

Exercise: Integration Testing of Web API Controllers

This document defines the exercises and homework assignments for the
"QA Back-End Test Automation" Course @ SoftUni.

Eventmi Home Explore ▾ Add ▾

EVENTMI

The place where events happen!

0. Preparing the Development Environment

Before diving into exploring the application and understanding its structure, it's crucial to ensure that your development environment is properly set up.

Connect Visual Studio with SSMS

Launch **SSMS** and connect to your **SQL Server** instance. Ensure that you have the necessary permissions to create and modify databases. Note down the connection details (server name, authentication method, etc.) for establishing a connection from **Visual Studio**.

Update the Database

Review the database configuration in the ASP.NET MVC project's **appsettings.json** file. Ensure that the **connection string** is correctly configured to connect to your **SQL Server** instance. Use **Entity Framework Core migrations** to **create or update the database schema**.

- Open **Package Manager Console** in Visual Studio (Tools > NuGet Package Manager > Package Manager Console)
- Run the following command to apply migrations and update the database: **Update-Database**

```
PM> update-database
Build started...
Build succeeded.
Microsoft.EntityFrameworkCore.Infrastructure[10403]
```

- This command will **apply any pending migrations and update the database** schema accordingly

The screenshot shows the Visual Studio interface. On the left, the 'Eventmi' project is expanded, showing 'Database Diagrams', 'Tables', 'System Tables', 'FileTables', 'External Tables', 'Graph Tables', 'dbo._EFMigrationsHistory', and 'dbo.Events'. On the right, the 'Results' pane displays a table with 5 rows and 5 columns: Id, Name, Start, End, and Place. The table contains event data.

	Id	Name	Start	End	Place
1	1	InnovateTech 2024	2024-06-15 09:00:00.0000000	2024-06-16 17:00:00.0000000	SofiaTechPark
2	2	DataCon 2024	2024-06-20 09:00:00.0000000	2024-06-22 18:00:00.0000000	CityTechHall
3	3	MedTech Summit 2024	2024-09-10 19:30:00.0000000	2024-09-10 22:00:00.0000000	MedicalHomePalace
4	4	ArtFest 2024	2024-08-22 12:00:00.0000000	2024-08-29 22:00:00.0000000	CentralSquare-OpenSpace
5	5	LaunchPad 2024	2024-10-10 22:00:00.0000000	2024-10-10 22:45:00.0000000	OnlineEventHappening

1. Discovering the Application

Before diving into testing with **RestSharp**, it's essential to understand the structure and **functionality** of the ASP.NET MVC project, particularly **focusing on the EventController** and its associated views. Here's what you need to know:

Purpose of the Application

The **Eventmi** application serves as a comprehensive platform designed to facilitate the management of events.

Understanding Controller Actions

In **ASP.NET MVC**, controllers play a central role in handling incoming **HTTP requests**, processing data, and generating responses. Controllers are responsible for **executing the application logic** and determining which **views** to render. Let's dive deeper into understanding controllers in the context of our Eventmi application.

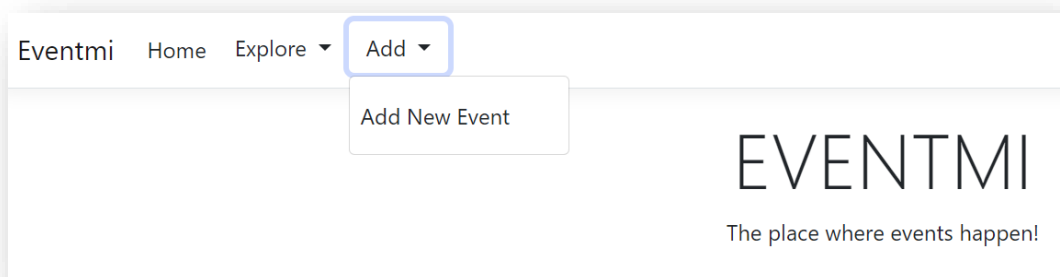
- **Definition:** A controller is a C# class that **inherits from the Controller base class** or **implements the IController** interface. It typically resides in the Controllers folder of the project
- **Responsibilities:**
 - Handle incoming HTTP requests from clients
 - Process and validate data
 - Interact with the model layer to retrieve or update data
 - Determine the appropriate view to render and pass data to the view
- **Actions:** An action is a public method within a controller that corresponds to a specific HTTP request. Each action is responsible for processing a particular request and generating an appropriate response

Functionality Overview

Review the functionalities provided by the **EventController**. This includes **adding, editing, deleting, and viewing** events.

➤ Adding Events

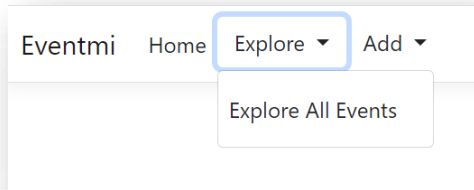
- The **Add action** within the EventController enables users to **add new events to the application**
- It **renders a view** where users can input details such as the event title, date, time, location
- Upon **form submission**, the controller **processes the input data** and **initiates the creation of a new event**

The screenshot displays the 'Add an Event' form. It contains input fields for 'Name' (with 'TechSummit-Reloaded' entered), 'Start' (11/21/2024 09:30 PM), 'End' (11/23/2024 12:00 AM), and 'Place' (Convention Center, Downtown Area). A blue 'Add' button is at the bottom.The screenshot shows a table titled 'Explore All Events' with a subtitle 'View All of the Events'. The table lists several events with their details and actions.

Event	Start	End	Place	Actions
InnovateTech 2024	6/15/2024 9:00:00 AM	6/16/2024 5:00:00 PM	SofiaTechPark	Details Edit Delete
DataCon 2024	6/20/2024 9:00:00 AM	6/22/2024 6:00:00 PM	CityTechHall	Details Edit Delete
MedTech Summit 2024	9/10/2024 7:30:00 PM	9/10/2024 10:00:00 PM	MedicalHomePalace	Details Edit Delete
ArtFest 2024	8/22/2024 12:00:00 PM	8/29/2024 10:00:00 PM	CentralSquare-OpenSpace	Details Edit Delete
LaunchPad 2024	10/10/2024 10:00:00 PM	10/10/2024 10:45:00 PM	OnlineEventHappening	Details Edit Delete
TechSummit-Reloaded	11/21/2024 3:30:00 PM	11/23/2024 12:00:00 AM	Convention Center, Downtown Area	Details Edit Delete

➤ Viewing All Events

- The **All action** retrieves and **displays a list of all events** currently available in the system
- Users can **access this list** to browse through the various events hosted on the platform
- Each **event listing typically includes essential details** such as the event name, start date and time, end date and time, and information about the location that will take place



Explore All Events				
View All of the Events				
Event	Start	End	Place	Actions
InnovateTech 2024	6/15/2024 9:00:00 AM	6/16/2024 5:00:00 PM	SofiaTechPark	Details Edit Delete
DataCon 2024	6/20/2024 9:00:00 AM	6/22/2024 6:00:00 PM	CityTechHall	Details Edit Delete
MedTech Summit 2024	9/10/2024 7:30:00 PM	9/10/2024 10:00:00 PM	MedicalHomePalace	Details Edit Delete
ArtFest 2024	8/22/2024 12:00:00 PM	8/29/2024 10:00:00 PM	CentralSquare-OpenSpace	Details Edit Delete
LaunchPad 2024	10/10/2024 10:00:00 PM	10/10/2024 10:45:00 PM	OnlineEventHappening	Details Edit Delete
TechSummit-Reloaded	11/21/2024 3:30:00 PM	11/23/2024 12:00:00 AM	Convention Center, Downtown Area	Details Edit Delete

➤ Viewing Event Details

- The **Details action** allows users to view detailed information about a specific event
- Users can **access the details page of an event** by clicking on the **Details button** or associated link
- The **controller retrieves the complete set of information** associated with the event and **renders it** on a dedicated details view

Event	Start	End	Place	Actions
InnovateTech 2024	6/15/2024 9:00:00 AM	6/16/2024 5:00:00 PM	SofiaTechPark	Details Edit Delete

Event Details

Event: InnovateTech 2024

Start: 6/15/2024 9:00:00 AM

End: 6/16/2024 5:00:00 PM

Place: SofiaTechPark

[Edit](#) [Delete](#)

➤ Editing Events

- The **Edit action** allows users to **modify existing events**
- Users can access an **edit form pre-populated with the current details** of the event they wish to modify
- After making the necessary **changes**, users **submit the form, triggering the controller to update the corresponding event** with the revised information

Edit an Event

Name

Start

End

Place

[Edit](#)

➤ Deleting Events

- With the **Delete** action, users can **remove events from the application**
- Typically, **this action is triggered by clicking a delete button or link associated with a specific event**
- The **controller initiates the deletion process**, permanently removing the event from the system

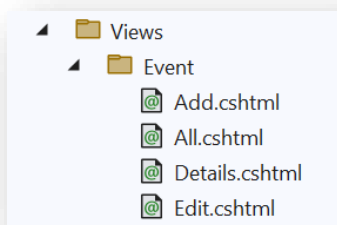
DataCon 2024	6/20/2024 9:00:00 AM	6/22/2024 6:00:00 PM	CityTechHall	Details	Edit	Delete
--------------	----------------------	----------------------	--------------	---------	------	--------

View Templates

Explore the **views associated with the EventController actions**. These views are **responsible for rendering HTML** content to the users.

In **ASP.NET MVC**, adopting consistent **naming conventions** for views helps maintain a clear and organized project structure. By following naming conventions, developers can easily locate and identify **views related to specific controllers or actions**.

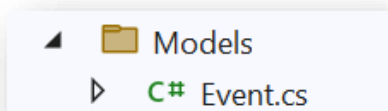
Views are typically **organized in folders** that **correspond to the controller names**. Within each controller folder, views are further **organized into subfolders** named after the **controller's actions**. This convention helps group related views together and makes it easier to locate views for specific controller actions.



Data Model

Examine the **data model associated with events**.

- What attributes does an event have?
- How is event data stored and retrieved?



```
7 references
public class Event
{
    [Key]
    2 references
    public int Id { get; set; }

    [Required]
    [MaxLength(EventNameMaxLength)]
    4 references
    public string Name { get; set; } = null!;

    [Required]
    4 references
    public DateTime Start { get; set; }

    [Required]
    4 references
    public DateTime End { get; set; }

    [Required]
    4 references
    public string Place { get; set; } = null!;
}
```

Dependencies

Identify any **external dependencies the controller relies on**, such as services or data access layers.

User Workflow

Understand the typical **user workflow within the application**.

- How do users interact with events?
- What actions can they perform?

By gaining a **comprehensive understanding of the application**, you'll be better equipped to design and execute **meaningful tests using RestSharp**. This knowledge will guide you in writing test cases that cover **critical aspects of the application's functionality**, ensuring its **reliability** and **correctness**.

The 'Add an Event' form includes a navigation bar with 'Eventmi', 'Home', 'Explore', and 'Add'. Below the navigation bar is a sub-header 'Add New Event'. The form fields are: Name (text input with 'Summer Merge'), Start (datetime picker with '07/28/2024 10:30 AM'), End (datetime picker with '07/28/2024 05:00 PM'), and Place (text input with 'Inter Expo Center'). An 'Add' button is at the bottom right.

The 'Explore All Events' page shows a table of events with columns: Event, Start, End, Place, and Actions. The Actions column contains 'Details', 'Edit', and 'Delete' buttons for each event.

Event	Start	End	Place	Actions
DataCon 2024	6/20/2024 9:00:00 AM	6/22/2024 6:00:00 PM	CityTechHall	Details Edit Delete
MedTech Summit 2024	9/10/2024 7:30:00 PM	9/10/2024 10:00:00 PM	MedicalHomePlace	Details Edit Delete
ArtFest 2024	8/22/2024 12:00:00 PM	8/28/2024 10:00:00 PM	CentralSquare-OpenSpace	Details Edit Delete
LaunchPad 2024	10/10/2024 10:00:00 PM	10/10/2024 10:45:00 PM	OnlineEventHappening	Details Edit Delete
Summer Merge	7/28/2024 10:30:00 AM	7/28/2024 5:00:00 PM	Inter Expo Center	Details Edit Delete

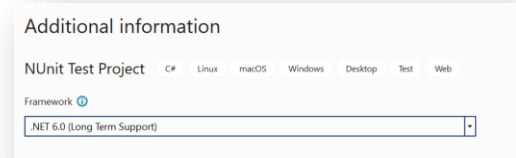
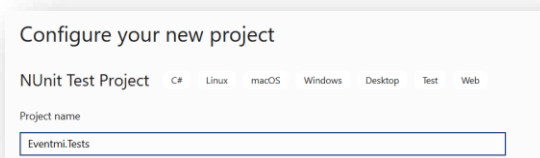
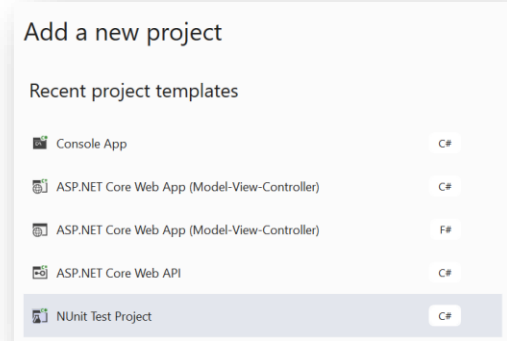
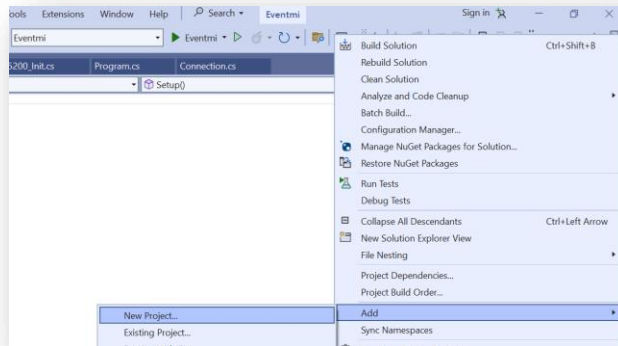
The 'Event Details' page displays the following information: Event: DataCon 2024, Start: 6/20/2024 9:00:00 AM, End: 6/22/2024 6:00:00 PM, and Place: CityTechHall. At the bottom are 'Edit' and 'Delete' buttons.

The 'Edit an Event' form shows the following fields: Name (DataCon 2024), Start (06/20/2024 09:00 AM), End (06/22/2024 06:00 PM), and Place (CityTechHall). An 'Edit' button is at the bottom right.

2. Setup RestSharp in your ASP.NET MVC project

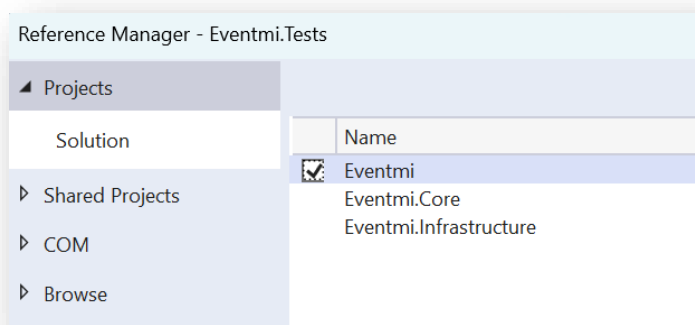
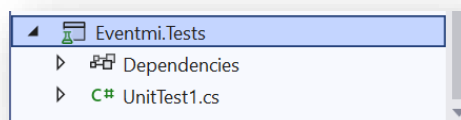
Create a New Test Project

- Right-click on your **solution** in Visual Studio
- Select **"Add" > "New Project"** from the context menu
- **Choose** the appropriate **project template** for your test project (e.g., **NUnit**)
- Ensure that the **.NET version matches your main ASP.NET MVC project** (e.g., .NET 6.0).
- Give your project a name (e.g., **Eventmi.Tests**) and click **"Create"**



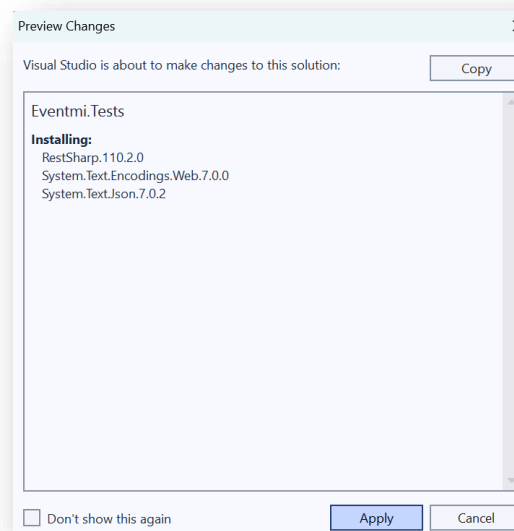
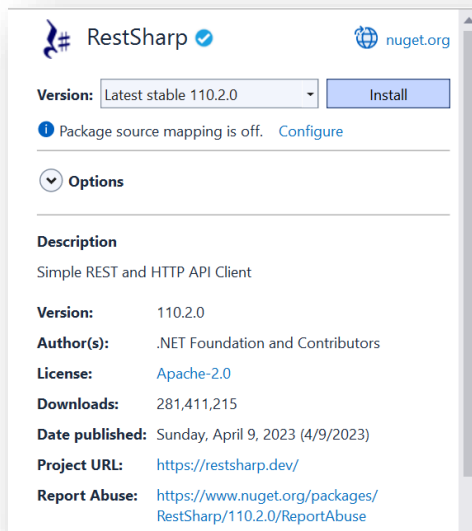
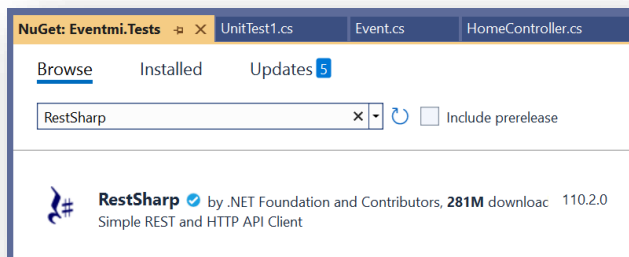
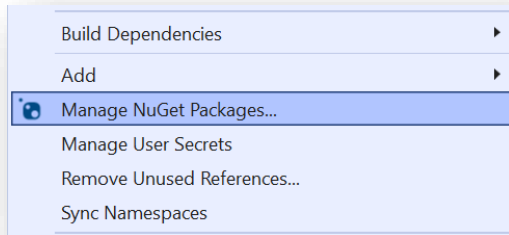
Add Project Reference

- Right-click on the test project in the Solution Explorer
- Select **"Add" > "Project Reference"** from the context menu
- In the **"Reference Manager"** window, select the **"Projects"** tab
- Check the checkbox next to the main ASP.NET MVC project to **add a reference to it**
- Click **"OK"** to confirm and add the reference



Add Dependencies

- Right-click on the test project in the Solution Explorer
- Select "**Manage NuGet Packages**" from the context menu
- Install the **RestSharp** package by searching for "RestSharp" and installing it



Why Adding Dependencies is Important

Adding dependencies, such as testing frameworks and RestSharp, ensures that **your test project has access to the required tools and libraries** for writing and executing tests effectively. These dependencies provide the necessary functionality to **simulate HTTP requests, assert behavior, and validate results** during testing.

Note:

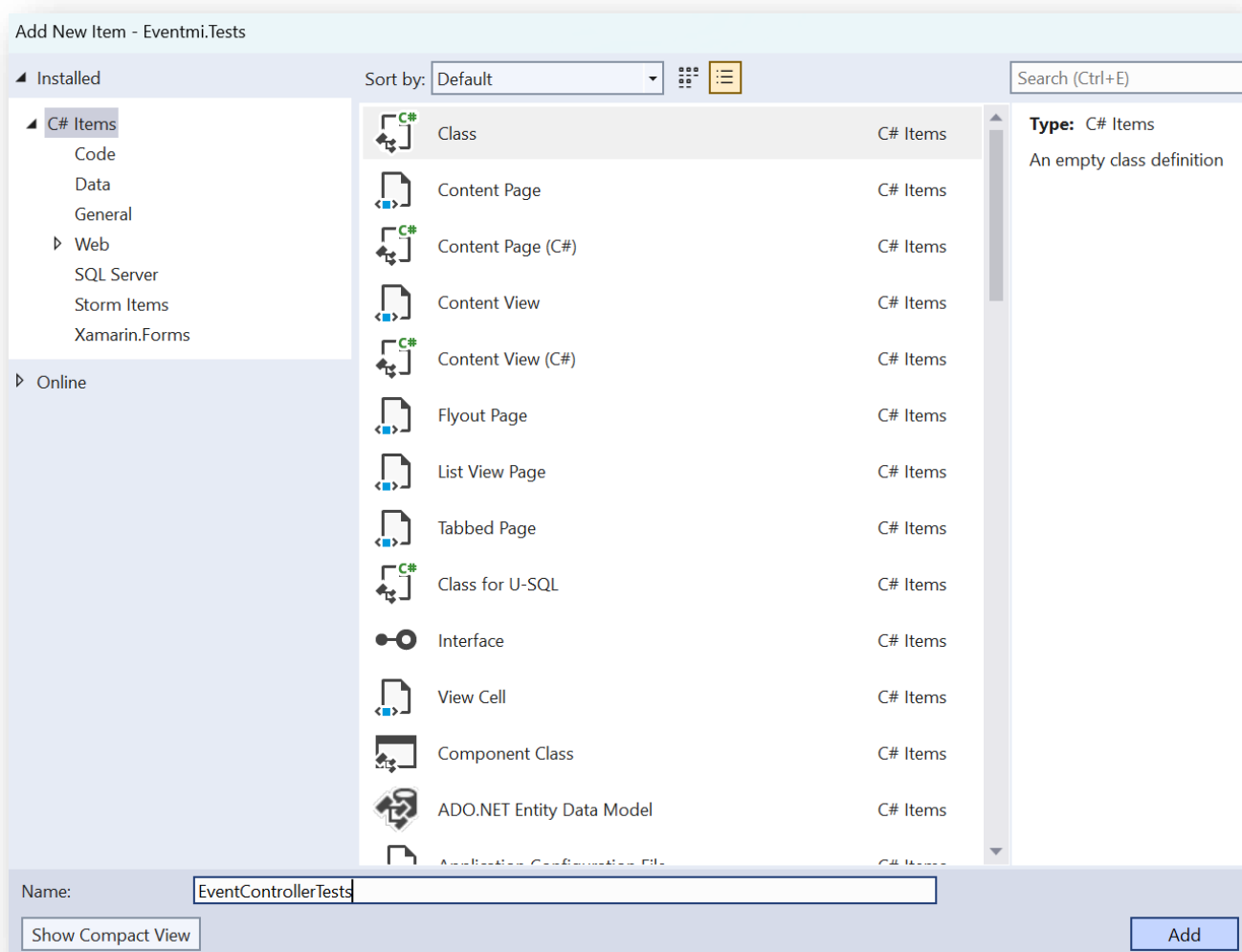
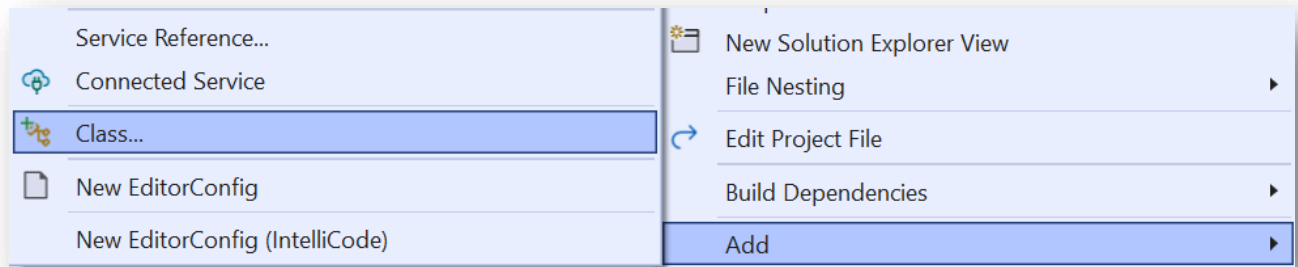
- Ensure that you install **compatible versions** of the dependencies that are **supported by your test framework** and target .NET version.
- **Keep dependencies up to date** by periodically checking for updates and installing the latest versions to leverage new features and improvements.
- By adding the necessary dependencies to your test project, you equip it with the **tools and libraries** required to write comprehensive tests for your ASP.NET MVC application, ensuring its functionality and reliability.

3. Writing Test Cases

- Open the test class **corresponding to your controller** in the test project
- Write test methods to **verify the behavior of the controller actions**
- In the test methods, **use RestSharp to simulate HTTP requests** to the controller and **validate the responses**

Create a New Class in the Test Project

- **Right-click** on the **test project** in the Solution Explorer
- Select **"Add" > "Class"** from the context menu
- **Name the class appropriately**, such as **EventControllerTests.cs**
- Click **"Add"** to create the class



AAA Pattern

In the newly created class (**EventControllerTests.cs**), you'll write test methods to **verify the behavior of the EventController actions**. Each test method should follow the **Arrange-Act-Assert** pattern

- **Arrange:** Set up any necessary pieces before executing the code you're testing. This could include creating mock data, **configuring a RestSharp client**, and setting up request objects
- **Act:** Execute the action that you're testing. This will typically be a **call to the controller action via RestSharp**
- **Assert:** Check that the action has had the expected outcome. This can include **verifying the status code, the data returned**, and that **certain methods were called**

Initialize Reusable Fields

private RestClient client:

- This field holds an **instance of the RestClient** which is part of the RestSharp library
- The **RestClient** is responsible for sending HTTP requests and receiving responses from your API
- It is **declared as private** because it **should only be used within the EventControllerTests** class and not be accessible from outside

```
using RestSharp;

namespace Eventmi.Tests
{
    [TestFixture]
    0 references
    public class EventControllerTests
    {
        private RestClient _client;
        private const string baseUrl = @"https://localhost:7236";
    }
}
```

baseUrl:

- In the context of testing an API using RestSharp refers to the **root address where the API is accessible**
- When you instantiate a **RestClient** object, you provide it with this base URL, and it will be used as the **starting point for all requests** made by that client
- If your API is running locally and **listening on port 7236**, your base URL might be **"https://localhost:7236"**



[SetUp] Method

Instead of a constructor, the **[SetUp]** method is used in NUnit to prepare the test environment before each test is run.

- **RestClient Initialization:** The `_client` field is assigned a new instance of `RestClient`, which is initialized with the `baseURL` field. This `RestClient` instance will be used to send HTTP requests to your API

```
[SetUp]
0 references
public void Setup()
{
    _client = new RestClient(baseURL);
}
```

Why Not Use a Constructor?

While you could theoretically use a constructor in a test class to initialize fields, it's not the standard practice for a few reasons:

- **Consistency:** NUnit's setup and teardown methods (**[SetUp]** and **[TearDown]**) provide a consistent way to initialize and **clean up resources before and after each test**. They're designed to work with NUnit's test lifecycle
- **Flexibility:** Using **[SetUp]** allows for more complex setup logic that might involve reading from configuration files, setting up mocks, or any other actions you'd want to perform before each test.
- **Isolation:** **[SetUp]** ensures that the state is reset before each test, providing better test isolation and reducing the risk of inter-test dependencies.
- By following the **[SetUp]** convention, your tests become more predictable and easier to understand for anyone familiar with NUnit.

Write Test Methods

```
[Test]
0 references
public void GetAllEvents_ReturnsSuccessSatusCode()
```

The name of the test method, `GetAllEvents_ReturnsSuccessStatusCode`, is designed to be **self-explanatory**, indicating both **the action being tested** and **the expected outcome**.

Breaking down the name:

- **GetAllEvents:** This suggests that the **test is focused on the functionality that retrieves all events**. It implies a **specific action or endpoint in your API** that is **responsible for returning a collection of event objects**
- **ReturnsSuccessStatusCode:** This part of the name specifies the **expected result of the test**. It indicates that, **upon invoking the GetAllEvents action, the API should respond with a success status code**, which is typically **200 OK** in HTTP terms. This status code signifies that the request was successfully processed

Arrange

- In the arrange section, you prepare the necessary setup for executing the test.
- **RestRequest object is created** with the endpoint **/Event/All** and the **HTTP method GET**.
- This setup involves specifying **which API endpoint you're testing** and **the type of HTTP request made**.

```
//Arrange: Prepare the HTTP request to the All action
var request = new RestRequest("/Event/All", Method.Get);
```

Act

- The act section is **where the action being tested is performed**.
- In this case, it's the execution of the **HTTP request against the specified endpoint**.
- By executing the request, you're simulating what would happen when this **endpoint is called under normal circumstances**.

```
//Act: Execute HTTP request
var response = _client.Execute(request);
```

Assert

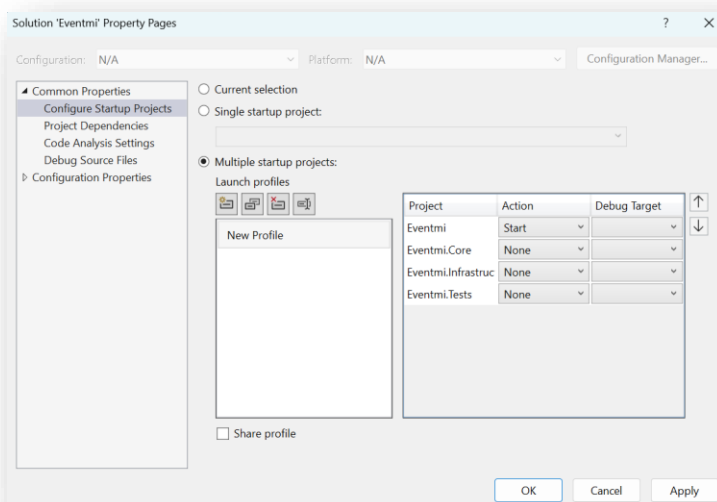
- The assert section **verifies** that the **outcome of the action meets the expectations**.
- For this test, the **expectation is that the API will return a 200 OK status code**, indicating success.

```
//Assert: Verify that the response status code is OK (200)
Assert.That(response.StatusCode, Is.EqualTo(System.Net.HttpStatusCode.OK));
```

4. Running Tests

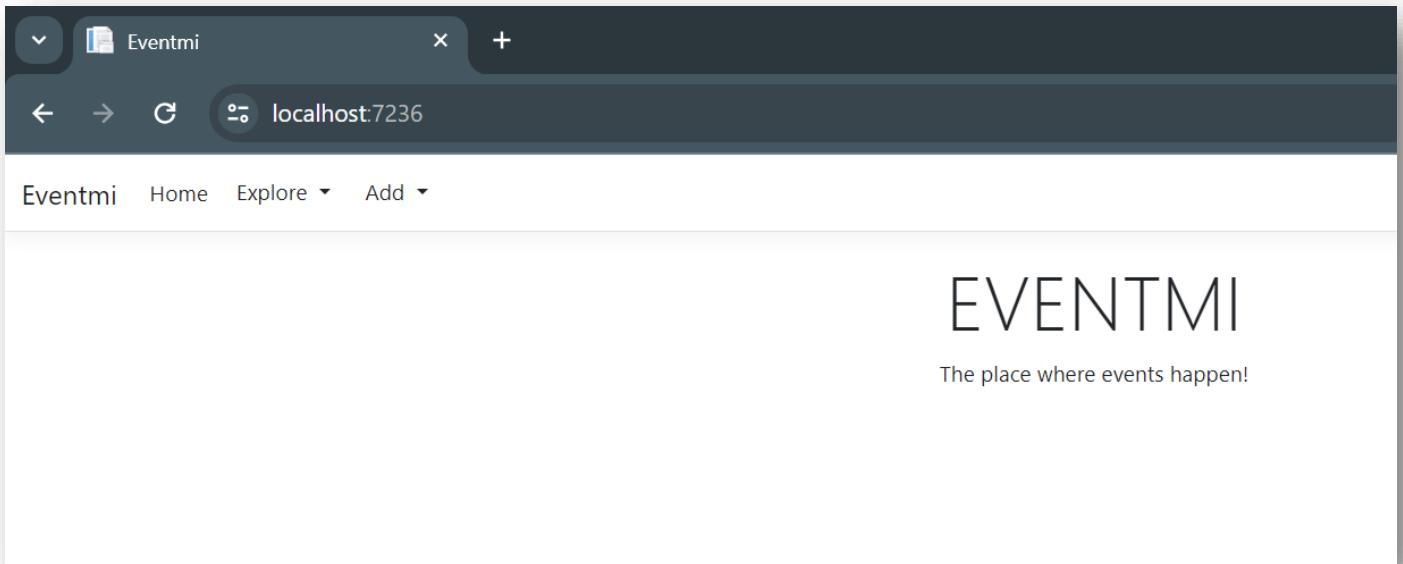
Configure Multiple Startup Projects

- Right-click on the solution in Visual Studio
- Select **"Properties"** from the context menu
- In the properties window, navigate to **"Common Properties" > "Startup Project"**
- Select **"Multiple startup projects"**
- Set the action for both your **ASP.NET Core MVC Eventmi** project and your **Eventmi.Tests** project to **"None"**



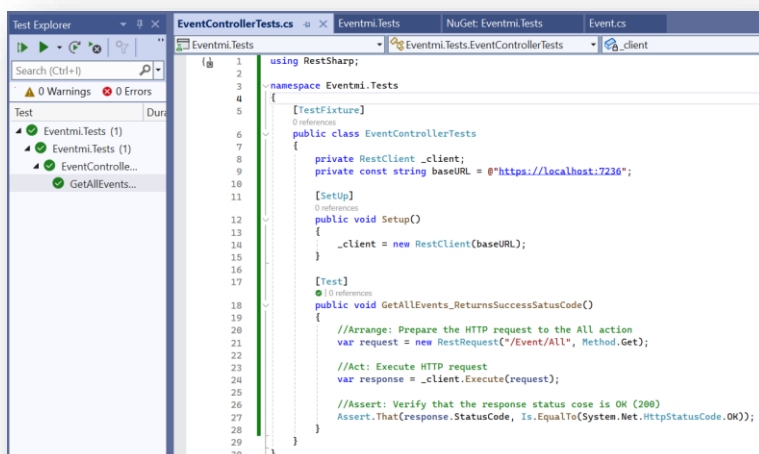
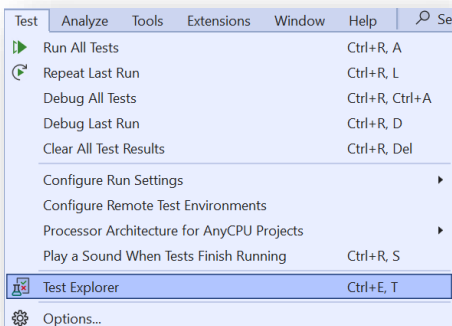
Start the API

Use "Start Without Debugging" (Ctrl+F5) to launch the API. This will start the API without attaching the debugger, freeing up Visual Studio to do other tasks like running tests.



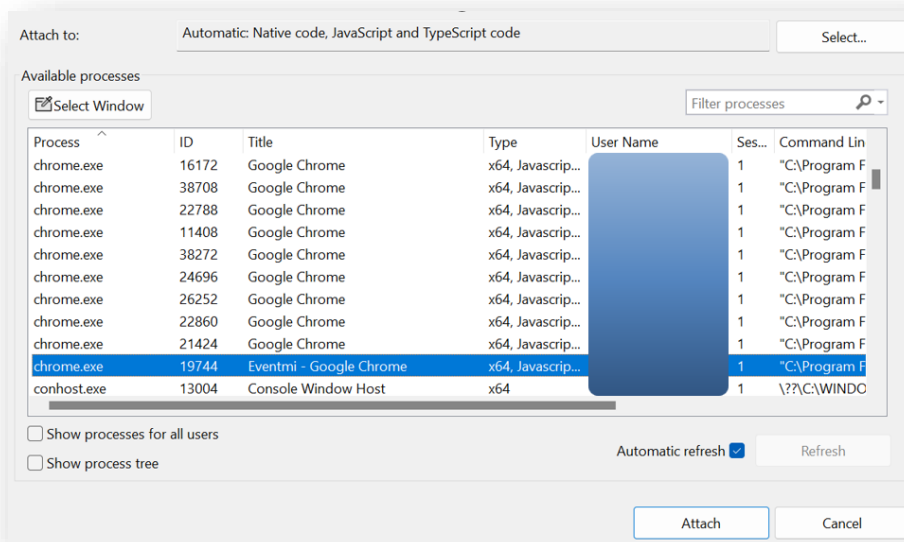
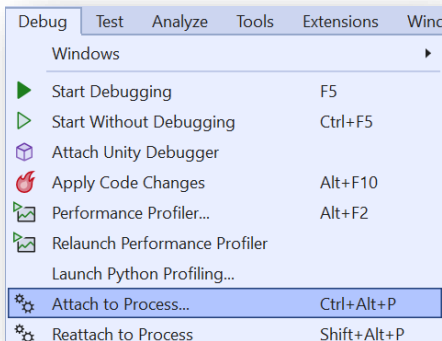
Run Tests

- Open the Test Explorer by going to "Test" > "Test Explorer" in the menu bar
- You can now run your tests manually by clicking "Run All" or selecting specific tests to run.
- This does not require the debugger, so it can be done while the API is running



Attach to Process for Debugging

- If you need to **debug the API**, you can use the "**Attach to Process**" feature.
- Go to "**Debug**" > "**Attach to Process**" and select your running API process.
- This will allow you to **attach the debugger to the running process without stopping it**



5. Testing of EventController Endpoints

This section will guide you through the process of **creating tests for each endpoint in the EventController**. We'll cover how to **structure your tests to ensure they are comprehensive, readable, and maintainable**. The goal is to **achieve high coverage of the controller's functionality**, ensuring every aspect of the **API's behavior is validated**.

Testing "Add" Action (GET Request)

This test will **verify** that the **Add view** is **returned correctly** when navigating to the **add event page**

```
[Test]
✓ | 0 references
public void Add_GetRequest_ReturnsAddView()
```

- **Arrange Phase:**

Creating a RestRequest: The first step involves **setting up the test environment** and **preparing the HTTP request**. Here, a **RestRequest** object is **created with the path "Event/Add"** and the **method Method.Get**. This object represents an **HTTP GET request** directed at the **/Event/Add endpoint** of your application, which is the URL path that users would visit to add a new event.

```
//Arrange
var request = new RestRequest("Event/Add", Method.Get);
```

- **Act Phase:**

Executing the Request: The prepared request is then sent to the server using `_client.Execute(request)`. The `_client` here is an instance of `RestClient` initialized with the base URL of your application, allowing it to make HTTP requests. This step simulates what happens when a user navigates to the "Add Event" page in a web browser.

```
//Act
var response = _client.Execute(request);
```

Assert Phase:

Verifying the Response: The final step is to verify the outcome of the request. The assertion checks that the **HTTP status code in the response is 200 OK**. This status code indicates that the request was successful, and the server has responded with the web page content where users can add a new event.

Testing "Add" Action (POST Request)

This test demonstrates how to perform an **integration test** that not only sends a request to a web application to add a new event but also verifies that the event has indeed been added to the database.

```
[Test]
| 0 references
public void Add_PostRequest_AddsEventAndRedirects()
{
```

- **Setup Request with RestSharp**

First, we create a new `EventFormModel` instance with the event details we want to add. This model mirrors what the web application expects to receive from a form submission

```
var newEvent = new EventFormModel
{
    Name = "DEV: Challenge Accepted",
    Start = new DateTime(2024, 09, 29, 09, 0, 0),
    End = new DateTime(2024, 09, 29, 19, 0, 0),
    Place = "Sofia Tech Park"
};
```

Then, we prepare a `RestRequest` to send to the `/Event/Add` endpoint

```
// Create a request object, specifying the endpoint and method
var request = new RestRequest("/Event/Add", Method.Post);
```

We specify that the request will use **x-www-form-urlencoded** content type, which is typical for form submissions

```
//Specify that the request will use collected data from form
request.AddHeader("Content-Type", "application/x-www-form-urlencoded");
```

We add **each property** of our EventFormModel as a **parameter to this request**

```
//Add form data to the request
request.AddParameter("Name", newEvent.Name);
request.AddParameter("Start", newEvent.Start.ToString("MM/dd/yyyy hh:mm tt"));
request.AddParameter("End", newEvent.End.ToString("MM/dd/yyyy hh:mm tt"));
request.AddParameter("Place", newEvent.Place);
```

- **Execute the Request**

We use a **RestSharp** client to send the request to the web application and capture the response

```
//Act
//Execute the request
var response = _client.Execute(request);
```

- **Assert the Response**

First, we **check the HTTP status code of the response to ensure it's 200 OK**, indicating the request was successfully processed

```
//Assert
Assert.That(response.StatusCode, Is.EqualTo(System.Net.HttpStatusCode.OK));
```

- **Verify Database Addition**

Finally, we **verify that the event has been added to the database**.

We define a **CheckEventExists** method that creates a **new DbContext instance** with the **appropriate options to connect to our database**. Using this context, we query the Events table (or whichever is appropriate) to check if an event with the specified name exists

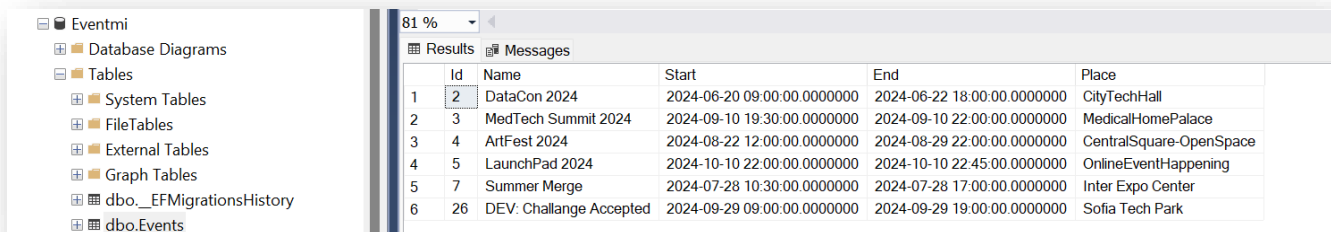
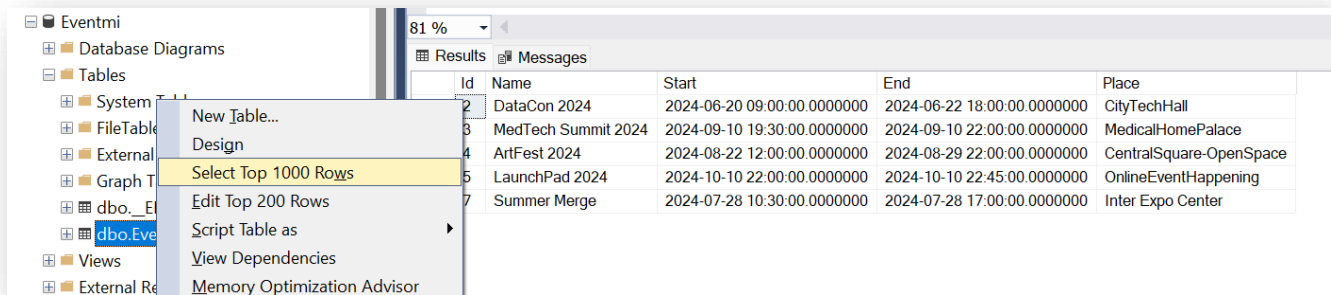
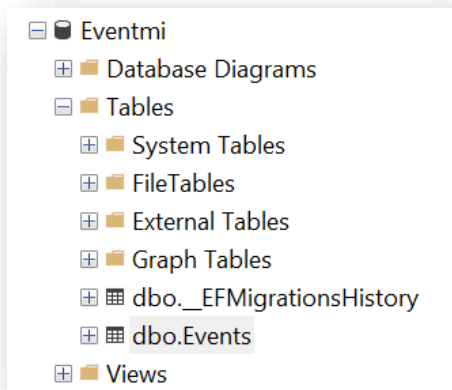
```
1 reference | 1/1 passing
private bool CheckEventExists(string eventName)
{
    // Define your DbContext options
    var options = new DbContextOptionsBuilder<EventmiContext>()
        .UseSqlServer("YourConnectionString")
        .Options;

    // Instantiate the context with the options
    using (var context = new EventmiContext(options))
    {
        // Perform the check
        return context.Events.Any(e => e.Name == eventName);
    }
}
```

And then we use this method in our test assertion

```
// Now check the database
Assert.IsTrue(CheckEventExists(newEvent.Name), "The event was not added to the database.");
```


- Verify the existence of a new event in your database using SQL Server Management Studio



Testing "Details" Action (GET Request)

[Test]

✓ | 0 references

```
public void GetEventDetails_ReturnsSuccessAndExpectedContent()
```

- **Arrange Phase:**

Declare and initialize the variables needed for the test.

Here, **eventId** represents the **ID of an event** you want to retrieve details for. It's set to 1 for this test, implying you're expecting to find an event with this ID in your database.

```
// Arrange: Assuming an event with a known ID exists
var eventId = 1; // Adjust this ID as needed
```


The RestRequest object is configured to make a GET request to the **/Event/Details/{eventId}** endpoint, where **{eventId}** is a placeholder for the actual event ID.

```
// Create a request object, specifying the endpoint and method
var request = new RestRequest($"/Event/Details/{eventId}", Method.Get);
```

- **Act Phase:**

Execute the operation or method being tested

This is where you **send the HTTP GET request** to the server using the **configured RestRequest object**. The Execute method of the **_client** (an instance of RestClient pointed at your API) sends the request to the endpoint and waits for the response. This mimics what happens when a user tries to access the details page of an event in a web application

```
// Act: Execute HTTP request
var response = _client.Execute(request);
```

- **Assert Phase:**

Verify the outcome of the Act phase against the expected result.

After receiving the **response from the server**, this phase checks if the **HTTP status code of the response is 200 (OK)**. This status code confirms that the request was successfully processed

```
// Assert: Verify that the response status code is OK (200)
Assert.That(response.StatusCode, Is.EqualTo(System.Net.HttpStatusCode.OK));
```

Testing "Edit" Action (GET Request)

[Test]

🟢 | 0 references

```
public async Task EditAction_ReturnsViewForValidId()
```

- **Test Setup (Arrange)**

```
// Arrange
int? eventId = 1; // Ensure this is a valid ID for an existing event
var request = new RestRequest($"/Event/Edit/{eventId}", Method.Get);
```

- **Executing the Request (Act)**

```
// Act
var response = await _client.ExecuteAsync(request);
```

- **Verifying the Response (Assert):**

```
// Assert
Assert.That(response.StatusCode, Is.EqualTo(System.Net.HttpStatusCode.OK));
```

Testing "Edit" Action (POST Request) (Successful Edit)

- **Arrange Phase: Test Setup**

Fetching the Event: Initially, you retrieve an **existing event by its eventId** (in this case, 6) to simulate editing an existing record. This step mimics **loading an event's current details to populate the form initially**.

```
// Arrange
var eventId = 6;

var eventToEdit = await GetEventByIdAsync(eventId);
```

```
1 reference | 1/1 passing
private async Task<Event> GetEventByIdAsync(int id)
{
    var options = new DbContextOptionsBuilder<EventmiContext>()
        .UseSqlServer(
            ("Server=.;Database=Eventmi;Trusted_Connection=True;MultipleActiveResultSets=true")
            .Options;

    using (var context = new EventmiContext(options))
    {
        return await context.Events.FirstOrDefaultAsync(e => e.Id == id);
    }
}
```

Updating the Event Model: Create an **EventFormModel** object, representing the **form data that will be submitted**. This model is **populated with the existing event's details** but with an **updated name** to simulate the user changing the event's name in the edit form.

```
var eventModel = new EventFormModel
{
    Id = eventToEdit.Id,
    Name = eventToEdit.Name,
    Start = eventToEdit.Start,
    End = eventToEdit.End,
    Place = eventToEdit.Place
};
```

```
string updatedName = "UpdatedEventName";

eventModel.Name = updatedName;
```

Preparing the Request: **RestRequest** is prepared to send to the **/Event/Edit/{eventId}** endpoint, using the **POST** method. The **content type is set to application/x-www-form-urlencoded**, mimicking a typical form submission from a web browser. The updated event details are added as parameters to the request.

```
var request = new RestRequest($"/Event/Edit/{eventId}", Method.Post);

request.AddHeader("Content-Type", "application/x-www-form-urlencoded");
request.AddParameter("Id", eventModel.Id);
request.AddParameter("Name", eventModel.Name);
request.AddParameter("Start", eventModel.Start.ToString("MM/dd/yyyy hh:mm tt"));
request.AddParameter("End", eventModel.End.ToString("MM/dd/yyyy hh:mm tt"));
request.AddParameter("Place", eventModel.Place);
```

- **Act Phase: Executing the Request**

Sending the Request: The prepared request is sent asynchronously using RestSharp's ExecuteAsync method. This simulates the user clicking the submit button on the edit form

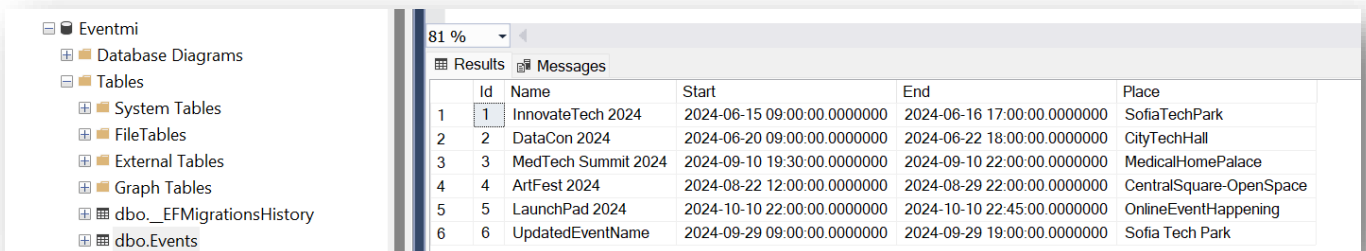
```
// Act
var response = await _client.ExecuteAsync(request);
```

- **Assert Phase: Verifying the Outcome**

Checking the Response: The test verifies that the server's response has a status code of **OK (HTTP 200)**. In a real-world scenario, you might expect a redirect status (like 302) after a successful edit, depending on how your application is designed to respond. For this test, an OK status indicates the request was processed successfully.

```
// Assert
Assert.That(response.StatusCode, Is.EqualTo(System.Net.HttpStatusCode.OK));
```

- **Verify the successful edit in your database using SQL Server Management Studio**



	Id	Name	Start	End	Place
1	1	InnovateTech 2024	2024-06-15 09:00:00.0000000	2024-06-16 17:00:00.0000000	SofiaTechPark
2	2	DataCon 2024	2024-06-20 09:00:00.0000000	2024-06-22 18:00:00.0000000	CityTechHall
3	3	MedTech Summit 2024	2024-09-10 19:30:00.0000000	2024-09-10 22:00:00.0000000	MedicalHomePalace
4	4	ArtFest 2024	2024-08-22 12:00:00.0000000	2024-08-29 22:00:00.0000000	CentralSquare-OpenSpace
5	5	LaunchPad 2024	2024-10-10 22:00:00.0000000	2024-10-10 22:45:00.0000000	OnlineEventHappening
6	6	UpdatedEventName	2024-09-29 09:00:00.0000000	2024-09-29 19:00:00.0000000	Sofia Tech Park

Testing "Edit" Action (POST Request) (Different Scenarios)

- **ID Mismatch Scenario**

```
[Test]
0 references
public async Task EditPostAction_WithIdMismatch_ReturnsNotFound()
```

- The method checks if the **id parameter matches the Id property of the eventModel**.

```
// Arrange
var urlEventId = 6; // The ID in the URL
var modelEventId = 7; // A different ID in the model, simulating a mismatch
```

- If they don't match, it **returns a NotFound result**
- Ensure the method correctly handles cases where the id in the URL does not match the Id in the form data, potentially indicating a bad request or data tampering

```
// Assert
Assert.That(response.StatusCode, Is.EqualTo(System.Net.HttpStatusCode.NotFound));
```

- Invalid Model Scenario

```
[Test]
| 0 references
public async Task EditPostAction_WithInvalidModel_ReturnsViewWithModel()
{
```

- If **ModelState.IsValid** returns false, the **method returns the eventModel back to the view**, allowing the user to correct any errors

```
// Arrange
var eventId = 6; // Assuming this is a valid ID for an existing event

// Here, we simulate an invalid model by intentionally leaving required fields blank or setting them to
// values known to fail validation
var eventModel = new EventFormModel
{
    Id = eventId,
    Name = "", // Assuming Name is required
    // Other fields can be left blank or set to invalid values depending on your validation
    // logic
};
```

- **Verify** the application **properly returns to the edit form with validation feedback** when submitted data fails model validation checks

```
// Assert
Assert.That(response.StatusCode, Is.EqualTo(System.Net.HttpStatusCode.OK));
```

Testing "Delete" Action (POST Request)

```
[Test]
| 0 references
public async Task DeleteAction_WithValidId_RedirectsToAllEvents()
{
```

Eventmi

Database Diagrams

Tables

System Tables

FileTables

External Tables

Graph Tables

dbo__EFMigrationsHistory

dbo.Events

81 %

Results

Messages

	Id	Name	Start	End	Place
1	1	InnovateTech 2024	2024-06-15 09:00:00.0000000	2024-06-16 17:00:00.0000000	SofiaTechPark
2	2	DataCon 2024	2024-06-20 09:00:00.0000000	2024-06-22 18:00:00.0000000	CityTechHall
3	3	MedTech Summit 2024	2024-09-10 19:30:00.0000000	2024-09-10 22:00:00.0000000	MedicalHomePalace
4	4	ArtFest 2024	2024-08-22 12:00:00.0000000	2024-08-29 22:00:00.0000000	CentralSquare-OpenSpace
5	5	LaunchPad 2024	2024-10-10 22:00:00.0000000	2024-10-10 22:45:00.0000000	OnlineEventHappening