Modernizing .NET Web Applications

Everything You Need to Know About Migrating ASP.NET Web Applications to the Latest Version of .NET

Tomáš Herceg

Apress®

Modernizing .NET Web Applications

Everything You Need to Know About Migrating ASP.NET Web Applications to the Latest Version of .NET

Tomáš Herceg

Apress®

Modernizing .NET Web Applications: Everything You Need to Know About Migrating ASP.NET Web Applications to the Latest Version of .NET

Tomáš Herceg
Praha, Czech Republic

ISBN-13 (pbk): 979-8-8688-0616-2 ISBN-13 (electronic): 979-8-8688-0617-9
https://doi.org/10.1007/979-8-8688-0617-9

Managing Director, Apress Media LLC: Welmoed Spahr
Acquisitions Editor: Ryan Byrnes
Development Editor: Laura Berendson
Editorial Project Manager: Gryffin Winkler

Cover designed by eStudioCalamar

Cover Photo by Sandro Katalina on Unsplash (unsplash.com)

Distributed to the book trade worldwide by Springer Science+Business Media New York, 1 New York Plaza, Suite 4600, New York, NY 10004-1562, USA. Phone 1-800-SPRINGER, fax (201) 348-4505, e-mail orders-ny@springer-sbm.com, or visit www.springeronline.com. Apress Media, LLC is a California LLC and the sole member (owner) is Springer Science + Business Media Finance Inc (SSBM Finance Inc). SSBM Finance Inc is a **Delaware** corporation.

For information on translations, please e-mail booktranslations@springernature.com; for reprint, paperback, or audio rights, please e-mail bookpermissions@springernature.com.

Apress titles may be purchased in bulk for academic, corporate, or promotional use. eBook versions and licenses are also available for most titles. For more information, reference our Print and eBook Bulk Sales web page at http://www.apress.com/bulk-sales.

Any source code or other supplementary material referenced by the author in this book is available to readers on GitHub. For more detailed information, please visit https://www.apress.com/gp/services/source-code.

If disposing of this product, please recycle the paper

Table of Contents

About the Author

Tomáš Herceg lives in the Czech Republic, and has been a Microsoft MVP since 2009. He runs a software consulting company called RIGANTI, and he has founded DotVVM, an open-source framework for building web apps using a popular MVVM design pattern. He often speaks at conferences and user groups, and he is the founder of Update Conference, the largest .NET developer event in the Czech Republic.

Tomáš spent a large part of his career helping his customers with technical decisions, such as cloud migration, microservices architecture, Domain-Driven Design, and modernization of large legacy applications. He wrote many technical articles about .NET development and conducted numerous courses and hands-on labs. Writing this book seemed like the next step forward.

About the Technical Reviewer

Joel Lopes is a Staff Software Engineer at Cruise LLC, where he specializes in creating cloud services for autonomous vehicle maps and routing with a strong emphasis on scalability, reliability, and availability. He is a senior member of IEEE, a Fellow of BCS, and a Fellow of IETE and has been honored with several awards for his significant contributions to the industry. Before joining Cruise, Joel accrued over a decade of experience architecting robust and secure software at major cloud providers like Microsoft and Google. He conceptualized YAML configuration as code for Azure Engineering Systems on top of Azure DevOps release and led release systems instrumental in developing private clouds such as Azure Government Cloud, Azure Germany, Azure China, Azure Secret Cloud, and Azure Top Secret Cloud. He also played a crucial role in developing hybrid cloud solutions for Azure's core build system. Joel holds a master's degree in Computer Science from the University of Missouri, Kansas City, and a bachelor's degree from the University of Mumbai.

Acknowledgments

On this place, I would like to express my gratitude to several people without whom this book would not exist.

To Mehdi, my dear husband, who supported me when I came up with the crazy idea of writing a book and every single day of this eight-month-long adventure.

To my family, who encouraged and helped me with whatever I chose to do and who always respected my decisions even when they had different opinions. Nothing is more valuable to me than accepting someone for who they are, and they succeeded in doing so for my entire life.

To Wolfgang Amadeus Mozart, whose Magic Flute, Don Giovanni, and other phenomenal operas were playing in my headphones most of the time while I was writing, researching, and thinking.

To Michal Altair Valášek, who helped me with the security parts of the book and gave me a lot of useful advice on getting the ASP.NET Identity examples right. Also, he was the person who taught me ASP.NET Web Forms at the beginning of my career. I remember every one of the sessions he presented in the Aquarius meeting room of the former Microsoft building in Prague. I was still in high school, and every month, with my fresh driver's license, I drove almost 200 kilometers to attend his session at 6 PM and another 200 kilometers to get back home before midnight to be ready for school I had to attend the next day.

To Štěpán Bechynský and Dalibor Kačmář, two amazing people from Microsoft Czech Republic, who convinced me to apply to study at Charles University even when I thought I was not going to finish it. Although I quite quickly confirmed my expectations that most of the knowledge taught in the lectures would not be directly useful or applicable to what I wanted

to do, I did not see the importance of the foundations and principles on which computer science stands, and this is the most valuable thing the university taught me. The frameworks and libraries change, but the principles remain.

To Roman Jašek, who reviewed parts of the text while we were traveling to attend the MVP Summit in Seattle and provided valuable feedback.

To Stanislav Lukeš, the most active contributor to the DotVVM project, without whom the project would never be more than a simple conference demo. In 2014, I published an initial prototype on GitHub and briefly mentioned it during one of my sessions at the MS Fest community conference. The same day, someone with a nickname exyi submitted a pull request to the repository and continued doing that ever since. Thank you for this amazing journey.

To people at Apress, who reached out to me with the idea of writing a book and helped me throughout the entire process. To my technical reviewer, who pointed out numerous suggestions on how to make the book text and examples better.

And finally, to all the people at RIGANTI, my day-to-day inspiration, who passionately work on our projects and deal with all the technical challenges we encounter. You help me learn new things and keep my mind open to trying new ways and concepts. You are the most amazing group of people to work with.

CHAPTER 1

Introduction

Every keynote of the annual Microsoft Build conference has a slide about how many developers are missing in the IT industry. Even with the rise of artificial intelligence and the introduction of tools like GitHub Copilot or ChatGPT, there are not enough developers to build and maintain applications. Even though these tools are improving every day and can help with the lack of workforce in our industry, I think the situation will not change anytime soon. Faster than ever, the requirements for new applications, digital transformation, and automation of manual processes create new demand for new software.

Try to look at any state administration; most of their activities still involve moving hand-filled forms or manual transfer of information from one system to another. Making this digital and automated requires a tremendous amount of work for solution architects, system integrators, and developers.

More and more lines of code are written by the developers or generated by AI-powered tools, and all this code needs to be maintained, often for years. Many companies and developer teams struggle to keep their projects updated to the new versions of libraries and frameworks, even when the versions being used contain security vulnerabilities. The worldwide deficit in the number of developers forces businesses and organizations to prioritize delivering new features rather than keeping the codebase in perfect condition.

T. Herceg, *Modernizing .NET Web Applications*,
https://doi.org/10.1007/979-8-8688-0617-9_1

The significance of this problem rises dramatically when the new version of the developer platform brings significant breaking changes. This is how technological debt emerges and grows.

The New Generation of .NET

A significant step in accumulating technological debt was created when Microsoft introduced .NET Core 1.0 in 2016. It was not just another update of .NET; a more precise term would be the *new generation*. The .NET Core and its subsequent versions, which are called just .NET, contain plenty of improvements in terms of performance, application architecture, and maintainability. However, they come with a great deal of challenges when you want to migrate to them from any previous version.

Table 1-1 shows the releases of .NET Framework and .NET Core, later renamed to .NET. You can see that .NET 5 was the successor of .NET Core 3.1. In the rest of this book, for any statement about .NET Core without a specific version, you can safely assume it is also valid for .NET 5, 6, 7, and subsequent releases from the .NET Core branch. I am going to refer to it also as "the new .NET".

You can notice that .NET Framework occasionally gets new versions, too. Since .NET Framework is an essential component of Windows, its release cycle and support are bound to the Windows life cycle.[1] However, Microsoft officially states that this branch of .NET will only be getting security fixes in the future, and all the new development will be done to the .NET Core branch. .NET Framework is undoubtedly not going away in the upcoming years; many Microsoft products (including Visual Studio and various components of Windows) are built using the .NET Framework.

[1] See https://dotnet.microsoft.com/en-us/platform/support/policy/dotnet-framework

Table 1-1. *Versions and release dates of .NET Framework and .NET Core*

.NET Framework Branch	Release Date	.NET Core Branch	Release Date
.NET Framework 4.6.2	August 2016	.NET Core 1.0	June 2016
		.NET Core 1.1	November 2016
.NET Framework 4.7	April 2017	.NET Core 2.0	August 2017
.NET Framework 4.7.1	October 2017	.NET Core 2.1	May 2018
.NET Framework 4.7.2	April 2018	.NET Core 2.2	December 2018
.NET Framework 4.8	April 2019	.NET Core 3.0	September 2019
		.NET Core 3.1	December 2019
		.NET 5	November 2020
		.NET 6	November 2021
.NET Framework 4.8.1	August 2022	.NET 7	November 2022
		.NET 8	November 2023

Bridging the Gap with .NET Standard

.NET is not just a runtime for running C#, VB.NET, and F# code. It is also a rich set of libraries, API, and, most importantly, complete frameworks for creating web, desktop, or mobile applications.

Because the API surface of these two branches differed and Microsoft wanted to allow easier code sharing between them, .NET Standard was introduced. While both .NET Framework and .NET Core are runtimes accompanied by a collection of libraries, .NET Standard is entirely different. You can think of it only as a list of APIs, something like a contract to be implemented by some runtime. Instead of using Portable Class

Libraries,[2] you can create a standard Class Library project that targets a specific version of .NET Standard. Thanks to this, the library is guaranteed to interact only with the APIs present in the .NET Standard, and thus the same DLL can be used with .NET Framework and .NET Core versions that implement this standard. If you try to call an API that is not part of .NET Standard, the compiler will produce an error.

.NET Standard ceased to fully serve its purpose since its version 2.1 was not implemented by any version of .NET Framework. Thus, the latest version of .NET Standard that is useful for modernization techniques described in this book is .NET Standard 2.0, which is implemented by .NET Framework 4.6.2+[3] and .NET Core 2.0+. Table 1-2 shows the platforms' compliance with various versions of .NET Standard.

Table 1-2. *.NET Framework and .NET Core compliance with .NET Standard*[4]

.NET Standard Version	Compatible .NET Framework Version	Compatible .NET Core Version
.NET Standard 1.6	.NET Framework 4.6.2+	.NET Core 1.0+
.NET Standard 2.0	.NET Framework 4.6.2+	.NET Core 2.0+
.NET Standard 2.1	-	.NET Core 3.0+

[2] Portable Class Library was an older concept that allowed compiling the same code to target multiple platforms. See https://learn.microsoft.com/en-us/xamarin/cross-platform/app-fundamentals/pcl for more information.

[3] Technically, .NET Framework 4.6.1 claims to implement .NET Standard 2.0, but there are several compatibility issues in various NuGet packages. It is recommended to upgrade to a newer version.

[4] See https://learn.microsoft.com/en-us/dotnet/standard/net-standard for a complete listing of supported runtimes with all versions of .NET Standard.

What Has Changed

There were numerous changes in the Base Class Library[5] itself, but these don't create significant issues when migrating the code from .NET Framework to .NET Core.

In most cases, the edits in the code are only cosmetic; for example, sometimes you must use a different overload of some method or another constructor. For instance, in .NET Framework 4.7.2, the `ToHashSet` extension method gets an argument of type `IEqualityComparer`, but this overload is missing in the .NET Standard 2.0. If you migrate the code directly to .NET Core 2.0 or higher, you will not even notice this, as the same overload is present in .NET Core. However, a small code change will be necessary when the code is moved into a class library targeting .NET Standard, as shown in Listing 1-1.

Listing 1-1. An example of overload supported in .NET Framework and .NET Core but missing from .NET Standard 2.0

```
IEnumerable<string> names = new[] { "ABC", "abc" };

// does not work in .NET Standard 2.0
var uniqueNames = names.ToHashSet(
    StringComparer.CurrentCultureIgnoreCase);

// works in .NET Standard 2.0
var uniqueNames = new HashSet<string>(names,
    StringComparer.CurrentCultureIgnoreCase);
```

[5] The Base Class Library is the basic set of APIs included in .NET. It contains the built-in data types to represent strings, numbers, and dates; it contains collections, APIs to work with input and output, globalization and culture-based formatting of values, representing text in various encodings like UTF-8, and more. https://en.wikipedia.org/wiki/Standard_Libraries_(CLI)#Base_Class_Library

Some APIs in .NET Core have slightly different behavior, often because of added support for Linux and Mac OS. This is the case of `UseShellExecute` in the `ProcessStartInfo` class.[6] Since some .NET areas are just a thin wrapper over the Windows API, you can find methods that throw `PlatformNotSupportedException` on non-Windows platforms because it is not possible to implement such APIs.

Furthermore, some APIs like Code Access Security or Remoting were removed completely; luckily, they were not used often.

The situation is dramatically different when it comes to application frameworks like ASP.NET MVC, Entity Framework, WPF, and almost all others that are part of .NET. Some of these frameworks were available even in the first version of the .NET Core, albeit with a handful of breaking changes. This is the case with the previously mentioned ASP.NET MVC and ASP.NET Web API.

Others, such as Windows Forms and WPF, were added later, namely, in .NET Core 3.0. They also introduced numerous incompatibilities, especially in the design-time experience in Visual Studio, but they are officially supported.

Several libraries, for example, the Entity Framework and ASP.NET Identity, were completely reimplemented, heavily changed, and renamed to Entity Framework Core and ASP.NET Core Identity.

Finally, some frameworks have no alternative in the new .NET Core world. This is the case of WCF (Windows Communication Foundation) which is available for the new platform only as an open source community–maintained project called *CoreWCF*. An even bitter story is the story of ASP.NET Web Forms, which was so challenging to reimplement

[6] The differences in the shell execute behavior are described in the documentation page at https://learn.microsoft.com/en-us/dotnet/core/compatibility/fx-core#change-in-default-value-of-useshellexecute

for .NET Core without introducing significant breaking changes that it was decided not to support this technology on the new platform at all. There is no similar framework available on the .NET Core. Thus, migrating a Web Forms application necessarily means rewriting all pages and UI components to use a different presentation technology.

Even though the programming language remained the same on the old and the new versions, all the frameworks have been modified or rethought to work in the new era of cloud and distributed applications. There is clearly more focus on performance, removing unnecessary robustness, and making things more lightweight (as you can see, for instance, in the Minimal API approach introduced in .NET 6). Much work has also been done to get rid of tight coupling with operating system components like IIS (Internet Information Services). The old ASP.NET depended heavily on the infrastructure brought by IIS, while the new ASP.NET Core is completely decoupled from the web server.

Another excellent example of this drift is Entity Framework. The querying mechanism in Entity Framework 6 was very robust and designed for databases with hundreds or even thousands of tables. The startup time and the time to the first query were not the most critical metric because most workloads were server applications running for days or weeks, and the cadence of releases was significantly lower in the days before the rise of DevOps. Restarts of such applications were infrequent and did occur several times a month at most. Business-critical applications were typically running in clusters of several physical or virtual machines. In contrast, when you look at the Entity Framework Core, its first versions supported only basic queries, and the focus of the team was split between excellent query performance and providing a pleasant experience when working with smaller database models that suit the modern microservices approach. This approach is more fruitful in cloud environments where the application running in containers can be stopped and started many times a day, either because of frequent releases or various auto-scaling configurations where the underlying infrastructure moves the application

instances between servers. You can also notice a shift in the EF team's attention to a more expressive API for defining the database model in the code, clearly inspired by favorite approaches like Domain-Driven Design. Even the most recent versions of Entity Framework Core do not beat the power of the query translator in the old Entity Framework 6. However, as a developer, you have more convenient ways to express how your database model will look and how it will be mapped to the database schema.

The new .NET Core was definitely a great step forward, and practically all application frameworks that are part of today's .NET ecosystem were significantly improved to better suit today's requirements for modern applications. However, in most cases, it is extremely difficult to take the codebase of a project targeting .NET Framework and run it on the new .NET. The upgrade process, often referred to as *modernization*, usually requires weeks or months of planning, code editing, and testing.

Ten Thousand Feet View of the Book

Each chapter of this book looks at different aspects and stages of this modernization challenge, explains the differences between the old and new, and offers advice or examples of the decision that needs to be made and steps that need to be taken.

In this chapter, I will try to give an overview of the changes that happened in various application frameworks between .NET Framework and .NET Core and lay out the high-level migration steps to be taken.

Chapter 2 will discuss the most common reasons for modernization, including the nontechnical ones such as explaining the benefits to company management and other stakeholders. I will also compare the complete rewrite approach with the incremental modernization and discuss their pros and cons.

In Chapter 3, I will discuss the steps to take before you start the modernization journey. I will talk about side-by-side migration using YARP (Yet Another Reverse Proxy) and compare it with an in-place approach that can be done using an open source framework called DotVVM. Dealing with change requests and bug fixes during the modernization will also be one of the crucial topics discussed in the chapter.

Chapter 4 will focus on the communication frameworks: WCF, ASP.NET SignalR, and old ASP.NET Web Services. I will show their equivalents available in the .NET Core world: ASP.NET Core SignalR, CoreWCF, and SoapCore open source projects. I will also show how to use an `HttpContext` adapter package to prepare the codebase for migration to the new .NET.

In Chapter 5, I will explain how to migrate data access code. Starting with the classic ADO.NET approach, where not many things have changed, I'll spend most of the time describing the migration journey for Entity Framework 6, which is the most favorite ORM in the .NET world. I will briefly mention other technologies, namely, LINQ to SQL, and show what code changes you can expect.

Chapter 6 will talk about migrating from legacy identity providers. I will begin with the old ASP.NET Membership and Role providers and describe how to migrate from ASP.NET Identity to its new version called ASP.NET Core Identity. Since the new library uses a different format of hashed passwords stored in the database, I will also show two strategies on how to migrate them safely: one will require all users to reset their passwords, and the other will store passwords in the old format in a safe way and migrate them to the new format on the first successful sign-in.

In Chapter 7, I will show a minimum-effort migration method of an ASP.NET Web Forms application using an open source MVVM framework called DotVVM. The reason for using DotVVM is that it is the only .NET-based web framework that supports both the old .NET Framework and the new .NET Core worlds, and so it can be used as a bridge

between these two worlds. I will also show how much easier it is to migrate applications where the business logic is well separated from the presentation code and how many things will have to be changed in applications where the business logic is spawned anywhere from the database to code-behind or even markup files.

Chapters 8 and 9 will focus on the most universal method of modernization of web applications – the side-by-side approach based on the YARP. In these chapters, I will explain how to create a new project for the migrated pages, how to configure the proxy to redirect the traffic to the legacy application for pages that have not been rewritten yet, and, most importantly, how to deal with all complications arising from the fact you are running two separate applications: creating the single sign-on experience for the two applications, synchronizing the session data, invalidating the cache on changes made in the other application, and more.

The last chapter will close the modernization topic with some examples and tips on separating business logic and keeping it independent of the presentation technology. This allows you to save a lot of effort if you face a similar "platform restart" as it happened with the arrival of .NET Core. In general, the web frameworks have a shorter life span than the back-end code; thus, it makes sense to be prepared for the future replacement of the presentation technology.

Get Ready

Naturally, no single path would suit all applications and teams. In each case, you will have to consider not only technical but also business and real-world aspects, which sometimes prove to be more challenging than changing the code itself. There are even cases when modernization does not make sense from the business perspective, especially when the project is not business-critical, and its long-term maintenance costs are negligible.

What makes this task even more challenging is that, most commonly, the developer team cannot dedicate all its capacity to the migration itself. Still, it must deliver new features and fix issues in the meantime. That is why modernization typically needs to be done in small incremental steps. An approach similar to A/B testing[7] can be involved to prevent a complete unavailability of essential functions or to be able to quickly retreat and return to the previous implementation in case of issues. You could see this approach in recent years in many consumer applications; there is a button to try the new experience, but you can always fall back to the old one. In combination with good telemetry, the developer team is able to deliver new functionality quickly and make sure it works seamlessly without risking that the users will not be able to do the things they want.

About the Author

It is always good to know something about the person whose text you are reading. Each of us has different backgrounds, worked for different companies, participated in projects of various sizes, talked to different customers and stakeholders, and hunted different bugs. Even though I have seen many software projects in various circumstances, indeed, I have not seen everything, and my worldview is thus incomplete. Feel free to confront the ideas you read in this book with your own experience and environment.

[7] A/B testing is an approach where the same functionality is implemented twice. Some percentage of users are forwarded to first implementation, while the rest use the second implementation. In combination with telemetry, you can evaluate which one performs better. In the case of modernization, you can do A/B testing by preserving the old implementation of a feature while rolling out its new version gradually.

I started programming in QBASIC when I was seven years old. I spent almost all my teenage life in front of the computer screen where you would most of the time see Turbo Pascal, Visual Basic 3 to 6, followed by Visual Studio .NET and all its subsequent versions: 2003, 2005, and so on.

I liked building games; the most complex was a 3D multiplayer strategy from the Antic Greece world, including even the AI player and a simple scripting language with an interpreter for implementing campaign missions. I also built numerous websites in PHP and ASP.NET Web Forms. For many of them, I got paid, which helped me study in Prague, the capital of the Czech Republic. With one of my friends, I created the largest website dedicated to programming in .NET in Czechia and published hundreds of articles and blog posts there. In 2009, I got my first Microsoft MVP award.

I got a bachelor's degree at Charles University in Prague. At first, I wasn't amazed by the school much. Although they gave me a solid theoretical background in computer science and algorithmic thinking, I wanted to learn more practical skills and get experience from real-world projects. That is why I quit the master's program, and in my incredible naivety, I founded a software consulting company called RIGANTI. I loved programming but didn't realize that as the CEO, I would hardly have time to write code. On the other hand, it allowed me to be part of hundreds of software projects and get the big picture of the day-to-day challenges of the developers without the need to spend thousands of hours on each one. I had the chance to see the struggles and complications of mobile development in Xamarin; I saw how much time it took to make a great web UI before jQuery, as well as in the modern JavaScript framework era. I got insights into the challenges of optimizing SQL indexes, hunting memory leaks caused by invalid registrations in a dependency injection container, and much more. I got invited to dig into the code many times, especially when my colleagues were dealing with some mysterious issue.

For most of the years, my job was not the day-to-day developer's work. However, I was present whenever important technological decisions were made. I helped design Azure-based solutions for global companies,

I got the opportunity to see the birth of innovative products in several startups, and even saw low-level programming of hardware devices in manufacturing companies.

Seeing these projects from the management level allowed me to better understand the business incentives and motivations of the stakeholders. I learned, often the hard way, how important it is to communicate with the developers and customers. Every single time, the technical issues we ran into could be solved in a matter of hours or days. Still, it was always the lack of communication that caused problems and broken relationships that were so difficult to restore.

One important category of projects coming to RIGANTI was legacy applications developed by someone else, and our task was to maintain these applications. In this kind of work, the developers always appreciate good architecture, especially the separation of the codebase into layers. Additionally, we found that it is good when the codebase is somehow predictable. No matter what architectural patterns or conventions are used, it is helpful when your intuition helps you navigate the code and find what you are looking for. With many projects, we didn't have this luxury; business logic was literally everywhere. In combination with non-existing or at least incomplete documentation, we clearly saw that even the tiniest changes in the application can be expensive to fix.

The company participated in several modernization projects and helped our customers fight their technological debt; this is how this book was born. It summarizes several years of my observations and experience. I sincerely hope that you find these insights helpful.

Application Framework Equivalents in the .NET Core World

The old .NET Framework introduced several application frameworks which are widely used in thousands of web applications. This section briefly introduces the modernization of applications using each of these frameworks.

One category of frameworks includes *ASP.NET Web Forms*, *ASP.NET MVC*, and *ASP.NET Web Pages*. All these technologies can be used to create web user interfaces, and at the end of the day, their main job is processing HTTP requests and rendering the corresponding HTML output.

The second category includes *ASP.NET XML Web Services*, *WCF (Windows Communication Foundation)*, *ASP.NET Web API*, and *ASP.NET SignalR*. These libraries were widely used to implement communication interfaces between multiple applications and often allowed .NET to interact with other platforms, such as Java and PHP.

A special category belongs to *LINQ to SQL* and *Entity Framework*, Object-Relational Mapper libraries widely used to retrieve and store information in SQL databases.

And finally, there is *ASP.NET Identity*, a universal library to implement identity, role management, and persistence of user accounts. This technology was often combined with *Entity Framework* as many developers prefer storing user information and application data in the same database.

Frameworks for Web UI

Let's start with the first category of presentation frameworks. The oldest of them is *ASP.NET Web Forms*, which tried to offer a similar level of simplicity in building user interfaces as we know from the world of desktop applications. A few years after its release, *ASP.NET MVC* was introduced to provide an alternative approach to building web applications. Finally, *ASP.NET Web Pages* allowed to strip off a part of MVC boilerplate code.

ASP.NET Web Forms

In ASP.NET Web Forms, the individual pages of the application were defined in `.aspx` markup files accompanied by code-behind files. The markup file contained an HTML code enhanced with the concept of server controls, page directives, and data-binding expressions. Whenever the server received an HTTP request, the hierarchy of server controls defined in the page was instantiated; the sequence of `Init`, `Load`, and `PreRender` life cycle events was called; and finally, the server controls rendered HTML, which could be interpreted by the browser. The entire page had to be one big `<form>` element, and any time the user clicked any button, the form was submitted to the server. This action was called the *postback*. The server extracted the values from the request body, which included a special hidden field called *View State*, which enabled Web Forms to restore the hierarchy of server controls to the exact state as it was at the end of the previous request. This concept helped to create a *stateful experience* on top of the stateless HTTP communication. The application logic was invoked, and a newly rendered HTML was sent back to the browser.

This process did not lead to the best user experience. When the users interacted with the page and caused the postback, they had to wait until the server produced a completely new HTML. It wasn't unusual that the page flickered or was unresponsive for a while, and the postback itself

could take a significant amount of time because of the size of the state-persisting hidden field. This was the primary reason why many .NET developers did not like this technology at all. In numerous cases, it was extremely difficult to keep the view state field reasonably small, especially on pages containing many controls or multiple sections.

Later, this concept was enhanced by ASP.NET AJAX, an extension to Web Forms, which used JavaScript to replace the full postback with an asynchronous AJAX call, which allowed the server to render just the part of the page that was expected to change. It was done by the `UpdatePanel` control. The user experience became significantly better, but the view state field remained the biggest problem.

After the first versions of .NET Core arrived, Microsoft announced that ASP.NET Web Forms was not going to be supported on .NET Core because it would be extremely difficult to avoid making significant breaking changes. Given the amount of existing Web Forms applications with hundreds or thousands of pages, the effort required to perform complete functional testing would probably discourage many people from even trying the upgrade. Additionally, ASP.NET Web Forms were tightly integrated with the IIS, so tightly that it was almost impossible to run ASP.NET Web Forms applications reliably in a different web server environment.

The Web Forms included several smaller frameworks, for example, the Membership, Role, and Profile providers. These technologies were predecessors of ASP.NET Identity and provided a useful abstraction over storing the user account information. Although the .NET Framework is still supported by Microsoft, the algorithms the Membership providers use to store passwords in the database are not considered secure by today's standards.

Another problem with this technology is that it tried to make things simple, and in some cases, it went a bit too far. Using components like `SqlDataSource` provided a quick way of implementing CRUD interfaces with a minimum amount of code, but it did not push the developers to