsync();

indexExists = result != null && result.ToString() == "IDX\_PRODUCTS\_ID";

}

// Assert

Assert.That(indexExists, Is.True, "The IDX\_PRODUCTS\_ID index should exist after CreateIndexAsync is called.");

}

[TestCase("Test Product", 19.99, null)]

public async Task InsertProductAsync\_And\_SelectProductByIdAsync\_ReturnsInsertedProduct(string description, double price, string? image)

{

// Arrange

Product product = new()

{

Id = Guid.NewGuid(),

Description = description,

Price = price,

Image = image

};

// Act

await \_dataBase.InsertProductAsync(product);

Product? selected = await \_dataBase.SelectProductByIdAsync(product.Id);

// Assert

Assert.That(selected, Is.Not.Null, "Inserted product should be found by SelectProductByIdAsync.");

Assert.That(selected!.Id, Is.EqualTo(product.Id));

Assert.That(selected.Description, Is.EqualTo(product.Description));

Assert.That(selected.Price, Is.EqualTo(product.Price));

}

[TestCase(

"Original Description", 10.0, null,

"Updated Description", 15.5, null)]

public async Task UpdateProductAsync\_UpdatesExistingProduct(

string originalDescription, double originalPrice, string? originalImage,

string updatedDescription, double updatedPrice, string? updatedImage)

{

// Arrange

Product product = new()

{

Id = Guid.NewGuid(),

Description = originalDescription,

Price = originalPrice,

Image = originalImage

};

await \_dataBase.InsertProductAsync(product);

// Act

product.Description = updatedDescription;

product.Price = updatedPrice;

product.Image = updatedImage;

await \_dataBase.UpdateProductAsync(product);

Product? updated = await \_dataBase.SelectProductByIdAsync(product.Id);

// Assert

Assert.That(updated, Is.Not.Null, "Updated product should be found.");

Assert.That(updated!.Description, Is.EqualTo(updatedDescription));

Assert.That(updated.Price, Is.EqualTo(updatedPrice));

}

[TestCase("Test Product", 19.99, null)]

public async Task DeleteProductAsync\_RemovesProduct(string description, double price, string? image)

{

// Arrange

Product product = new()

{

Id = Guid.NewGuid(),

Description = description,

Price = price,

Image = image

};

await \_dataBase.InsertProductAsync(product);

// Act

await \_dataBase.DeleteProductAsync(product.Id);

Product? deleted = await \_dataBase.SelectProductByIdAsync(product.Id);

// Assert

Assert.That(deleted, Is.Null, "Product should be null after deletion.");

}

[TestCase(

"Original Description", 10.0, null,

"Updated Description", 15.5, null)]

public async Task SelectProductsAsync\_ReturnsAllInsertedProducts(

string firstDescription, double firstPrice, string? firstImage,

string secondDescription, double secondPrice, string? secondImage)

{

// Arrange

Product product1 = new()

{

Id = Guid.NewGuid(),

Description = firstDescription,

Price = firstPrice,

Image = firstImage

};

Product product2 = new()

{

Id = Guid.NewGuid(),

Description = secondDescription,

Price = secondPrice,

Image = secondImage

};

await \_dataBase.InsertProductAsync(product1);

await \_dataBase.InsertProductAsync(product2);

// Act

IEnumerable<Product> products = await \_dataBase.SelectProductsAsync();

List<Product> list = products.ToList();

// Assert

Assert.That(list.Any(p => p.Id == product1.Id), Is.True, "Product 1 should be in the result set.");

Assert.That(list.Any(p => p.Id == product2.Id), Is.True, "Product 2 should be in the result set.");

}

}

}

**Вывод**

Создали простое Web-приложение