Hydrogen Sessions

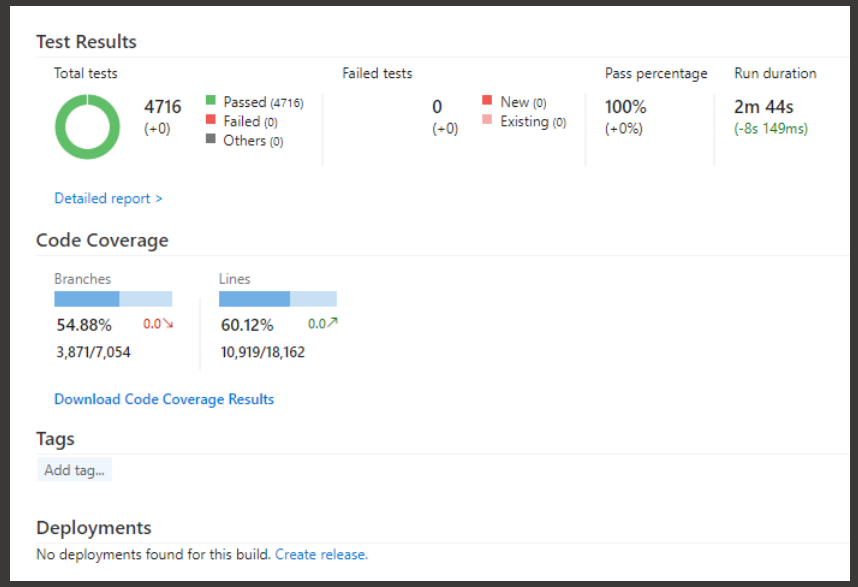
#9 - Testing

Objetivo: code coverage

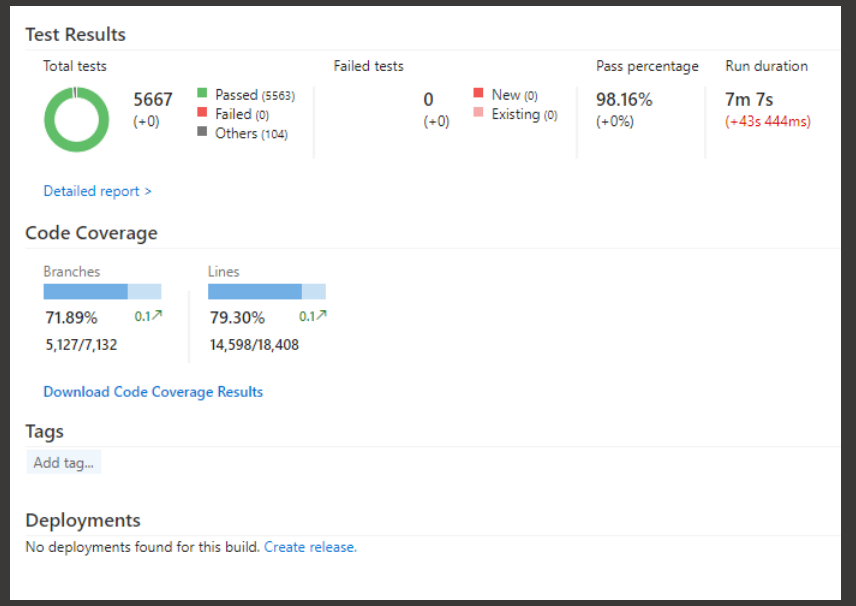
Não é necessário discutir a pertinência de construir um leque muito alargado de testes sobre TODOS os tipos definidos no Hydrogen.

Mais importante que assegurar o funcionamento correto de determinado componente, estes testes dão garantias num aspeto muito importante: a compatibilidade. Asseguram que tudo (ou virtualmente tudo) continua a funcionar quando fazemos "major upgrades", seja do .NET Core, seja nos próprios componentes.

Portanto, é fundamental que a cobertura de código – a % do código que efetivamente é executado nos testes – seja a maior possível. Nunca será possível atingir 100% (porque o esforço a partir de determinado ponto torna-se exponencial) e por isso convencionou-se atingir pelo menos 75%. Podia ser outro valor, mas não deixa de ser assinalável que tenhamos já neste momento uma % tão alta.



60% na build development. ~4700 testes. ~3 minutos para executar os testes.



79% na build mainline. ~5700 testes. ~7 minutos para executar os testes.

Testes unitários e testes de integração

Os testes unitários validam os componentes individualmente. Os testes de integração testam os componentes relacionados em conjunto.

No caso do Hydrogen, os testes de integração são, na realidade, híbridos:

- Há testes de integração que validam de facto a integração de vários componentes.
- Há testes de integração que validam componentes individuais mas com testes que implicam a utilização de recursos externos (nomeadamente, no Azure).

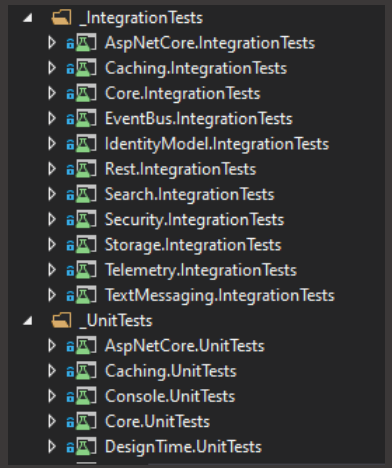
As razões para essa classificação do 2º tipo de testes como testes de integração são as seguintes:

- Há componentes cujo comportamento só pode ser verdadeiramente 100% validado quando utilizam os recursos externos. Eles podem ser "mockados", mas não poderíamos ter a certeza de determinado funcionamento.
- Esses testes com recursos externos são, por natureza, mais lentos e, por isso, implicam muito no tempo total da build.

Os testes unitários são executados na build development e na build mainline.

Os testes de integração são executados apenas na build mainline.

Assim conseguimos que a build development seja mais rápida – para permitir validar rapidamente os check-ins realizados – mas continuamos a garantir o comportamento total dos componentes na build mainline (que é aquela que produz as livrarias que são utilizadas de facto pelas aplicações, micro serviços, etc.) (é isto que explica a insistência para que só sejam referenciados componentes da mainline do Hydrogen).



xUnit

O Visual Studio suporta várias “plataformas” para execução de testes.

Optamentos pelo xUnit porque é mais simples e porque é, provavelmente, aquela que tem mais aceitação e adesão.

É de esperar que, por isso, continue a ser muito bem suportada pelas ferramentas e continue a evoluir.

FluentAssertions

Uma parte fundamental dos testes são as assertions, as verificações.

Embora o xUnit tenha a sua própria API para isso, optamos por esta biblioteca porque torna essas asserções mais “human-readable”.

Todas as assertions devem ser feitas usando esta API.

```
/// <summary>
/// Tests the <see cref="SmartGuard.NotNull{T}(Expression{Func{T}}, T)"/> method.
/// </summary>
[Fact]
[Trait(TraitName, TraitValue)]
[SuppressMessage("Microsoft.Naming", "CA1707:IdentifiersShouldNotContainUnderscores")]
| 0 references
public void SmartGuard_NotNull()
{
    string s = null;
    Action action = () => SmartGuard.NotNull(() => s, s);
    action.Should().Throw<ArgumentNullException>().And.Message.Should().Contain(CoreResources.RES_Exception_ArgCannotBeNull);
}
```

```
[Fact]
[Trait(TraitName, TraitValue)]
[SuppressMessage("Microsoft.Naming", "CA1707:IdentifiersShouldNotContainUnderscores")]
| 0 references | Hugo Ribeiro, 135 days ago | 1 author, 1 change
public async Task Pipeline_InvokeAsync_Context_Null()
{
    PipelineDelegate<PipelineContext> handler = new PipelineDelegate<PipelineContext>(
        (context) =>
        {
            return Task.CompletedTask;
        });

    Pipeline<PipelineContext> pipeline = new Pipeline<PipelineContext>(handler);
    Func<Task> action = () => pipeline.InvokeAsync(null);
    await action.Should().ThrowAsync<ArgumentNullException>().ConfigureAwait(false);
}
```

```
[Fact]
[Trait(TraitName, TraitValue)]
[SuppressMessage("Microsoft.Naming", "CA1707:IdentifiersShouldNotContainUnderscores")]
| 0 references | Hugo Ribeiro, 135 days ago | 1 author, 1 change
public async Task Pipeline_InvokeAsync()
{
    string value = null;

    PipelineDelegate<PipelineContext> handler = new PipelineDelegate<PipelineContext>(
        (context) =>
        {
            value = "Hello";
            return Task.CompletedTask;
        });

    Pipeline<PipelineContext> pipeline = new Pipeline<PipelineContext>(handler);
    await pipeline.InvokeAsync(new PipelineContext()).ConfigureAwait(false);

    value.Should().Be("Hello");
}
```

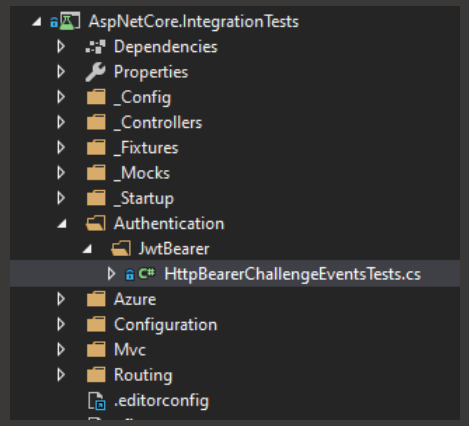
Classes e métodos de teste

Deve procurar-se uma implementação uniforme dos testes. Ou seja, criar todas as classes, métodos, mocks, etc. de uma forma similar.

Os padrões a observar são os seguintes:

- A classe de teste deve ter um nome da norma [Componente]Tests.
- Estas classes devem ser colocadas em pastas com a mesma lógica de namespaces dos componentes normais.
- O método de teste deve ter um nome descritivo (usando _ para separar palavras).
- Deve ter a anotação xUnit adequada (ex.: [Fact]).
- Não deve ter a anotação [Trait] porque não acrescenta nada no VS2019 (ainda há testes que a têm mas devem ser progressivamente corrigidos).
- Os mocks devem ser colocados numa pasta própria (_Mocks).
- Outras dependências devem ser colocadas em pastas próprias (_Config, _Fixtures, _Files), como fizer mais sentido.

Exemplo




```
namespace Primavera.Hydrogen.AspNetCore.IntegrationTests.Authentication.JwtBearer
{
    /// <summary>
    /// Provides unit tests on the <see cref="HttpBearerChallengeEvents"/> class.
    /// </summary>
    1 reference | Hugo Ribeiro, 134 days ago | 1 author, 1 change
    public sealed partial class HttpBearerChallengeEventsTests
    {
        #region Test Methods

        #region Challenge

        /// <summary>
        /// Tests the <see cref="HttpBearerChallengeEvents.Challenge(JwtBearerChallengeContext)" /> method.
        /// </summary>
        /// <returns>
        /// The <see cref="Task"/> that represents the asynchronous operation.
        /// </returns>
        [Fact]
        [SuppressMessage("Microsoft.Naming", "CA1707:IdentifiersShouldNotContainUnderscores")]
        0 references | Hugo Ribeiro, 134 days ago | 1 author, 1 change
        public async Task HttpBearerChallengeEvents_Challenge()
        {
            using ServiceClientFixture<ServiceClientStartup> fixture = new ServiceClientFixture<ServiceClientStartup>();

            using EmployeesServiceClient client = fixture.GetClientCredentialsClientWithCallback();

            ServiceOperationResult<EmployeeData> result1 = await client.GetEmployeeAsync("id").ConfigureAwait(false);
            result1.Should().NotBeNull();
        }
```

Fixtures

Esta funcionalidade do xUnit permite associar contexto a vários testes:

https://xunit.net/docs/shared-context

Pode ser utilizada para vários fins, mas no Hydrogen é utilizada em particular nos testes de integração, para preparar e configurar os “servidores” usados nos testes, seja para rodar uma Web API, seja para rodar um Identity Server em memória.

Exemplos:

```
/// <summary>
/// Tests the model validation in a API controller.
/// </summary>
/// <returns>The <see cref="Task"/> that represents the asynchronous operation.</returns>
[Fact]
[SuppressMessage("Microsoft.Naming", "CA1707:IdentifiersShouldNotContainUnderscores")]
0 references | Hugo Ribeiro, 134 days ago | 1 author, 1 change
public async Task ApiModelValidation_PlainModel_Valid()
{
    using ApiModelValidationFixture fixture = new ApiModelValidationFixture();

    using HttpRequestMessage request = new HttpRequestMessage(HttpMethod.Post, "http://localhost/api1/v1/markets");
    this.AddJson(
        request,
        new MarketData()
        {
            Id = "market1",
            Name = "Market 1",
            Share = 10.1
        });

    using HttpResponseMessage response = await this.GetResponseAsync(request, fixture.TestServerFixture.TestServer.CreateHandler()).ConfigureAwait(false);
    response.StatusCode.Should().Be(HttpStatusCode.Created);

    response.Headers.Location.AbsoluteUri.Should().Be("http://localhost/api1/v1/markets/market1");

    string content = await response.Content.ReadAsStringAsync().ConfigureAwait(false);
    content.Should().Be("market1");

    response.Content.Headers.ContentType.ToString().Should().Be("text/plain; charset=utf-8");
}
```

```
/// <summary>
/// Provides unit tests for the <see cref="ServiceClient{T}" /> type.
/// </summary>
/// <seealso cref="EnglishCultureTestClass"/>
[SuppressMessage("Microsoft.Maintainability", "CA1506:AvoidExcessiveClassCoupling")]
4 references | Hugo Ribeiro, 45 days ago | 1 author, 3 changes | 1 incoming change
public sealed partial class ServiceClientTestsAuth :
    EnglishCultureTestClass,
    IClassFixture<ServiceClientTestsAuthFixture<ServiceClientWithAuthStartup>>
{
    #region Private Properties

    72 references | Hugo Ribeiro, 45 days ago | 1 author, 2 changes
    private ServiceClientTestsAuthFixture<ServiceClientWithAuthStartup> Fixture
    {
        get;
        set;
    }

    #endregion

    Constructors

    #region Test Methods
```

Primavera.Hydrogen.DesignTime

O Hydrogen também inclui 2 assemblies com tipos que ajudam no desenvolvimento dos próprios testes:

Primavera.Hydrogen.DesignTime.UnitTesting é vocacionada para testes unitários. Inclui helpers para ler embedded resources (EmbeddedResourcesService), classes base para as classes de teste (ex.: EnglishCultureTestClass), etc.

Primavera.Hydrogen.DesignTime.IntegrationTesting é vocacionada para testes de integração. Inclui classes base para fixtures (ex.: TestServerFixture), o Identity Test Server e ainda uma coleção de referências para recursos do Azure que devem ser usados nos testes (ExternalTestResources).

TestServer

O .NET Core inclui uma funcionalidade muito interessante que permite executar um Web server (uma API, uma aplicação MVC, etc.) em memória e assim simular o comportamento de determinado servidor necessário aos testes. É uma técnica de mocking de servidores, digamos assim.

https://docs.microsoft.com/en-us/aspnet/core/test/integration-tests

https://docs.microsoft.com/en-us/dotnet/api/microsoft.aspnetcore.testhost.testserver

Este tipo tira partido do generic host do .NET Core e assim permite que se "customize" o comportamento do servidor das formas necessárias aos testes. As fixtures TestServerFixture e TestServerFixture<TStartup> disponíveis em Primavera.Hydrogen.DesignTime.IntegrationTesting tiram exatamente partido disso.

```
/// <summary>
/// Initializes a new instance of the <see cref="ApiModelValidationFixture"/> class.
/// </summary>
/// <param name="enableModelInvalidResponse">If true then <see cref="ApiBehaviorOptions.SuppressModelStateInvalidFilter"/> will be false.</param>
6 references | Hugo Ribeiro, 134 days ago | 1 author, 1 change
public ApiModelValidationFixture(bool enableModelInvalidResponse = false)
{
    // Initialize test server

    this.TestServerFixture = new TestServerFixture()
    {
        ConfigureLoggingAction =
            (context, builder) =>
            {
                builder.SetMinimumLevel(LogLevel.Trace);
                builder.AddDebug();
            },
        ConfigureServicesAction =
            (context, services) =>
            {
                // Stores

                services
                    .AddSingleton<CustomersStore>()
                    .AddSingleton<MarketsStore>();

                // MVC

                services
                    .AddControllers()
                    .ConfigureApiBehaviorOptionsWithoutModelInvalidResponse(
                        (options) =>
                        {
                            if (enableModelInvalidResponse)
                            {
                                options.SuppressModelStateInvalidFilter = false;
                            }
                        })
                    .AddApplicationPart(typeof(CustomersController).Assembly);

                services.AddApiVersioning(new ApiVersion(1, 0));
            },
        ConfigureAction =
            (app) =>
            {
                // Static files
```

IdentityTestServer

Este tipo é apenas um test server especial, que permite configurar o Identity Server em memória, especificando no código de teste (tipicamente na fixture) os API Resources, Identity Resources, Clients, etc. necessários.

Combinando este servidor com um test server sobre determinada API é então possível fazer testes de integração sobre a API com o Identity Server (o que é um caso muito importante para grande parte dos componentes do Hydrogen).

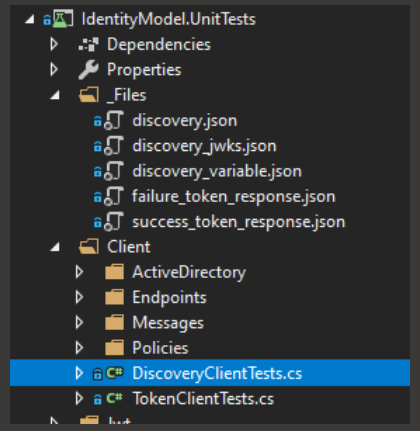
Há N exemplos disso nos testes de integração disponíveis em Rest.IntegrationTests (ver, por exemplo, ServiceClientTestsAuthFixture).

Embedded resources

Outra técnica muito aplicada nos testes do Hydrogen é a de armazenar os "resultados esperados" dos testes em embedded resources na mesma assembly do teste. Nomeadamente, quando esses resultados são ficheiros mesmo ou então são mais fáceis de manipular como ficheiros (exemplo: JSON extensos).

O serviço EmbeddedResourcesService facilita o carregamento do resource.

Exemplo:



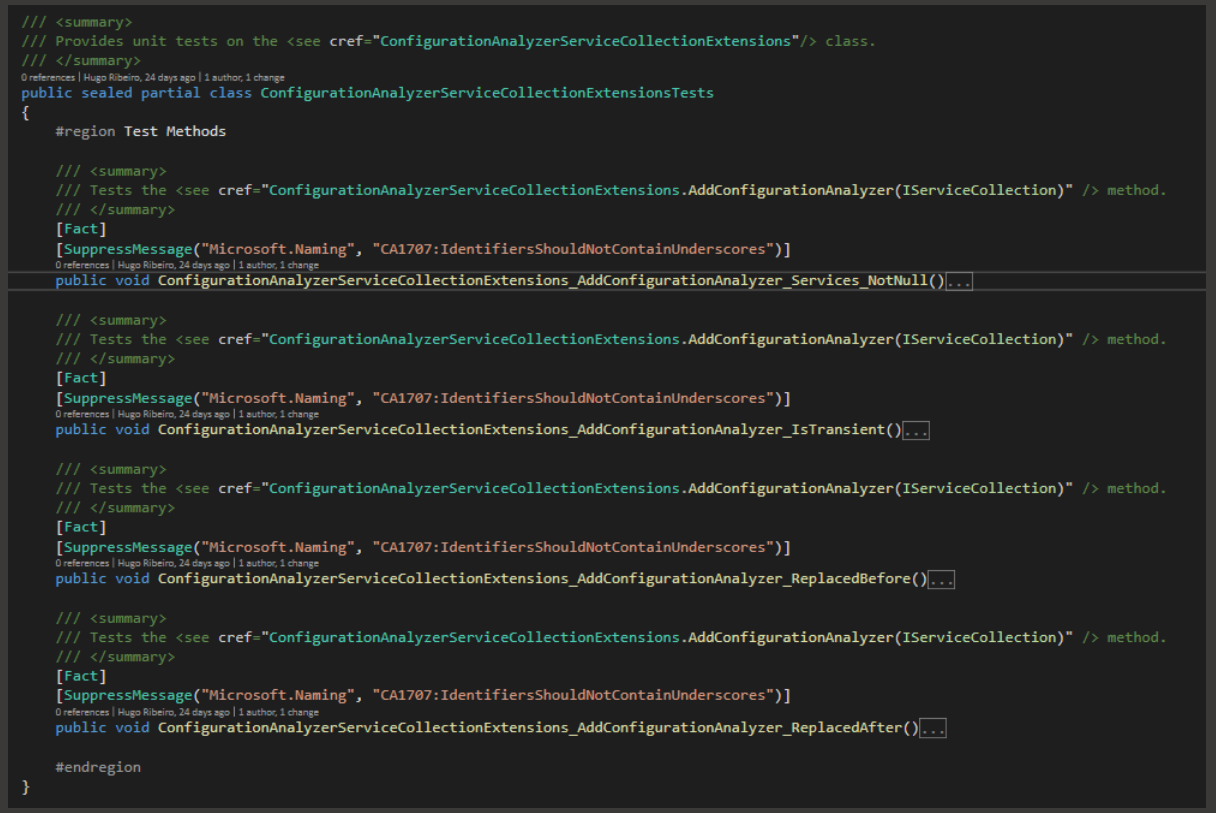

```
/// <summary>
/// Tests the <see cref="DiscoveryClient.DiscoveryClient(HttpMessageHandler, bool)" /> constructor.
/// </summary>
/// <returns>
/// The <see cref="Task"/> that represents the asynchronous operation.
/// </returns>
[Fact]
[Trait(TraitName, TraitValue)]
[SuppressMessage("Microsoft.Naming", "CA1707:IdentifiersShouldNotContainUnderscores")]
0 references | Hugo Ribeiro, 136 days ago | 1 author, 1 change | 1 incoming change
public async Task DiscoveryClient_Constructor3_Disposed()
{
    string disco = EmbeddedResourcesService.ReadText<DiscoveryClientTests>("Primavera.Hydrogen.IdentityModel.UnitTests._Files.discovery.json");

    using MockHttpMessageHandler handler = MockHttpMessageHandler.ReturnContents(HttpStatusCode.OK, disco);

    DiscoveryClient client1 = null;
    using (DiscoveryClient client2 = new DiscoveryClient(handler, true))
    {
        client1 = client2;
    }

    Func<Task> action = () => client1.GetDocumentAsync("https://demo.identityserver.io");
    await action.Should().ThrowAsync<ObjectDisposedException>().ConfigureAwait(false);
}
```

Dependency injection

A DI tem 2 relações com o tema dos testes ao Hydrogen:

- Sendo os componentes desenhados para serem usados via DI, esse é um aspeto crucial que precisa de ser muito bem testado.
- O facto de serem desenhados assim facilita por si a construção dos testes sobre as suas funcionalidades. Ou seja, a DI facilita imenso a injeção de mocks sobre as dependências dos serviços.

A primeira questão deve ser assegurada pelos testes realizados sobre as classes que contêm os métodos de extensão para registo dos serviços. É fundamental, verificar o registo, o lifetime do serviço, a aplicação das opções de configuração, etc. Particularmente por razões de compatibilidade futura.



```
/// <summary>
/// Provides unit tests on the <see cref="ConfigurationAnalyzerServiceCollectionExtensions"/> class.
/// </summary>
0 references | Hugo Ribeiro, 24 days ago | 1 author, 1 change
public sealed partial class ConfigurationAnalyzerServiceCollectionExtensionsTests
{
    #region Test Methods

    /// <summary>
    /// Tests the <see cref="ConfigurationAnalyzerServiceCollectionExtensions.AddConfigurationAnalyzer(IServiceCollection)" /> method.
    /// </summary>
    [Fact]
    [SuppressMessage("Microsoft.Naming", "CA1707:IdentifiersShouldNotContainUnderscores")]
    0 references | Hugo Ribeiro, 24 days ago | 1 author, 1 change
    public void ConfigurationAnalyzerServiceCollectionExtensions_AddConfigurationAnalyzer_Services_NotNull()...

    /// <summary>
    /// Tests the <see cref="ConfigurationAnalyzerServiceCollectionExtensions.AddConfigurationAnalyzer(IServiceCollection)" /> method.
    /// </summary>
    [Fact]
    [SuppressMessage("Microsoft.Naming", "CA1707:IdentifiersShouldNotContainUnderscores")]
    0 references | Hugo Ribeiro, 24 days ago | 1 author, 1 change
    public void ConfigurationAnalyzerServiceCollectionExtensions_AddConfigurationAnalyzer_IsTransient()...

    /// <summary>
    /// Tests the <see cref="ConfigurationAnalyzerServiceCollectionExtensions.AddConfigurationAnalyzer(IServiceCollection)" /> method.
    /// </summary>
    [Fact]
    [SuppressMessage("Microsoft.Naming", "CA1707:IdentifiersShouldNotContainUnderscores")]
    0 references | Hugo Ribeiro, 24 days ago | 1 author, 1 change
    public void ConfigurationAnalyzerServiceCollectionExtensions_AddConfigurationAnalyzer_ReplacedBefore()...

    /// <summary>
    /// Tests the <see cref="ConfigurationAnalyzerServiceCollectionExtensions.AddConfigurationAnalyzer(IServiceCollection)" /> method.
    /// </summary>
    [Fact]
    [SuppressMessage("Microsoft.Naming", "CA1707:IdentifiersShouldNotContainUnderscores")]
    0 references | Hugo Ribeiro, 24 days ago | 1 author, 1 change
    public void ConfigurationAnalyzerServiceCollectionExtensions_AddConfigurationAnalyzer_ReplacedAfter()...

    #endregion
}
```

Dependency injection

O segundo aspeto é evidente em centenas de testes existentes no Hydrogen.

Eis um exemplo bastante simples:

```
/// <summary>
/// Tests the <see cref="RecaptchaService.ValidateAsync(RecaptchaSettings, string, string, CancellationToken)" /> method.
/// </summary>
/// <returns>
/// The asynchronous task.
/// </returns>
[Fact]
[SuppressMessage("Microsoft.Naming", "CA1707:IdentifiersShouldNotContainUnderscores")]
0 references | Hugo Ribeiro, 29 days ago | 1 author, 2 changes
public async Task RecaptchaService_ValidateAsync4_WithoutRemoteIp()
{
    using MockHttpClientHandler handler = new MockHttpClientHandler();

    IRecaptchaService service = GetService(true, handler);

    RecaptchaResponse response = await service
        .ValidateAsync(
            new RecaptchaSettings()
            {
                SecretKey = "secretkey",
                UseRemoteIp = false
            },
            "122.122.122.122",
            "1001")
        .ConfigureAwait(false);

    response.Should().NotBeNull();
    response.Success.Should().BeTrue();
    response.Score.Should().Be(0.4m);
    response.Action.Should().Be("action");
    response.ChallengeTimestamp.Should().Be(new DateTime(2019, 1, 31));
    response.HostName.Should().Be("myserver");

    handler.LastSecret.Should().Be("secretkey");
    handler.LastResponse.Should().Be("1001");
    handler.LastRemoteIp.Should().BeNull();
}
```

```
25 references | Hugo Ribeiro, 29 days ago | 1 author, 1 change
private static IRecaptchaService GetService(bool mockHttpClientFactory, MockHttpClientHandler handler = null)
{
    // Build service collection

    ServiceCollection services = new ServiceCollection();

    // Add HTTP client factory?

    if (mockHttpClientFactory)
    {
        services
            .AddTransient<IHttpClientFactory>(
                (provider) =>
                {
                    return new MockHttpClientFactory(handler);
                });
    }

    // Add service

    services.AddRecaptcha();

    // Build service provider

    IServiceProvider serviceProvider = services.BuildServiceProvider();

    // Result service

    return serviceProvider.GetRequiredService<IRecaptchaService>();
}
```

Mocks

Há várias técnicas para “aplicar mocks” às dependências do tipo que é alvo de teste.

A descrita no slide anterior é a mais simples mas pode utilizar-se também a biblioteca Moq (referenciada em Primavera.Hydrogen.DesignTime.UnitTesting).

```
/// <summary>
/// Tests the <see cref="IEventBusService.Unsubscribe{T}(string)"/> method.
/// </summary>
[Fact]
0 references | Ivo Gomes, 2 days ago | 1 author, 1 change | 7 work items
public void InMemoryEventBus_SingleThreaded_Unsubscribe_ShouldRemoveSubscription()
{
    InMemoryEventBusOptions options = new InMemoryEventBusOptions(InMemoryEventBusStrategy.SingleThreaded);

    using IEventBusService eventBus = new InMemoryEventBusService(options);

    Mock<IEventBusEventHandler<string>> handler = MocksService.BuildMockOf<IEventBusEventHandler<string>>();
    handler.Setup(o => o.Handle(It.IsAny<IEventBusEvent<string>>())).Returns(() =>
    {
        return Task.FromResult(true);
    });

    eventBus.Subscribe(Constants.Paths.Tests, handler.Object);

    Action action = () => eventBus.Unsubscribe<string>(Constants.Paths.Tests);

    action.Should().NotThrow<EventBusServiceException>();
}
```

Há situações em que este tipo de construções é muito eficaz e torna os testes mais simples e/ou fáceis de analisar.

Boas práticas

Para além dos vários aspetos que são enumerados antes, devem observar-se as seguintes boas práticas genéricas:

1. Cada um dos tipos do Hydrogen deve ter sempre pelo menos 1 teste.
2. Devem preferenciar-se os testes unitários aos testes de integração.
3. Devem fazer-se testes de integração sempre que isso for útil.
4. Os testes devem ser o mais simples possível.
5. Os testes devem executar o mais rápido possível.
6. Os testes devem cobrir o máximo de código possível (de cenários, portanto).
7. Qualquer teste que requeira interação com um recurso externo (seja de que tipo for) deve ser considerado de integração.
8. Os testes devem ser resilientes à cultura em que são executados (tipicamente na máquina de desenvolvimento a cultura é PT e nas máquinas de builds é EN) (ver classes base de teste que permitem fixar a cultura do teste, para os casos mais extremos).