Dealing with breaking changes during the migration of an application to .NET 8

Nagib Sabbag Filho
FIAP (Faculty of Informatics and Administration Paulista) Avenida Paulista, 1106 - 7º andar - Bela Vista, São Paulo, Brazil.
e-mail: profnagib.filho@fiap.com.br
PermaLink: https://leaders.tec.br/article/90a9d8
jul 08 2024

Abstract:
Dealing with breaking changes during the migration to .NET 8: strategies to ensure a smooth transition.

Key words:
migration, breaking changes, .net 8, compatibility, apis, tests.

Understanding Breaking Changes

Known as breaking changes, these are modifications in the code that can cause incompatibilities with the previous functioning of the application. During the migration to .NET 8, it is important to identify and handle these changes efficiently to ensure that the application continues to function correctly. They can include:

1. API Changes: Alterations in method signatures, parameters, or properties that impact existing usage. For example, the removal of a widely used method or a change in a method signature that requires new parameters.

2. Different Behavior: Modifications in the behavior of functions or classes that can break existing workflows. For example, a function that previously returned a default value may now throw an exception in certain scenarios.

3. Deprecation of Features: Removal or discontinuation of obsolete features or functionalities. This can include methods, classes, or even entire libraries that are no longer supported in the new version.

Main Challenges During Migration to .NET 8

One of the main challenges during the migration to .NET 8 is the need to update libraries and frameworks used by the application. It is common for new versions of .NET to bring significant changes that can directly impact the functioning of existing code. Additionally, the documentation may not be complete or up to date, which can make it difficult to identify all necessary changes.

Another challenge is ensuring that all third-party dependencies are compatible with the new version. This may require updating or replacing NuGet packages and other external libraries.

Strategies for Dealing with Breaking Changes

One of the most effective strategies for dealing with breaking changes during the migration to .NET 8 is to conduct comprehensive regression testing. This includes running automated and manual tests to identify potential issues caused by changes in the code. It is important to have good test coverage to ensure that all critical parts of the application are checked.

Another strategy is to use code analysis tools to identify breaking changes. Tools like the .NET Portability Analyzer can help detect incompatibilities between different versions of .NET.

Additionally, it is advisable to create a detailed migration plan that includes all necessary steps to update the

application. This may include updating libraries, modifying code for compatibility with .NET 8, and conducting extensive testing to ensure everything works correctly.

Practical Example of Unit Tests Acting as Regression Tests

Suppose that during the migration to .NET 8, a change in the behavior of an encryption function causes incompatibilities with the rest of the application. In this case, it is important to review the code of the affected function and adjust it according to the new guidelines of .NET 8. For example, if an encryption method has been updated to use a new algorithm by default, you may need to adjust the code to specify the old algorithm if necessary.

For example... Below we have a C# code providing an encryption solution

```
using System;
using System.Security.Cryptography;
using System.Text;

public class CryptographyService
{
    private readonly string key = "example_key_1234";

    public string Encrypt(string plainText)
    {
        using (Aes aes = Aes.Create())
        {
            byte[] keyBytes = Encoding.UTF8.GetBytes(key);
            aes.Key = keyBytes;

            ICryptoTransform encryptor = aes.CreateEncryptor(aes.Key, aes.IV);
            using (var ms = new MemoryStream())
            {
                ms.Write(aes.IV, 0, aes.IV.Length);
                using (var cs = new CryptoStream(ms, encryptor,
CryptoStreamMode.Write))
                using (var sw = new StreamWriter(cs))
                {
                    sw.Write(plainText);
                }
                return Convert.ToBase64String(ms.ToArray());
            }
        }
    }

    public string Decrypt(string cipherText)
    {
        byte[] fullCipher = Convert.FromBase64String(cipherText);
        using (Aes aes = Aes.Create())
        {
            byte[] iv = new byte[aes.BlockSize / 8];
            byte[] cipher = new byte[fullCipher.Length - iv.Length];

            Array.Copy(fullCipher, iv, iv.Length);
            Array.Copy(fullCipher, iv.Length, cipher, 0, cipher.Length);
```

```
            aes.Key = Encoding.UTF8.GetBytes(key);
            aes.IV = iv;

            ICryptoTransform decryptor = aes.CreateDecryptor(aes.Key, aes.IV);
            using (var ms = new MemoryStream(cipher))
            using (var cs = new CryptoStream(ms, decryptor, CryptoStreamMode.Read))
            using (var sr = new StreamReader(cs))
            {
                return sr.ReadToEnd();
            }
        }
    }
}
```

Below is an example of a regression test for the encryption method using the XUnit framework:

```
using System;
using Xunit;

public class CryptographyServiceTests
{
    private readonly CryptographyService _cryptographyService;

    public CryptographyServiceTests()
    {
        _cryptographyService = new CryptographyService();
    }

    [Fact]
    public void EncryptDecrypt_ShouldReturnOriginalString()
    {
        // Arrange
        string originalText = "Hello, World!";

        // Act
        string encryptedText = _cryptographyService.Encrypt(originalText);
        string decryptedText = _cryptographyService.Decrypt(encryptedText);

        // Assert
        Assert.Equal(originalText, decryptedText);
    }

    [Fact]
    public void Encrypt_ShouldReturnDifferentString()
    {
        // Arrange
        string originalText = "Hello, World!";

        // Act
        string encryptedText = _cryptographyService.Encrypt(originalText);
```

```
        // Assert
        Assert.NotEqual(originalText, encryptedText);
    }

    [Fact]
    public void Decrypt_InvalidData_ShouldThrowException()
    {
        // Arrange
        string invalidCipherText = "invalid_cipher_text";

        // Act & Assert
        Assert.Throws(() => _cryptographyService.Decrypt(invalidCipherText));
    }
}
```

Another very useful point would be to consult the official pages. For example, the page Breaking Changes in .NET 8 helps us better understand this scenario, presenting all the points that need to be addressed, in topics:

Asp.NET Core: Changes in controllers, views, and middleware that may affect the behavior of the web application.
Containers: Updates in configurations and support for Docker and Kubernetes containers.
Main .NET Libraries: Changes in common libraries such as System.IO, System.Net, and others that may impact various parts of the application.
Cryptography: Changes in algorithms and encryption methods that may affect the security and compatibility of encrypted data.
Deployment: New practices and tools for deployment that may require adjustments in the CI/CD pipeline.
Entity Framework Core: Updates in the ORM that may affect entity mapping and database queries.
Other considerably relevant topics: Includes performance improvements, security, and new features that may require adaptations in the code.

Conclusion

Dealing with breaking changes during the migration to .NET 8 can be challenging, but with the right strategies and proper planning, it is possible to minimize impacts and ensure a smooth transition. Stay updated with new versions of .NET and always be aware of changes that may affect your application. Additionally, consider participating in developer communities and forums to share experiences and get support during the migration process.

References

Breaking Changes in .NET 7. Available at https://learn.microsoft.com/en-us/dotnet/core/compatibility/7.0;
Breaking Changes in .NET 8. Available at https://learn.microsoft.com/en-us/dotnet/core/compatibility/8.0.

Nagib is a University Professor and Tech Manager. He has a track record of achievements in technical and agile certifications, including MCSD, MCSA, and PSM1. He holds a postgraduate degree in IT Management from SENAC and an MBA in Software Technology from USP. Nagib has completed extension programs from MIT and the University of Chicago. Other achievements include the authorship of a peer-reviewed article on chatbots, presented at the University of Barcelona.