# Integration tests in ASP.NET Core

01/06/2019 • 38 minutes to read • +2

**In this article**



By Luke Latham, Javier Calvarro Nelson, Steve Smith, and Jos van der Til

Integration tests ensure that an app's components function correctly at a level that includes the app's supporting infrastructure, such as the database, file system, and network. ASP.NET Core supports integration tests using a unit test framework with a test web host and an in-memory test server.

This topic assumes a basic understanding of unit tests. If unfamiliar with test concepts, see the Unit Testing in .NET Core and .NET Standard topic and its linked content.

View or download sample code (how to download)

The sample app is a Razor Pages app and assumes a basic understanding of Razor Pages. If unfamiliar with Razor Pages, see the following topics:

- Introduction to Razor Pages
- Get started with Razor Pages
- Razor Pages unit tests

> **Note**
>
> For testing SPAs, we recommended a tool such as **Selenium**, which can automate a browser.

# Introduction to integration tests

Integration tests evaluate an app's components on a broader level than unit tests. Unit tests are used to test isolated software components, such as individual class methods. Integration tests confirm that two or more app components work together to produce an expected result, possibly including every component required to fully process a request.

These broader tests are used to test the app's infrastructure and whole framework, often including the following components:

- Database
- File system
- Network appliances
- Request-response pipeline

Unit tests use fabricated components, known as *fakes* or *mock objects*, in place of infrastructure components.

In contrast to unit tests, integration tests:

- Use the actual components that the app uses in production.
- Require more code and data processing.
- Take longer to run.

Therefore, limit the use of integration tests to the most important infrastructure scenarios. If a behavior can be tested using either a unit test or an integration test, choose the unit test.

> **Tip**
>
> Don't write integration tests for every possible permutation of data and file access with databases and file systems. Regardless of how many places across an app interact with databases and file systems, a focused set of read, write, update, and delete integration tests are usually capable of adequately testing database and file system components. Use unit tests for routine tests of method logic that interact with these components. In unit tests, the use of infrastructure fakes/mocks result in faster test execution.

> **Note**
>
> In discussions of integration tests, the tested project is frequently called the *System Under Test*, or "SUT" for short.
>
> *"SUT" is used throughout this topic to refer to the tested ASP.NET Core app.*

# ASP.NET Core integration tests

Integration tests in ASP.NET Core require the following:

- A test project is used to contain and execute the tests. The test project has a reference to the SUT.
- The test project creates a test web host for the SUT and uses a test server client to handle requests and responses with the SUT.
- A test runner is used to execute the tests and report the test results.

Integration tests follow a sequence of events that include the usual *Arrange*, *Act*, and *Assert* test steps:

1. The SUT's web host is configured.
2. A test server client is created to submit requests to the app.
3. The *Arrange* test step is executed: The test app prepares a request.
4. The *Act* test step is executed: The client submits the request and receives the response.
5. The *Assert* test step is executed: The *actual* response is validated as a *pass* or *fail* based on an *expected* response.
6. The process continues until all of the tests are executed.
7. The test results are reported.

Usually, the test web host is configured differently than the app's normal web host for the test runs. For example, a different database or different app settings might be used for the tests.

Infrastructure components, such as the test web host and in-memory test server (TestServer), are provided or managed by the Microsoft.AspNetCore.Mvc.Testing package. Use of this package streamlines test creation and execution.

The `Microsoft.AspNetCore.Mvc.Testing` package handles the following tasks:

- Copies the dependencies file (*.deps*) from the SUT into the test project's *bin* directory.
- Sets the content root to the SUT's project root so that static files and pages/views are found when the tests are executed.
- Provides the WebApplicationFactory class to streamline bootstrapping the SUT with `TestServer`.

The unit tests documentation describes how to set up a test project and test runner, along with detailed instructions on how to run tests and recommendations for how to name tests and test classes.

> **Note**

When creating a test project for an app, separate the unit tests from the integration tests into different projects. This helps ensure that infrastructure testing components aren't accidentally included in the unit tests. Separation of unit and integration tests also allows control over which set of tests are run.

There's virtually no difference between the configuration for tests of Razor Pages apps and MVC apps. The only difference is in how the tests are named. In a Razor Pages app, tests of page endpoints are usually named after the page model class (for example, `IndexPageTests` to test component integration for the Index page). In an MVC app, tests are usually organized by controller classes and named after the controllers they test (for example, `HomeControllerTests` to test component integration for the Home controller).

## Test app prerequisites

The test project must:

- Reference the Microsoft.AspNetCore.Mvc.Testing package.
- Specify the Web SDK in the project file (`<Project Sdk="Microsoft.NET.Sdk.Web">`).

These prerequisites can be seen in the sample app. Inspect the *tests/RazorPagesProject.Tests/RazorPagesProject.Tests.csproj* file. The sample app uses the xUnit test framework and the AngleSharp parser library, so the sample app also references:

- xunit
- xunit.runner.visualstudio
- AngleSharp

Entity Framework Core is also used in the tests. The app references:

- Microsoft.AspNetCore.Diagnostics.EntityFrameworkCore
- Microsoft.AspNetCore.Identity.EntityFrameworkCore
- Microsoft.EntityFrameworkCore
- Microsoft.EntityFrameworkCore.InMemory

- Microsoft.EntityFrameworkCore.Tools

## SUT environment

If the SUT's environment isn't set, the environment defaults to Development.

## Basic tests with the default WebApplicationFactory

WebApplicationFactory<TEntryPoint> is used to create a TestServer for the integration tests. `TEntryPoint` is the entry point class of the SUT, usually the `Startup` class.

Test classes implement a *class fixture* interface (IClassFixture) to indicate the class contains tests and provide shared object instances across the tests in the class.

The following test class, `BasicTests`, uses the `WebApplicationFactory` to bootstrap the SUT and provide an HttpClient to a test method, `Get_EndpointsReturnSuccessAndCorrectContentType`. The method checks if the response status code is successful (status codes in the range 200-299) and the `Content-Type` header is `text/html; charset=utf-8` for several app pages.

CreateClient creates an instance of `HttpClient` that automatically follows redirects and handles cookies.

C# Copy

```
public class BasicTests
    : IClassFixture<WebApplicationFactory<RazorPagesProject.Startup>>
{
    private readonly WebApplicationFactory<RazorPagesProject.Startup> _factory;

    public BasicTests(WebApplicationFactory<RazorPagesProject.Startup> factory)
    {
        _factory = factory;
    }
```

```
    [Theory]
    [InlineData("/")]
    [InlineData("/Index")]
    [InlineData("/About")]
    [InlineData("/Privacy")]
    [InlineData("/Contact")]
    public async Task Get_EndpointsReturnSuccessAndCorrectContentType(string url)
    {
        // Arrange
        var client = _factory.CreateClient();

        // Act
        var response = await client.GetAsync(url);

        // Assert
        response.EnsureSuccessStatusCode(); // Status Code 200-299
        Assert.Equal("text/html; charset=utf-8",
            response.Content.Headers.ContentType.ToString());
    }
}
```

By default, non-essential cookies aren't preserved across requests when the GDPR consent policy is enabled. To preserve non-essential cookies, such as those used by the TempData provider, mark them as essential in your tests. For instructions on marking a cookie as essential, see Essential cookies.

## Customize WebApplicationFactory

Web host configuration can be created independently of the test classes by inheriting from `WebApplicationFactory` to create one or more custom factories:

1. Inherit from `WebApplicationFactory` and override ConfigureWebHost. The IWebHostBuilder allows the configuration of the service collection with ConfigureServices:

C# Copy

```
public class CustomWebApplicationFactory<TStartup>
    : WebApplicationFactory<TStartup> where TStartup: class
{
    protected override void ConfigureWebHost(IWebHostBuilder builder)
    {
        builder.ConfigureServices(services =>
        {
            // Remove the app's ApplicationDbContext registration.
            var descriptor = services.SingleOrDefault(
                d => d.ServiceType ==
                    typeof(DbContextOptions<ApplicationDbContext>));

            if (descriptor != null)
            {
                services.Remove(descriptor);
            }

            // Add ApplicationDbContext using an in-memory database for testing.
            services.AddDbContext<ApplicationDbContext>(options =>
            {
                options.UseInMemoryDatabase("InMemoryDbForTesting");
            });

            // Build the service provider.
            var sp = services.BuildServiceProvider();

            // Create a scope to obtain a reference to the database
            // context (ApplicationDbContext).
            using (var scope = sp.CreateScope())
            {
                var scopedServices = scope.ServiceProvider;
                var db = scopedServices.GetRequiredService<ApplicationDbContext>();
                var logger = scopedServices
                    .GetRequiredService<ILogger<CustomWebApplicationFactory<TStartup>>>();
```

```
                // Ensure the database is created.
                db.Database.EnsureCreated();

                try
                {
                    // Seed the database with test data.
                    Utilities.InitializeDbForTests(db);
                }
                catch (Exception ex)
                {
                    logger.LogError(ex, "An error occurred seeding the " +
                        "database with test messages. Error: {Message}", ex.Message);
                }
            }
        });
    }
}
```

Database seeding in the sample app is performed by the `InitializeDbForTests` method. The method is described in the Integration tests sample: Test app organization section.

The SUT's database context is registered in its `Startup.ConfigureServices` method. The test app's `builder.ConfigureServices` callback is executed *after* the app's `Startup.ConfigureServices` code is executed. The execution order is a breaking change for the Generic Host with the release of ASP.NET Core 3.0. To use a different database for the tests than the app's database, the app's database context must be replaced in `builder.ConfigureServices`.

The sample app finds the service descriptor for the database context and uses the descriptor to remove the service registration. Next, the factory adds a new `ApplicationDbContext` that uses an in-memory database for the tests.

To connect to a different database than the in-memory database, change the `UseInMemoryDatabase` call to connect the context to a different database. To use a SQL Server test database:

- Reference the Microsoft.EntityFrameworkCore.SqlServer NuGet package in the project file.

- Call `UseSqlServer` with a connection string to the database.

C# Copy

```
services.AddDbContext<ApplicationDbContext>((options, context) =>
{
    context.UseSqlServer(
        Configuration.GetConnectionString("TestingDbConnectionString"));
});
```

2. Use the custom `CustomWebApplicationFactory` in test classes. The following example uses the factory in the `IndexPageTests` class:

C# Copy

```
public class IndexPageTests :
    IClassFixture<CustomWebApplicationFactory<RazorPagesProject.Startup>>
{
    private readonly HttpClient _client;
    private readonly CustomWebApplicationFactory<RazorPagesProject.Startup>
        _factory;

    public IndexPageTests(
        CustomWebApplicationFactory<RazorPagesProject.Startup> factory)
    {
        _factory = factory;
        _client = factory.CreateClient(new WebApplicationFactoryClientOptions
            {
                AllowAutoRedirect = false
            });
    }
```

The sample app's client is configured to prevent the `HttpClient` from following redirects. As explained later in the Mock authentication section, this permits tests to check the result of the app's first response. The first response is a redirect in many of

these tests with a `Location` header.

3. A typical test uses the `HttpClient` and helper methods to process the request and the response:

C# Copy

```
[Fact]
public async Task Post_DeleteAllMessagesHandler_ReturnsRedirectToRoot()
{
    // Arrange
    var defaultPage = await _client.GetAsync("/");
    var content = await HtmlHelpers.GetDocumentAsync(defaultPage);

    //Act
    var response = await _client.SendAsync(
        (IHtmlFormElement)content.QuerySelector("form[id='messages']"),
        (IHtmlButtonElement)content.QuerySelector("button[id='deleteAllBtn']"));

    // Assert
    Assert.Equal(HttpStatusCode.OK, defaultPage.StatusCode);
    Assert.Equal(HttpStatusCode.Redirect, response.StatusCode);
    Assert.Equal("/", response.Headers.Location.OriginalString);
}
```

Any POST request to the SUT must satisfy the antiforgery check that's automatically made by the app's data protection antiforgery system. In order to arrange for a test's POST request, the test app must:

1. Make a request for the page.
2. Parse the antiforgery cookie and request validation token from the response.
3. Make the POST request with the antiforgery cookie and request validation token in place.

The `SendAsync` helper extension methods (*Helpers/HttpClientExtensions.cs*) and the `GetDocumentAsync` helper method (*Helpers/HtmlHelpers.cs*) in the sample app use the AngleSharp parser to handle the antiforgery check with the following methods:

- `GetDocumentAsync` – Receives the HttpResponseMessage and returns an `IHtmlDocument`. `GetDocumentAsync` uses a factory that prepares a *virtual response* based on the original `HttpResponseMessage`. For more information, see the AngleSharp documentation.
- `SendAsync` extension methods for the `HttpClient` compose an HttpRequestMessage and call SendAsync(HttpRequestMessage) to submit requests to the SUT. Overloads for `SendAsync` accept the HTML form (`IHtmlFormElement`) and the following:
  - Submit button of the form (`IHtmlElement`)
  - Form values collection (`IEnumerable<KeyValuePair<string, string>>`)
  - Submit button (`IHtmlElement`) and form values (`IEnumerable<KeyValuePair<string, string>>`)

> **Note**
>
> AngleSharp is a third-party parsing library used for demonstration purposes in this topic and the sample app. AngleSharp isn't supported or required for integration testing of ASP.NET Core apps. Other parsers can be used, such as the Html Agility Pack (HAP). Another approach is to write code to handle the antiforgery system's request verification token and antiforgery cookie directly.

# Customize the client with WithWebHostBuilder

When additional configuration is required within a test method, WithWebHostBuilder creates a new `WebApplicationFactory` with an IWebHostBuilder that is further customized by configuration.

The `Post_DeleteMessageHandler_ReturnsRedirectToRoot` test method of the sample app demonstrates the use of `WithWebHostBuilder`. This test performs a record delete in the database by triggering a form submission in the SUT.

Because another test in the `IndexPageTests` class performs an operation that deletes all of the records in the database and may run before the `Post_DeleteMessageHandler_ReturnsRedirectToRoot` method, the database is reseeded in this test method to ensure that a

record is present for the SUT to delete. Selecting the first delete button of the `messages` form in the SUT is simulated in the request to the SUT:

C# Copy

```
[Fact]
public async Task Post_DeleteMessageHandler_ReturnsRedirectToRoot()
{
    // Arrange
    var client = _factory.WithWebHostBuilder(builder =>
        {
            builder.ConfigureServices(services =>
            {
                var serviceProvider = services.BuildServiceProvider();

                using (var scope = serviceProvider.CreateScope())
                {
                    var scopedServices = scope.ServiceProvider;
                    var db = scopedServices
                        .GetRequiredService<ApplicationDbContext>();
                    var logger = scopedServices
                        .GetRequiredService<ILogger<IndexPageTests>>();

                    try
                    {
                        Utilities.ReinitializeDbForTests(db);
                    }
                    catch (Exception ex)
                    {
                        logger.LogError(ex, "An error occurred seeding " +
                            "the database with test messages. Error: {Message}",
                            ex.Message);
                    }
                }
            });
        })
```

```
        .CreateClient(new WebApplicationFactoryClientOptions
        {
            AllowAutoRedirect = false
        });
    var defaultPage = await client.GetAsync("/");
    var content = await HtmlHelpers.GetDocumentAsync(defaultPage);

    //Act
    var response = await client.SendAsync(
        (IHtmlFormElement)content.QuerySelector("form[id='messages']"),
        (IHtmlButtonElement)content.QuerySelector("form[id='messages']")
            .QuerySelector("div[class='panel-body']")
            .QuerySelector("button"));

    // Assert
    Assert.Equal(HttpStatusCode.OK, defaultPage.StatusCode);
    Assert.Equal(HttpStatusCode.Redirect, response.StatusCode);
    Assert.Equal("/", response.Headers.Location.OriginalString);
}
```

# Client options

The following table shows the default WebApplicationFactoryClientOptions available when creating `HttpClient` instances.

| Option | Description | Default |
| --- | --- | --- |
| AllowAutoRedirect | Gets or sets whether or not `HttpClient` instances should automatically follow redirect responses. | `true` |
| BaseAddress | Gets or sets the base address of `HttpClient` instances. | `http://localhost` |
| HandleCookies | Gets or sets whether `HttpClient` instances should handle cookies. | `true` |

| Option | Description | Default |
| --- | --- | --- |
| MaxAutomaticRedirections | Gets or sets the maximum number of redirect responses that `HttpClient` instances should follow. | 7 |

Create the `WebApplicationFactoryClientOptions` class and pass it to the CreateClient method (default values are shown in the code example):

C# Copy

```
// Default client option values are shown
var clientOptions = new WebApplicationFactoryClientOptions();
clientOptions.AllowAutoRedirect = true;
clientOptions.BaseAddress = new Uri("http://localhost");
clientOptions.HandleCookies = true;
clientOptions.MaxAutomaticRedirections = 7;

_client = _factory.CreateClient(clientOptions);
```

# Inject mock services

Services can be overridden in a test with a call to ConfigureTestServices on the host builder. **To inject mock services, the SUT must have a `Startup` class with a `Startup.ConfigureServices` method.**

The sample SUT includes a scoped service that returns a quote. The quote is embedded in a hidden field on the Index page when the Index page is requested.

*Services/IQuoteService.cs*:

C# Copy

```
public interface IQuoteService
{
    Task<string> GenerateQuote();
}
```

*Services/QuoteService.cs*:

C# Copy

```
// Quote ©1975 BBC: The Doctor (Tom Baker); Dr. Who: Planet of Evil
// https://www.bbc.co.uk/programmes/p00pyrx6
public class QuoteService : IQuoteService
{
    public Task<string> GenerateQuote()
    {
        return Task.FromResult<string>(
            "Come on, Sarah. We've an appointment in London, " +
            "and we're already 30,000 years late.");
    }
}
```

*Startup.cs*:

C# Copy

```
services.AddScoped<IQuoteService, QuoteService>();
```

*Pages/Index.cshtml.cs*:

C# Copy

```
public class IndexModel : PageModel
{
```

```
    private readonly ApplicationDbContext _db;
    private readonly IQuoteService _quoteService;

    public IndexModel(ApplicationDbContext db, IQuoteService quoteService)
    {
        _db = db;
        _quoteService = quoteService;
    }

    [BindProperty]
    public Message Message { get; set; }

    public IList<Message> Messages { get; private set; }

    [TempData]
    public string MessageAnalysisResult { get; set; }

    public string Quote { get; private set; }

    public async Task OnGetAsync()
    {
        Messages = await _db.GetMessagesAsync();

        Quote = await _quoteService.GenerateQuote();
    }
```

*Pages/Index.cs*:

CSHTML Copy

```
<input id="quote" type="hidden" value="@Model.Quote">
```

The following markup is generated when the SUT app is run:

HTML

Copy

```
<input id="quote" type="hidden" value="Come on, Sarah. We&#x27;ve an appointment in
    London, and we&#x27;re already 30,000 years late.">
```

To test the service and quote injection in an integration test, a mock service is injected into the SUT by the test. The mock service replaces the app's `QuoteService` with a service provided by the test app, called `TestQuoteService`:

*IntegrationTests.IndexPageTests.cs*:

C#

Copy

```
// Quote ©1975 BBC: The Doctor (Tom Baker); Pyramids of Mars
// https://www.bbc.co.uk/programmes/p00pys55
public class TestQuoteService : IQuoteService
{
    public Task<string> GenerateQuote()
    {
        return Task.FromResult<string>(
            "Something's interfering with time, Mr. Scarman, " +
            "and time is my business.");
    }
}
```

`ConfigureTestServices` is called, and the scoped service is registered:

C#

Copy

```
[Fact]
public async Task Get_QuoteService_ProvidesQuoteInPage()
{
    // Arrange
    var client = _factory.WithWebHostBuilder(builder =>
```

```
        {
            builder.ConfigureTestServices(services =>
            {
                services.AddScoped<IQuoteService, TestQuoteService>();
            });
        })
        .CreateClient();

    //Act
    var defaultPage = await client.GetAsync("/");
    var content = await HtmlHelpers.GetDocumentAsync(defaultPage);
    var quoteElement = content.QuerySelector("#quote");

    // Assert
    Assert.Equal("Something's interfering with time, Mr. Scarman, " +
        "and time is my business.", quoteElement.Attributes["value"].Value);
}
```

The markup produced during the test's execution reflects the quote text supplied by `TestQuoteService`, thus the assertion passes:

HTML Copy

```
<input id="quote" type="hidden" value="Something&#x27;s interfering with time,
    Mr. Scarman, and time is my business.">
```

# Mock authentication

Tests in the `AuthTests` class check that a secure endpoint:

- Redirects an unauthenticated user to the app's Login page.
- Returns content for an authenticated user.

In the SUT, the `/SecurePage` page uses an AuthorizePage convention to apply an AuthorizeFilter to the page. For more information, see Razor Pages authorization conventions.

C# Copy

```
services.AddRazorPages()
    .AddRazorPagesOptions(options =>
    {
        options.Conventions.AuthorizePage("/SecurePage");
    });
```

In the `Get_SecurePageRedirectsAnUnauthenticatedUser` test, a WebApplicationFactoryClientOptions is set to disallow redirects by setting AllowAutoRedirect to `false`:

C# Copy

```
[Fact]
public async Task Get_SecurePageRedirectsAnUnauthenticatedUser()
{
    // Arrange
    var client = _factory.CreateClient(
        new WebApplicationFactoryClientOptions
        {
            AllowAutoRedirect = false
        });

    // Act
    var response = await client.GetAsync("/SecurePage");

    // Assert
    Assert.Equal(HttpStatusCode.Redirect, response.StatusCode);
    Assert.StartsWith("http://localhost/Identity/Account/Login",
        response.Headers.Location.OriginalString);
}
```

By disallowing the client to follow the redirect, the following checks can be made:

- The status code returned by the SUT can be checked against the expected HttpStatusCode.Redirect result, not the final status code after the redirect to the Login page, which would be HttpStatusCode.OK.
- The `Location` header value in the response headers is checked to confirm that it starts with `http://localhost/Identity/Account/Login`, not the final Login page response, where the `Location` header wouldn't be present.

The test app can mock an AuthenticationHandler<TOptions> in ConfigureTestServices in order to test aspects of authentication and authorization. A minimal scenario returns an AuthenticateResult.Success:

C# Copy

```
public class TestAuthHandler : AuthenticationHandler<AuthenticationSchemeOptions>
{
    public TestAuthHandler(IOptionsMonitor<AuthenticationSchemeOptions> options,
        ILoggerFactory logger, UrlEncoder encoder, ISystemClock clock)
        : base(options, logger, encoder, clock)
    {
    }

    protected override Task<AuthenticateResult> HandleAuthenticateAsync()
    {
        var claims = new[] { new Claim(ClaimTypes.Name, "Test user") };
        var identity = new ClaimsIdentity(claims, "Test");
        var principal = new ClaimsPrincipal(identity);
        var ticket = new AuthenticationTicket(principal, "Test");

        var result = AuthenticateResult.Success(ticket);

        return Task.FromResult(result);
    }
}
```

The `TestAuthHandler` is called to authenticate a user when the authentication scheme is set to `Test` where `AddAuthentication` is registered for `ConfigureTestServices`:

C# Copy

```
[Fact]
public async Task Get_SecurePageIsReturnedForAnAuthenticatedUser()
{
    // Arrange
    var client = _factory.WithWebHostBuilder(builder =>
        {
            builder.ConfigureTestServices(services =>
            {
                services.AddAuthentication("Test")
                    .AddScheme<AuthenticationSchemeOptions, TestAuthHandler>(
                        "Test", options => {});
            });
        })
        .CreateClient(new WebApplicationFactoryClientOptions
        {
            AllowAutoRedirect = false,
        });

    client.DefaultRequestHeaders.Authorization =
        new AuthenticationHeaderValue("Test");

    //Act
    var response = await client.GetAsync("/SecurePage");

    // Assert
    Assert.Equal(HttpStatusCode.OK, response.StatusCode);
}
```

For more information on `WebApplicationFactoryClientOptions`, see the Client options section.

## Set the environment

By default, the SUT's host and app environment is configured to use the Development environment. To override the SUT's environment:

- Set the `ASPNETCORE_ENVIRONMENT` environment variable (for example, `Staging`, `Production`, or other custom value, such as `Testing`).
- Override `CreateHostBuilder` in the test app to read environment variables prefixed with `ASPNETCORE`.

C# Copy

```
protected override IHostBuilder CreateHostBuilder() =>
    base.CreateHostBuilder()
        .ConfigureHostConfiguration(
            config => config.AddEnvironmentVariables("ASPNETCORE"));
```

## How the test infrastructure infers the app content root path

The `WebApplicationFactory` constructor infers the app content root path by searching for a WebApplicationFactoryContentRootAttribute on the assembly containing the integration tests with a key equal to the `TEntryPoint` assembly `System.Reflection.Assembly.FullName`. In case an attribute with the correct key isn't found, `WebApplicationFactory` falls back to searching for a solution file (*.sln*) and appends the `TEntryPoint` assembly name to the solution directory. The app root directory (the content root path) is used to discover views and content files.

## Disable shadow copying

Shadow copying causes the tests to execute in a different directory than the output directory. For tests to work properly, shadow copying must be disabled. The sample app uses xUnit and disables shadow copying for xUnit by including an *xunit.runner.json* file with

the correct configuration setting. For more information, see Configuring xUnit with JSON.

Add the *xunit.runner.json* file to root of the test project with the following content:

JSON Copy

```
{
  "shadowCopy": false
}
```

# Disposal of objects

After the tests of the `IClassFixture` implementation are executed, TestServer and HttpClient are disposed when xUnit disposes of the WebApplicationFactory. If objects instantiated by the developer require disposal, dispose of them in the `IClassFixture` implementation. For more information, see Implementing a Dispose method.

# Integration tests sample

The sample app is composed of two apps:

| App | Project directory | Description |
| --- | --- | --- |
| Message app (the SUT) | *src/RazorPagesProject* | Allows a user to add, delete one, delete all, and analyze messages. |
| Test app | *tests/RazorPagesProject.Tests* | Used to integration test the SUT. |

The tests can be run using the built-in test features of an IDE, such as Visual Studio. If using Visual Studio Code or the command line, execute the following command at a command prompt in the *tests/RazorPagesProject.Tests* directory:

console Copy

```
dotnet test
```

## Message app (SUT) organization

The SUT is a Razor Pages message system with the following characteristics:

- The Index page of the app (*Pages/Index.cshtml* and *Pages/Index.cshtml.cs*) provides a UI and page model methods to control the addition, deletion, and analysis of messages (average words per message).
- A message is described by the `Message` class (*Data/Message.cs*) with two properties: `Id` (key) and `Text` (message). The `Text` property is required and limited to 200 characters.
- Messages are stored using Entity Framework's in-memory database†.
- The app contains a data access layer (DAL) in its database context class, `AppDbContext` (*Data/AppDbContext.cs*).
- If the database is empty on app startup, the message store is initialized with three messages.
- The app includes a `/SecurePage` that can only be accessed by an authenticated user.

†The EF topic, Test with InMemory, explains how to use an in-memory database for tests with MSTest. This topic uses the xUnit test framework. Test concepts and test implementations across different test frameworks are similar but not identical.

Although the app doesn't use the repository pattern and isn't an effective example of the Unit of Work (UoW) pattern, Razor Pages supports these patterns of development. For more information, see Designing the infrastructure persistence layer and Test controller logic (the sample implements the repository pattern).

## Test app organization

The test app is a console app inside the *tests/RazorPagesProject.Tests* directory.

| Test app directory | Description |
| --- | --- |

| Test app directory | Description |
| --- | --- |
| *AuthTests* | Contains test methods for:<br>• Accessing a secure page by an unauthenticated user.<br>• Accessing a secure page by an authenticated user with a mock AuthenticationHandler<TOptions>.<br>• Obtaining a GitHub user profile and checking the profile's user login. |
| *BasicTests* | Contains a test method for routing and content type. |
| *IntegrationTests* | Contains the integration tests for the Index page using custom `WebApplicationFactory` class. |
| *Helpers/Utilities* | • *Utilities.cs* contains the `InitializeDbForTests` method used to seed the database with test data.<br>• *HtmlHelpers.cs* provides a method to return an AngleSharp `IHtmlDocument` for use by the test methods.<br>• *HttpClientExtensions.cs* provide overloads for `SendAsync` to submit requests to the SUT. |

The test framework is xUnit. Integration tests are conducted using the Microsoft.AspNetCore.TestHost, which includes the TestServer. Because the Microsoft.AspNetCore.Mvc.Testing package is used to configure the test host and test server, the `TestHost` and `TestServer` packages don't require direct package references in the test app's project file or developer configuration in the test app.

### Seeding the database for testing

Integration tests usually require a small dataset in the database prior to the test execution. For example, a delete test calls for a database record deletion, so the database must have at least one record for the delete request to succeed.

The sample app seeds the database with three messages in *Utilities.cs* that tests can use when they execute:

C# Copy

```
public static void InitializeDbForTests(ApplicationDbContext db)
{
    db.Messages.AddRange(GetSeedingMessages());
    db.SaveChanges();
}

public static void ReinitializeDbForTests(ApplicationDbContext db)
{
    db.Messages.RemoveRange(db.Messages);
    InitializeDbForTests(db);
}

public static List<Message> GetSeedingMessages()
{
    return new List<Message>()
    {
        new Message(){ Text = "TEST RECORD: You're standing on my scarf." },
        new Message(){ Text = "TEST RECORD: Would you like a jelly baby?" },
        new Message(){ Text = "TEST RECORD: To the rational mind, " +
            "nothing is inexplicable; only unexplained." }
    };
}
```

The SUT's database context is registered in its `Startup.ConfigureServices` method. The test app's `builder.ConfigureServices` callback is executed *after* the app's `Startup.ConfigureServices` code is executed. To use a different database for the tests, the app's database context must be replaced in `builder.ConfigureServices`. For more information, see the Customize WebApplicationFactory section.

# Additional resources

- Unit tests
- Razor Pages unit tests in ASP.NET Core
- ASP.NET Core Middleware

- Test controller logic in ASP.NET Core

### Is this page helpful?

Yes No