GetawayGo

Integration testing

Fontys University of Applied Sciences

Anna Kadurina
18/01/2025

Table of Contents

Introduction

Integration tests are extremely important to endure that the different modules or services in a system work together as expected. Whereas, unit testing is vital, it only focuses on a single unit of code. In a microservices' architecture, the services communicate with each other and is vital to ensure that all functionalities are working as expected.

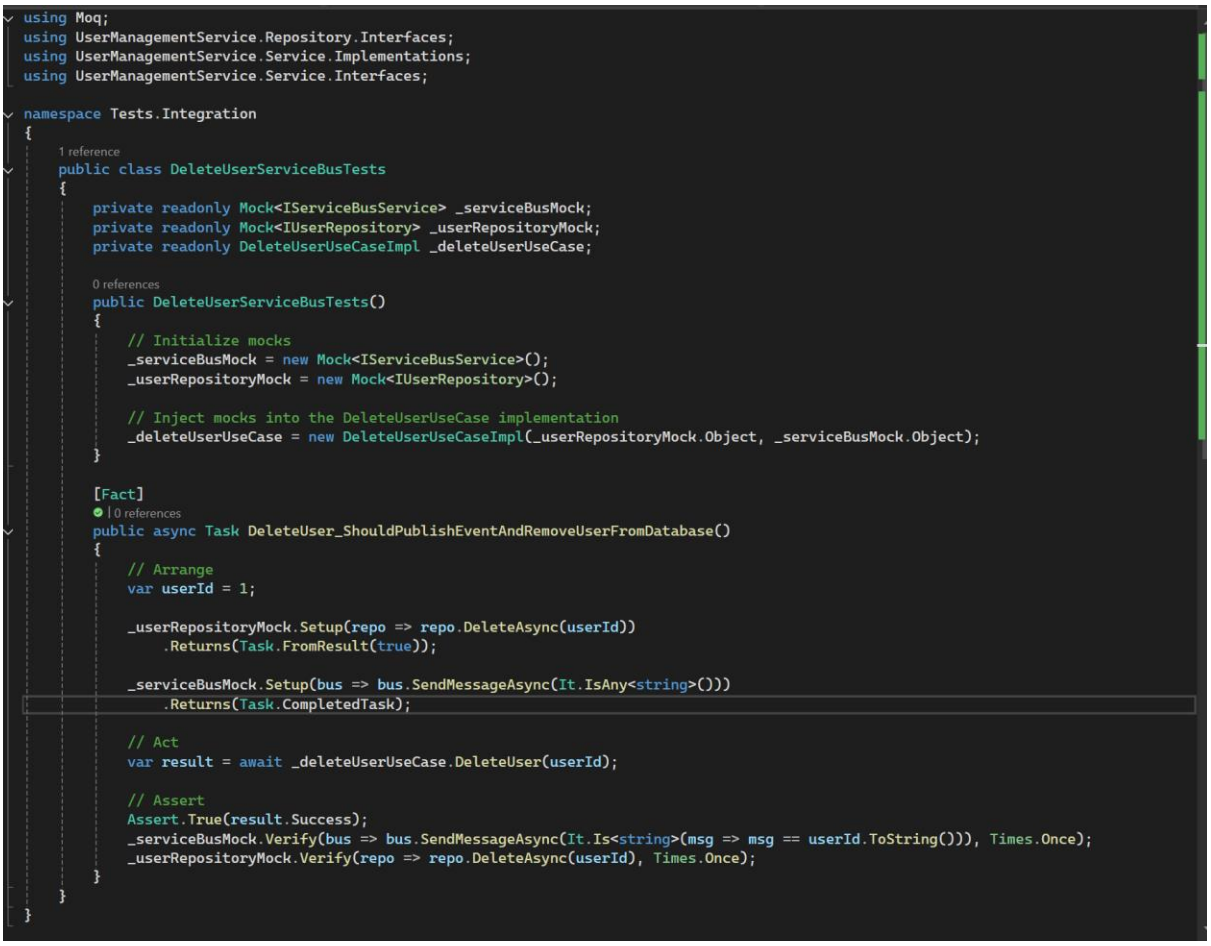
Testing Strategy and Implementation

The focus of the integration testing is on the interactions between the microservices (e.g. BookingService, UserService, NotificationService) and external systems (e.g. Azure Service Bus, Stripe, SendGrid).

All tests in the code run with the pipeline and the successful Build job depends on each one of the test types, including the integration testing.

To perform the tests, I am using the Moq library to mock the dependencies. It is a common and widely used practice to do integration testing.

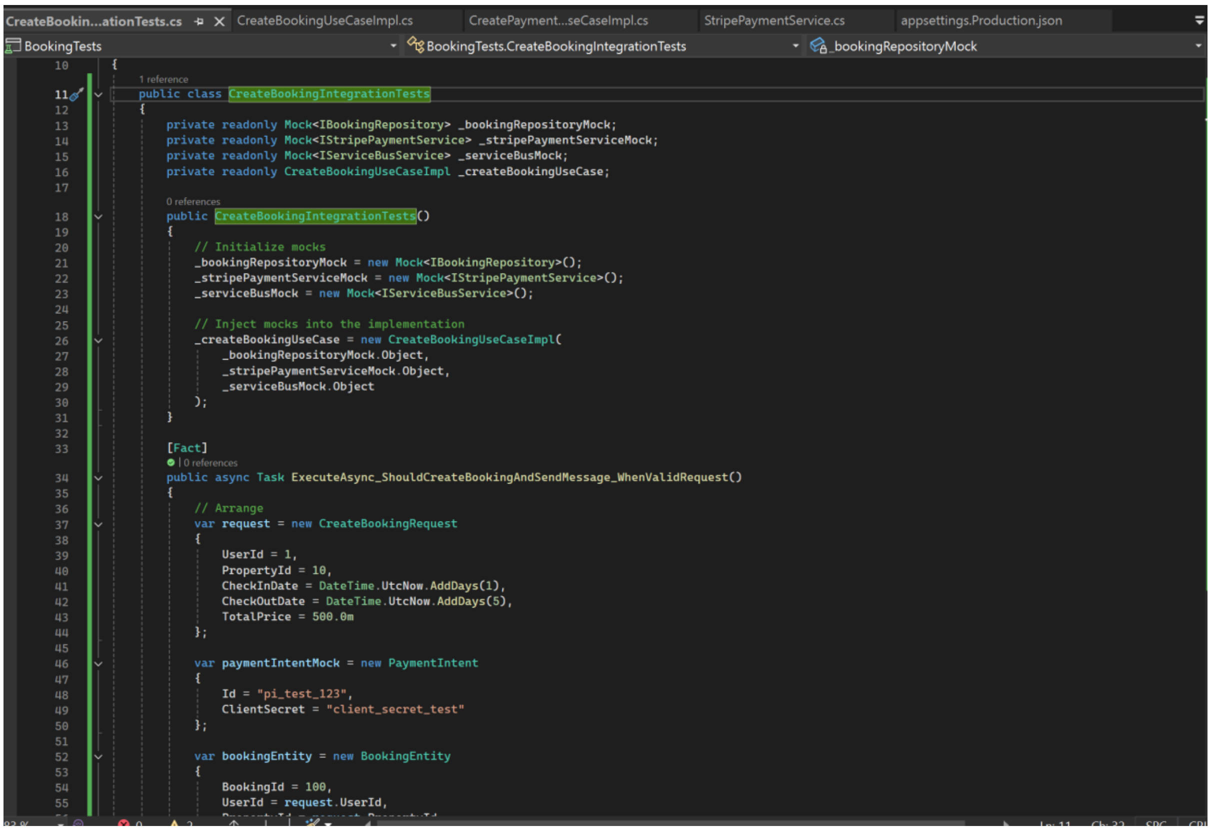
The first example I have provided in this document is of the deletion of a user. When a user is deleted, a message is sent via Azure Service Bus, so that other services can pick up that a user is being removed and all the other data can be removed as well.



```
using Moq;
using UserManagementService.Repository.Interfaces;
using UserManagementService.Service.Implementations;
using UserManagementService.Service.Interfaces;

namespace Tests.Integration
{
    1 reference
    public class DeleteUserServiceBusTests
    {
        private readonly Mock<IServiceBusService> _serviceBusMock;
        private readonly Mock<IUserRepository> _userRepositoryMock;
        private readonly DeleteUserUseCaseImpl _deleteUserUseCase;

        0 references
        public DeleteUserServiceBusTests()
        {
            // Initialize mocks
            _serviceBusMock = new Mock<IServiceBusService>();
            _userRepositoryMock = new Mock<IUserRepository>();

            // Inject mocks into the DeleteUserUseCase implementation
            _deleteUserUseCase = new DeleteUserUseCaseImpl(_userRepositoryMock.Object, _serviceBusMock.Object);
        }

        [Fact]
        | 0 references
        public async Task DeleteUser_ShouldPublishEventAndRemoveUserFromDatabase()
        {
            // Arrange
            var userId = 1;

            _userRepositoryMock.Setup(repo => repo.DeleteAsync(userId))
                .Returns(Task.FromResult(true));

            _serviceBusMock.Setup(bus => bus.SendMessageAsync(It.IsAny<string>()))
                .Returns(Task.CompletedTask);

            // Act
            var result = await _deleteUserUseCase.DeleteUser(userId);

            // Assert
            Assert.True(result.Success);
            _serviceBusMock.Verify(bus => bus.SendMessageAsync(It.Is<string>(msg => msg == userId.ToString())), Times.Once);
            _userRepositoryMock.Verify(repo => repo.DeleteAsync(userId), Times.Once);
        }
    }
}
```

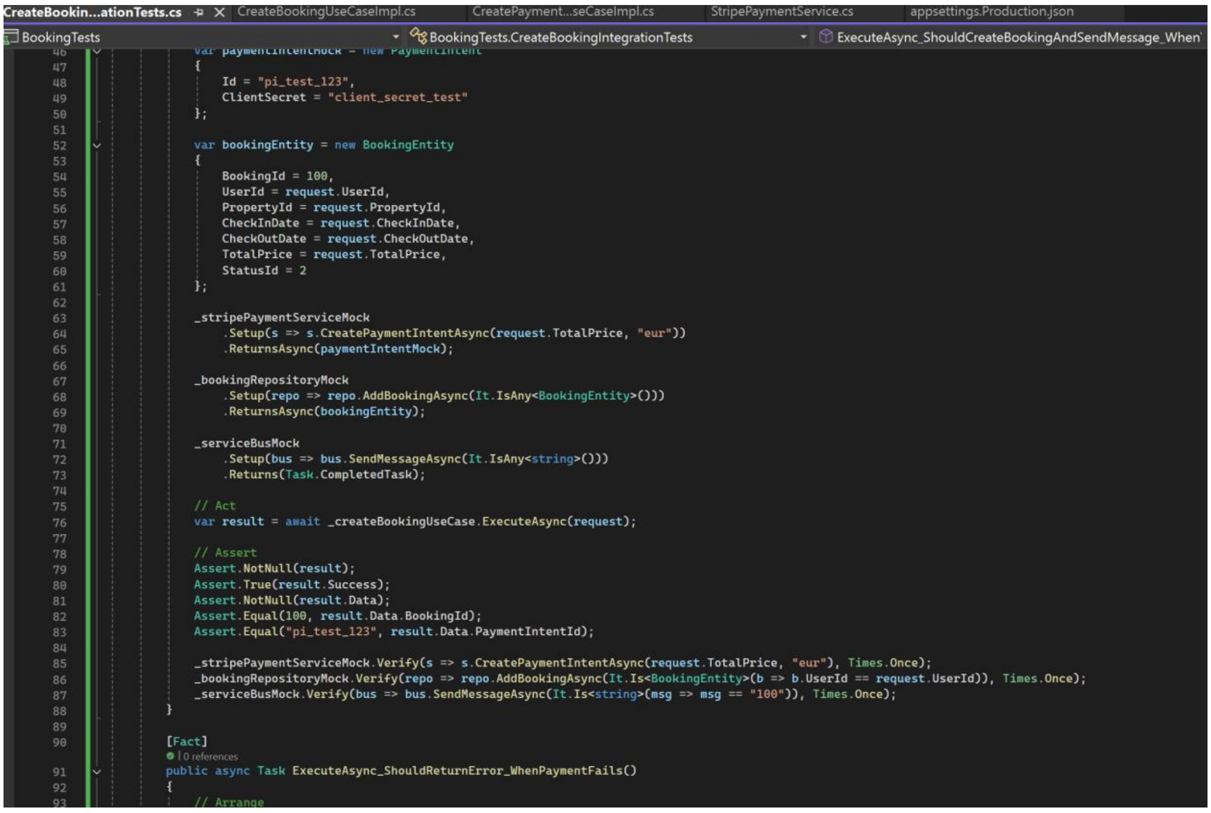
Figure 1 – Deletion of user integration test with Azure Service Bus

The next example is the creation of a booking integration test. In that process, I am utilizing Stripe as a payment system and Azure Service Bus to publish a message. In the test, the functionalities of those dependencies are tested with mocked behaviour to verify the successful execution and expected result.


CreateBookin...ationTests.cs
CreateBookingUseCaseImpl.cs
CreatePayment...seCaseImpl.cs
StripePaymentService.cs
appsettings.Production.json
BookingTests
BookingTests.CreateBookingIntegrationTests
_bookingRepositoryMock

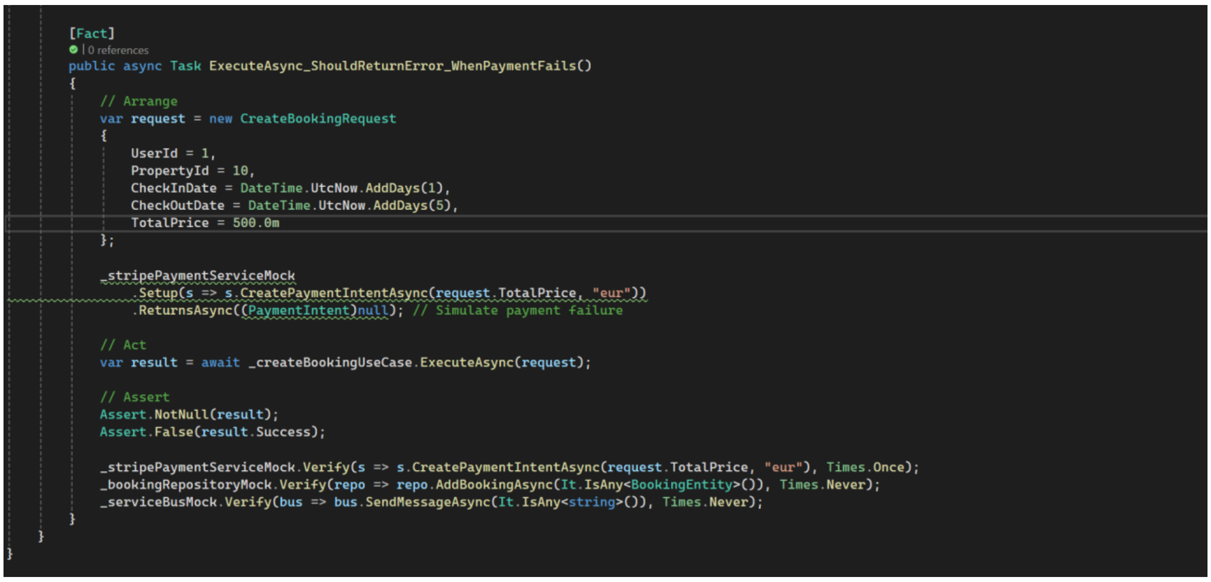
```csharp
{
    1 reference
    public class CreateBookingIntegrationTests
    {
        private readonly Mock<IBookingRepository> _bookingRepositoryMock;
        private readonly Mock<IStripePaymentService> _stripePaymentServiceMock;
        private readonly Mock<IServiceBusService> _serviceBusMock;
        private readonly CreateBookingUseCaseImpl _createBookingUseCase;

        0 references
        public CreateBookingIntegrationTests()
        {
            // Initialize mocks
            _bookingRepositoryMock = new Mock<IBookingRepository>();
            _stripePaymentServiceMock = new Mock<IStripePaymentService>();
            _serviceBusMock = new Mock<IServiceBusService>();

            // Inject mocks into the implementation
            _createBookingUseCase = new CreateBookingUseCaseImpl(
                _bookingRepositoryMock.Object,
                _stripePaymentServiceMock.Object,
                _serviceBusMock.Object
            );
        }

        [Fact]
        | 0 references
        public async Task ExecuteAsync_ShouldCreateBookingAndSendMessage_WhenValidRequest()
        {
            // Arrange
            var request = new CreateBookingRequest
            {
                UserId = 1,
                PropertyId = 10,
                CheckInDate = DateTime.UtcNow.AddDays(1),
                CheckOutDate = DateTime.UtcNow.AddDays(5),
                TotalPrice = 500.0m
            };

            var paymentIntentMock = new PaymentIntent
            {
                Id = "pi_test_123",
                ClientSecret = "client_secret_test"
            };

            var bookingEntity = new BookingEntity
            {
                BookingId = 100,
                UserId = request.UserId,
```

Figure 2 – Creation of booking integration test


CreateBookin...ationTests.cs
CreateBookingUseCaseImpl.cs
CreatePayment...seCaseImpl.cs
StripePaymentService.cs
appsettings.Production.json
BookingTests
BookingTests.CreateBookingIntegrationTests
ExecuteAsync_ShouldCreateBookingAndSendMessage_When

```csharp
            {
                Id = "pi_test_123",
                ClientSecret = "client_secret_test"
            };

            var bookingEntity = new BookingEntity
            {
                BookingId = 100,
                UserId = request.UserId,
                PropertyId = request.PropertyId,
                CheckInDate = request.CheckInDate,
                CheckOutDate = request.CheckOutDate,
                TotalPrice = request.TotalPrice,
                StatusId = 2
            };

            _stripePaymentServiceMock
                .Setup(s => s.CreatePaymentIntentAsync(request.TotalPrice, "eur"))
                .ReturnsAsync(paymentIntentMock);

            _bookingRepositoryMock
                .Setup(repo => repo.AddBookingAsync(It.IsAny<BookingEntity>()))
                .ReturnsAsync(bookingEntity);

            _serviceBusMock
                .Setup(bus => bus.SendMessageAsync(It.IsAny<string>()))
                .Returns(Task.CompletedTask);

            // Act
            var result = await _createBookingUseCase.ExecuteAsync(request);

            // Assert
            Assert.NotNull(result);
            Assert.True(result.Success);
            Assert.NotNull(result.Data);
            Assert.Equal(100, result.Data.BookingId);
            Assert.Equal("pi_test_123", result.Data.PaymentIntentId);

            _stripePaymentServiceMock.Verify(s => s.CreatePaymentIntentAsync(request.TotalPrice, "eur"), Times.Once);
            _bookingRepositoryMock.Verify(repo => repo.AddBookingAsync(It.Is<BookingEntity>(b => b.UserId == request.UserId)), Times.Once);
            _serviceBusMock.Verify(bus => bus.SendMessageAsync(It.Is<string>(msg => msg == "100")), Times.Once);
        }

        [Fact]
        | 0 references
        public async Task ExecuteAsync_ShouldReturnError_WhenPaymentFails()
        {
            // Arrange
```

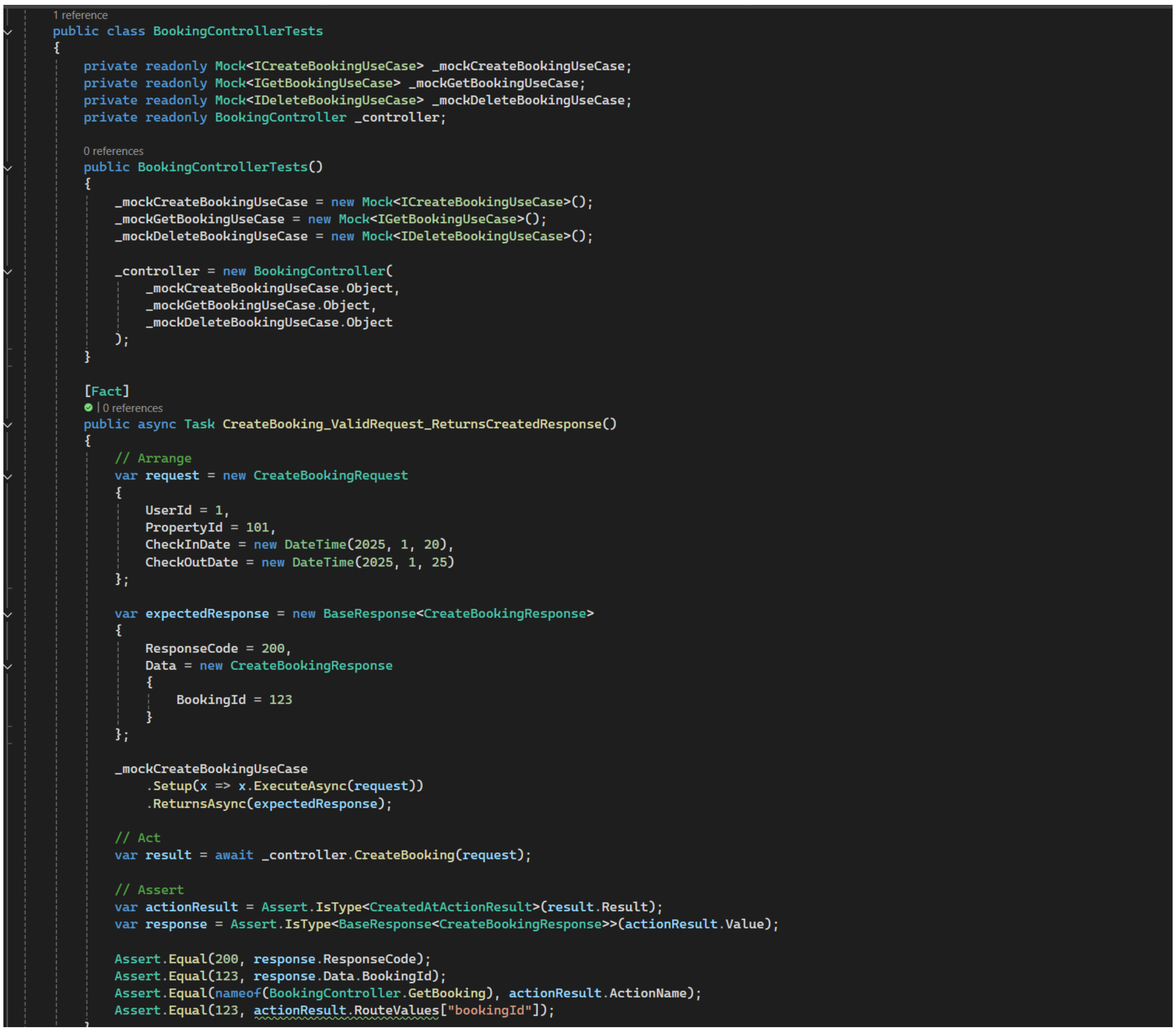
Figure 3 – Creation of booking integration test



```
        [Fact]
        ● | 0 references
        public async Task ExecuteAsync_ShouldReturnError_WhenPaymentFails()
        {
            // Arrange
            var request = new CreateBookingRequest
            {
                UserId = 1,
                PropertyId = 10,
                CheckInDate = DateTime.UtcNow.AddDays(1),
                CheckOutDate = DateTime.UtcNow.AddDays(5),
                TotalPrice = 500.0m
            };

            _stripePaymentServiceMock
                .Setup(s => s.CreatePaymentIntentAsync(request.TotalPrice, "eur"))
                .ReturnsAsync((PaymentIntent)null); // Simulate payment failure

            // Act
            var result = await _createBookingUseCase.ExecuteAsync(request);

            // Assert
            Assert.NotNull(result);
            Assert.False(result.Success);

            _stripePaymentServiceMock.Verify(s => s.CreatePaymentIntentAsync(request.TotalPrice, "eur"), Times.Once);
            _bookingRepositoryMock.Verify(repo => repo.AddBookingAsync(It.IsAny<BookingEntity>()), Times.Never);
            _serviceBusMock.Verify(bus => bus.SendMessageAsync(It.IsAny<string>()), Times.Never);
        }
    }
}
```

Figure 4 – Creation of booking integration test

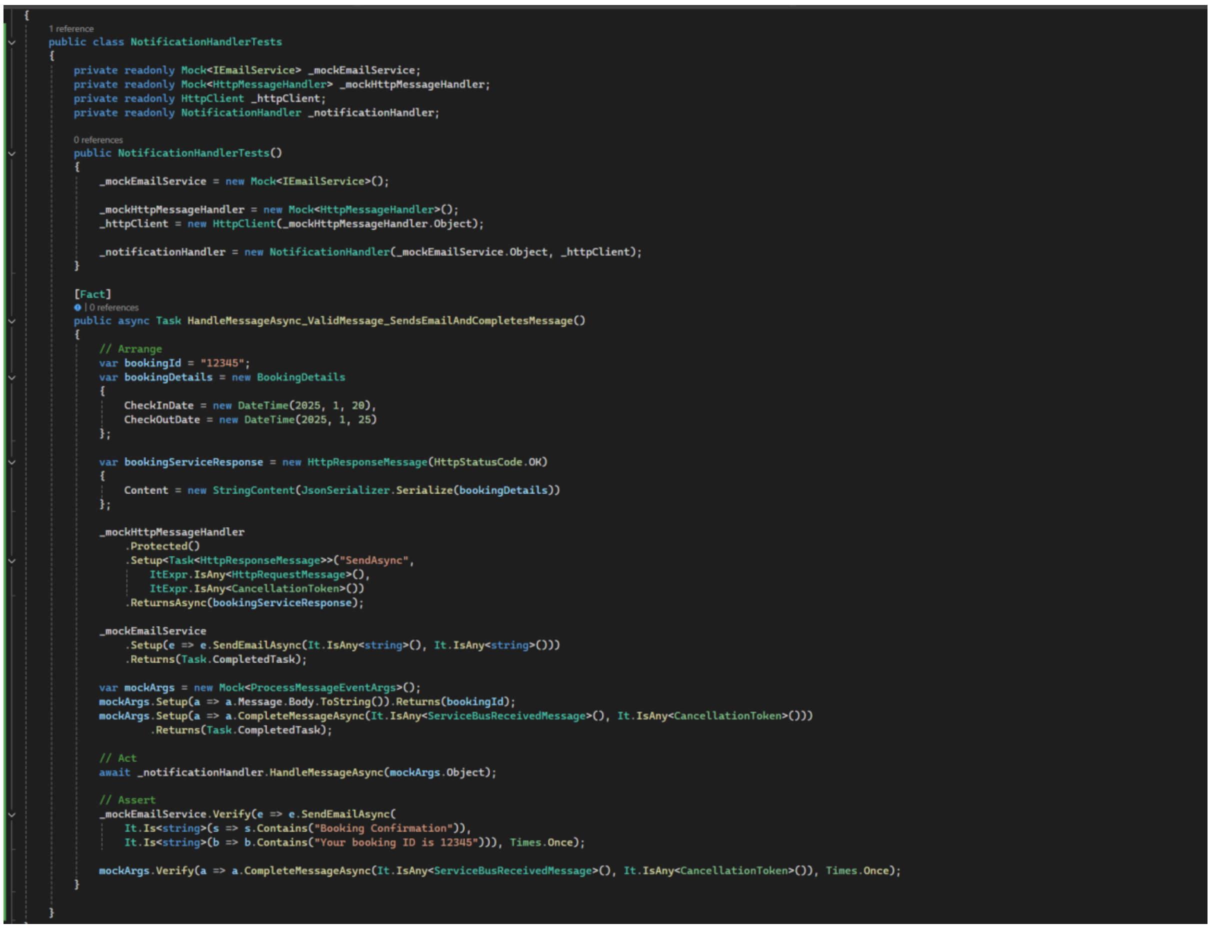
I have also included integration tests on controller level, where I test if the components inside the service itself communicate correctly with each other. The below example is from the BookingService.



```
1 reference
public class BookingControllerTests
{
    private readonly Mock<ICreateBookingUseCase> _mockCreateBookingUseCase;
    private readonly Mock<IGetBookingUseCase> _mockGetBookingUseCase;
    private readonly Mock<IDeleteBookingUseCase> _mockDeleteBookingUseCase;
    private readonly BookingController _controller;

    0 references
    public BookingControllerTests()
    {
        _mockCreateBookingUseCase = new Mock<ICreateBookingUseCase>();
        _mockGetBookingUseCase = new Mock<IGetBookingUseCase>();
        _mockDeleteBookingUseCase = new Mock<IDeleteBookingUseCase>();

        _controller = new BookingController(
            _mockCreateBookingUseCase.Object,
            _mockGetBookingUseCase.Object,
            _mockDeleteBookingUseCase.Object
        );
    }

    [Fact]
    | 0 references
    public async Task CreateBooking_ValidRequest_ReturnsCreatedResponse()
    {
        // Arrange
        var request = new CreateBookingRequest
        {
            UserId = 1,
            PropertyId = 101,
            CheckInDate = new DateTime(2025, 1, 20),
            CheckOutDate = new DateTime(2025, 1, 25)
        };

        var expectedResponse = new BaseResponse<CreateBookingResponse>
        {
            ResponseCode = 200,
            Data = new CreateBookingResponse
            {
                BookingId = 123
            }
        };

        _mockCreateBookingUseCase
            .Setup(x => x.ExecuteAsync(request))
            .ReturnsAsync(expectedResponse);

        // Act
        var result = await _controller.CreateBooking(request);

        // Assert
        var actionResult = Assert.IsType<CreatedAtActionResult>(result.Result);
        var response = Assert.IsType<BaseResponse<CreateBookingResponse>>(actionResult.Value);

        Assert.Equal(200, response.ResponseCode);
        Assert.Equal(123, response.Data.BookingId);
        Assert.Equal(nameof(BookingController.GetBooking), actionResult.ActionName);
        Assert.Equal(123, actionResult.RouteValues["bookingId"]);
```

Figure 5 – Booking Controller integration test

The next integration test focuses on verifying the interaction between the NotificationHandler and its mocked dependencies, IEmailService and HttpClient. The test simulates an HTTP response from the BookingService using a mocked HttpClient and validates that the handler processes the response correctly. Similarly, the mocked IemailService, which uses SendGrid as a third-party tool to send emails, verifies that the correct email is sent.



```
{
    1 reference
    public class NotificationHandlerTests
    {
        private readonly Mock<IEmailService> _mockEmailService;
        private readonly Mock<HttpMessageHandler> _mockHttpMessageHandler;
        private readonly HttpClient _httpClient;
        private readonly NotificationHandler _notificationHandler;

        0 references
        public NotificationHandlerTests()
        {
            _mockEmailService = new Mock<IEmailService>();

            _mockHttpMessageHandler = new Mock<HttpMessageHandler>();
            _httpClient = new HttpClient(_mockHttpMessageHandler.Object);

            _notificationHandler = new NotificationHandler(_mockEmailService.Object, _httpClient);
        }

        [Fact]
        | 0 references
        public async Task HandleMessageAsync_ValidMessage_SendsEmailAndCompletesMessage()
        {
            // Arrange
            var bookingId = "12345";
            var bookingDetails = new BookingDetails
            {
                CheckInDate = new DateTime(2025, 1, 20),
                CheckOutDate = new DateTime(2025, 1, 25)
            };

            var bookingServiceResponse = new HttpResponseMessage(HttpStatusCode.OK)
            {
                Content = new StringContent(JsonSerializer.Serialize(bookingDetails))
            };

            _mockHttpMessageHandler
                .Protected()
                .Setup<Task<HttpResponseMessage>>("SendAsync",
                    ItExpr.IsAny<HttpRequestMessage>(),
                    ItExpr.IsAny<CancellationToken>())
                .ReturnsAsync(bookingServiceResponse);

            _mockEmailService
                .Setup(e => e.SendEmailAsync(It.IsAny<string>(), It.IsAny<string>()))
                .Returns(Task.CompletedTask);

            var mockArgs = new Mock<ProcessMessageEventArgs>();
            mockArgs.Setup(a => a.Message.Body.ToString()).Returns(bookingId);
            mockArgs.Setup(a => a.CompleteMessageAsync(It.IsAny<ServiceBusReceivedMessage>(), It.IsAny<CancellationToken>()))
                    .Returns(Task.CompletedTask);

            // Act
            await _notificationHandler.HandleMessageAsync(mockArgs.Object);

            // Assert
            _mockEmailService.Verify(e => e.SendEmailAsync(
                It.Is<string>(s => s.Contains("Booking Confirmation")),
                It.Is<string>(b => b.Contains("Your booking ID is 12345"))), Times.Once);

            mockArgs.Verify(a => a.CompleteMessageAsync(It.IsAny<ServiceBusReceivedMessage>(), It.IsAny<CancellationToken>()), Times.Once);
        }

    }
```

Figure 6 – Notification Handler integration test

Conclusion

In conclusion, integration tests are crucial for ensuring the reliability of both individual components within a service and the interconnections between services and external tools or third-party systems. They validate that the components work together as intended and that the integration points—such as APIs, messaging systems, and external services—function correctly under expected scenarios.