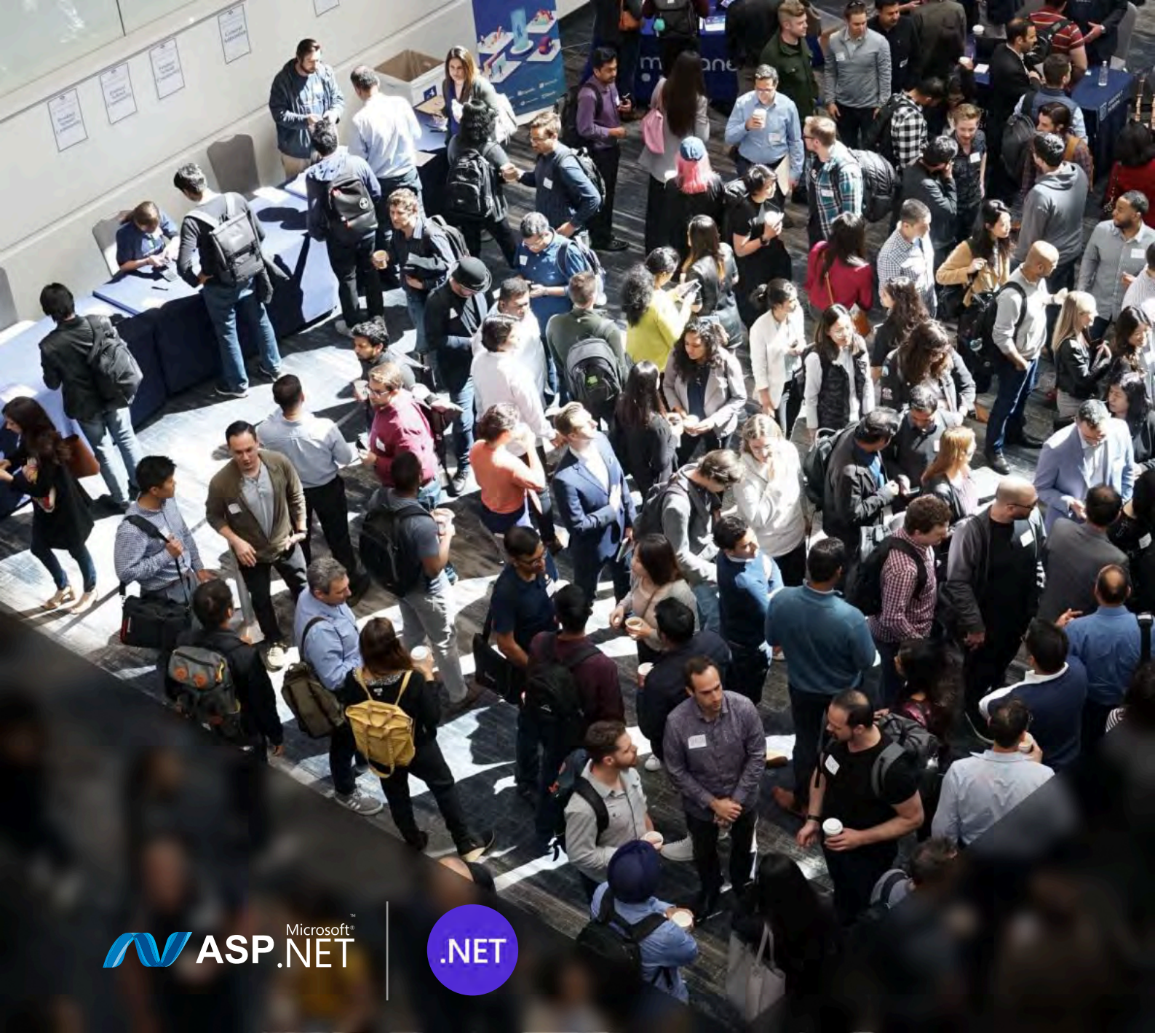


Migration Guide

ASP.NET Web Forms to Modern ASP.NET

Resolute

Introduction

If you've been paying attention to the .NET world of late, it would be hard not to notice that ASP.NET Web Forms development isn't as popular as it once was. There are several reasons for this, but the truth is that the technology is dated and doesn't offer many advantages over newer frameworks.

And, if you're like most developers and IT leaders, the idea of modernizing legacy ASP.NET Web Forms is a little intimidating, and it can be tough on your bottom line if things go wrong. But the reality is that not modernizing an existing application has some (not-so-obvious) drawbacks when it comes to competing in today's cloud-connected world. So how do you know when it's time to modernize your software?

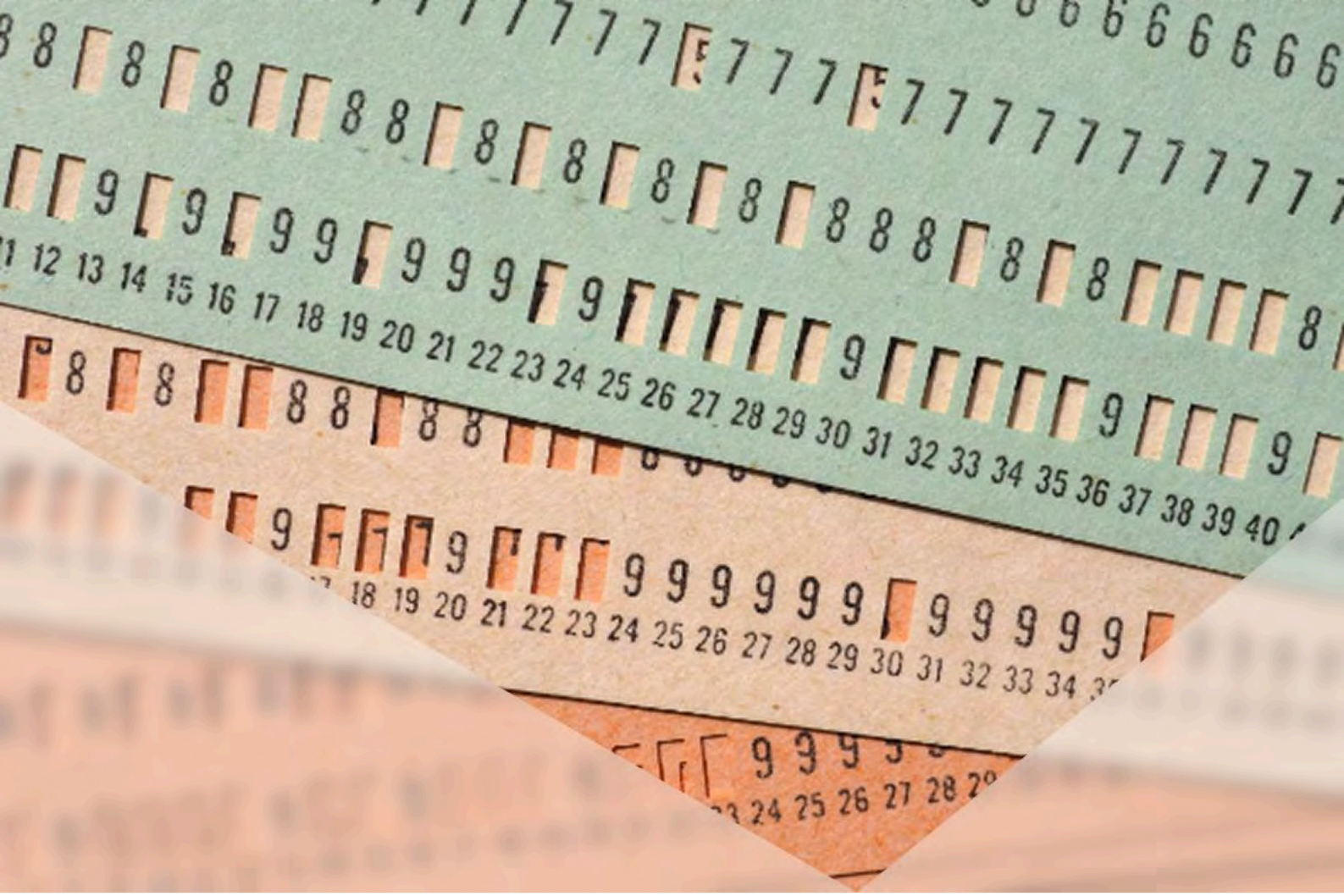
Over the past seven years, we've been fortunate to be involved with many ASP.NET modernization projects and have been able to learn a few things along the way. We've taken our knowledge and have compiled a short white paper on some technical areas for you to consider in your preparation for modernizing your software. Armed with the correct information and guidance, we hope that this whitepaper moves you to take the first steps in modernizing your software.

AUTHOR

Veli Pehlivanov, CTO at Resolute Software

Well-versed in web development, C#, .NET, JavaScript and Agile project management, Veli has played an active role in the creation of the Telerik ASP.NET AJAX suite. Later, he takes on the position of VP of Engineering at Progress, leading a team of over 100 talented software engineers. He has a lot to share about tackling IT challenges, building engineering teams and starting a software consultancy company spanning two continents.

Table of Contents



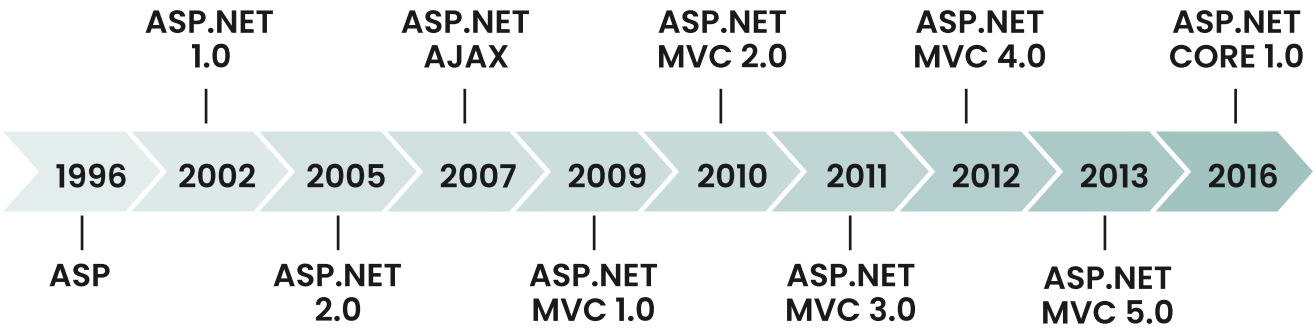
ASP.NET: A Look Back in Time

Microsoft released ASP.NET Web Forms way back in the Dark Ages, 2002. It was the dawn of web technologies, and Web Forms provided a cutting-edge framework for building web applications using patterns and paradigms from the desktop application development world, appealing to the larger community of desktop developers at the time. In 2007, with ASP.NET 2.0, Microsoft introduced the Asynchronous JavaScript and XML concept, more commonly known as AJAX. It enabled developers to create more interactive web applications that loaded content behind the scenes and reduced the typical “page flicker” effect of reloading browser pages.

A couple of years later, in 2009, ASP.NET took on a whole new web application development paradigm – the Model-View-Controller (MVC). Web technologies were moving ahead fast, and ASP.NET MVC was a cutting-edge programming model true to how the web works. Moreover, segregating the responsibilities of the different application layers provided the "right" way for building modern web applications, with framework constructs and abstractions that align more closely to the HTTP protocol and the way web servers work.

From 2009 to 2016, Microsoft had two equally important web application development frameworks that were both getting active support and were included in modern versions of .NET. Web Forms was front-and-center to all existing web applications written in .NET. At the same time, for green-field projects, developers could choose between Web Forms and MVC depending on project requirements and available developer skillsets. However, everything started to shift in late 2016, when Microsoft announced the future of .NET – a universal, cross-platform, open-source version of the framework – .NET – and there was room for only one flavor of ASP.NET in it. The modern ASP.NET web framework carries forward the strength of the MVC pattern but leaves Web Forms behind, making it a legacy technology.

ASP.NET Release Timeline



For you, it is important to know that Microsoft will not be “killing” ASP.NET Web Forms. Because Web Forms applications run on the classic .NET Framework, which is tied to the server’s operating system, you can continue running these apps for years to come. But, what this means is that your Web Forms applications will continue to work only as long as you have people to support them. Microsoft’s new-age development platforms (built on .NET 10, their latest release framework at the time of writing) are in active development and support, which means they have an ongoing release cadence and thus are introducing new features, improvements, and security updates.

Added to this, developers are typically interested in the latest, most exciting technologies. Microsoft is closely following the trends by launching modern web application paradigms such as MVC, Razor, Blazor, REST, and real-time APIs, which developers will likely favor more than legacy technologies like Web Forms.

In the modern app-driven business economy, the availability of engineering talent is a crucial prerequisite for successful software projects. A lack of interested developers makes Web Forms increasingly hard to justify as a framework of choice for large-scale, long-term software development with a critical impact on the business.

Risk-Based Approach to Software Modernization

The idea of rewriting an entire, often complex software system is understandingly overwhelming. The time and engineering investment in any system rewrite is a massive consideration for technical and business leaders looking to maximize the positive outcome for the organization. For any large-scale organizational effort, we recommend taking a Risk-Based approach to your modernization. This approach requires that stakeholders identify the key risks for a software system that, if realized, could lead to a negative business outcome.

Technology life cycle, project duration, impact on the business, technical skillset, and engineering capacity are some of the primary risk assessment areas we consider.

Risks identified in any of these areas could steer the software modernization strategy in different directions. For example, a lack of engineering talent in Web Forms could adversely affect the long-term maintainability of your Web Forms application. Ever-increasing user expectations for snappy graphical interfaces that load instantly and work anywhere could shift your decision towards modern web applications written in JavaScript and running on REST APIs.

- **Technology life cycle**
- **Project duration**
- **Impact on business**
- **Skillset**
- **Engineering capacity**

When creating mitigation strategies, understanding your potential software risks can make the difference between a successful software project that can boost the business and a software project that turns into a time and money pit with no clear sign of a positive outcome.

Technical Considerations for ASP.NET Web Forms Modernization

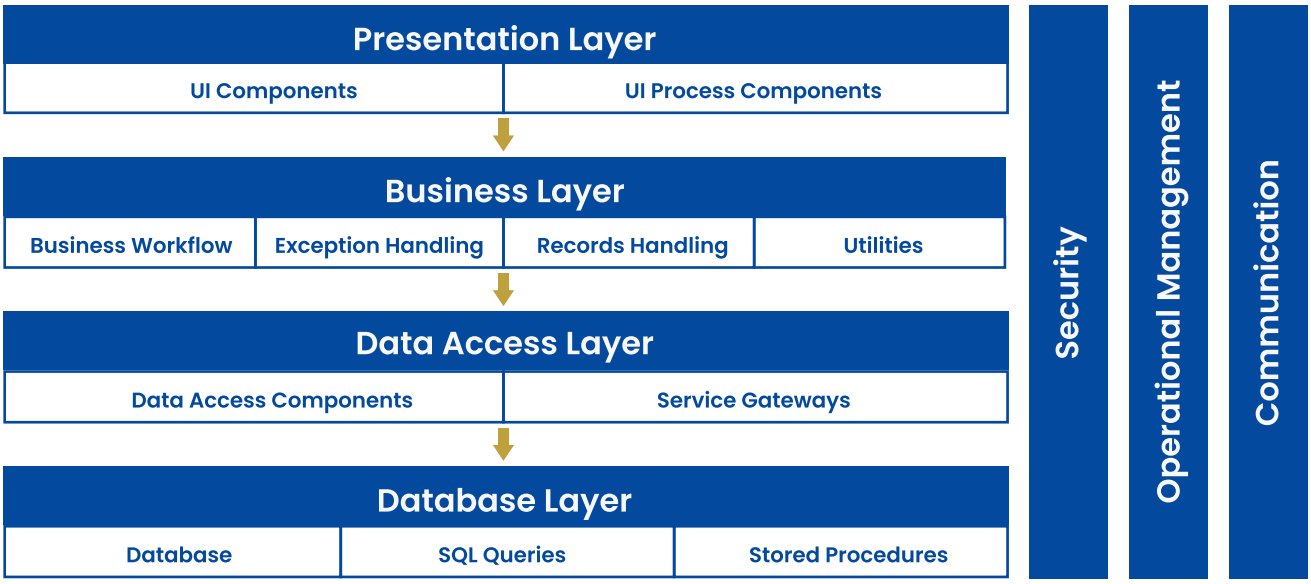
You should start any Web Forms application modernization with a set of key technical considerations. First, the basic assumption is that the outcome of this modernization effort is **a new and modern web application** that meets business objectives, provides a delightful user experience, and conforms to industry best practices for system design, efficiency, and maintainability.

To conform to these technical requirements, software architects map existing system capabilities onto a modern n-tier architecture that:

- Preserves and extends the capabilities of the original system
- Follows SOLID principles for code organization
- Segregates the application into multiple layers – Data Access, Business Logic, and Presentation
- Has well-defined and documented communication interfaces between layers
- Creates efficient data pipelines that minimize waste in storage, processing, and network bandwidth
- Provides snappy, intuitive, adaptive user interfaces that work on a multitude of browsers, devices, and screen sizes

Meeting the list of architectural requirements above can be a daunting task, especially when the modern system must preserve all the capabilities of the legacy system and minimize the negative impact of transitioning from the old to the new.

N-Tier Architecture



Our architects have taken these and many additional considerations and standardized them into a Modernization Assessment for our past projects – an intense, time-boxed requirements discovery and system architecture evaluation engagement. We take a deep dive into the legacy system, document the technical and business findings, and present a recommended system architecture, technology stack, and development approach. We thus help draft the big picture of a software modernization project and create a long-term roadmap to carry the development team successfully through the project.

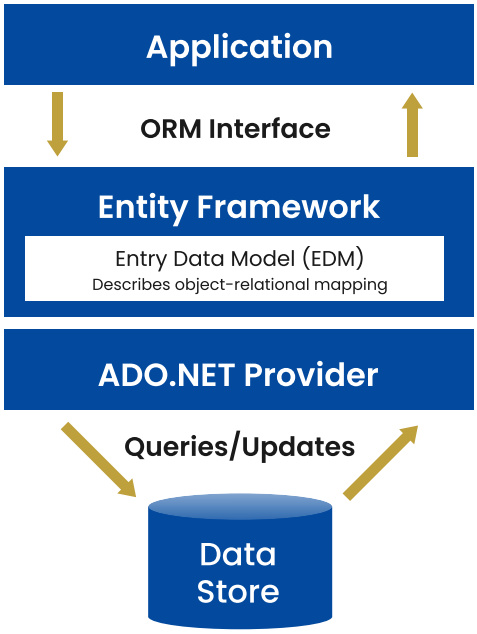
Considerations for the Data Access Layer

Data is the core asset in almost any business application. Our primary objective, as developers, is to create efficient ways to work with data – whether creating, storing, or consuming it. From a technical perspective, the data layer often constitutes the fixed, unchangeable, stable foundation on top of which legacy is maintained while the modern system evolves simultaneously.

The modern system's tech stack choices may create special considerations for the data layer, where the new system must run on the same database and schema version. For example, suppose your legacy Web Forms application uses Entity Framework or another enterprise quality ORM framework. In that case, it might be easier to preserve the data layer by replacing the ORM with a version compatible with the latest .NET framework like Entity Framework Core.

If you're going down that path, read our post on Eager Loading in Entity Framework Core for some tips and tricks around Entity Framework migration.

Creating Data Access Layer Using Entity Framework

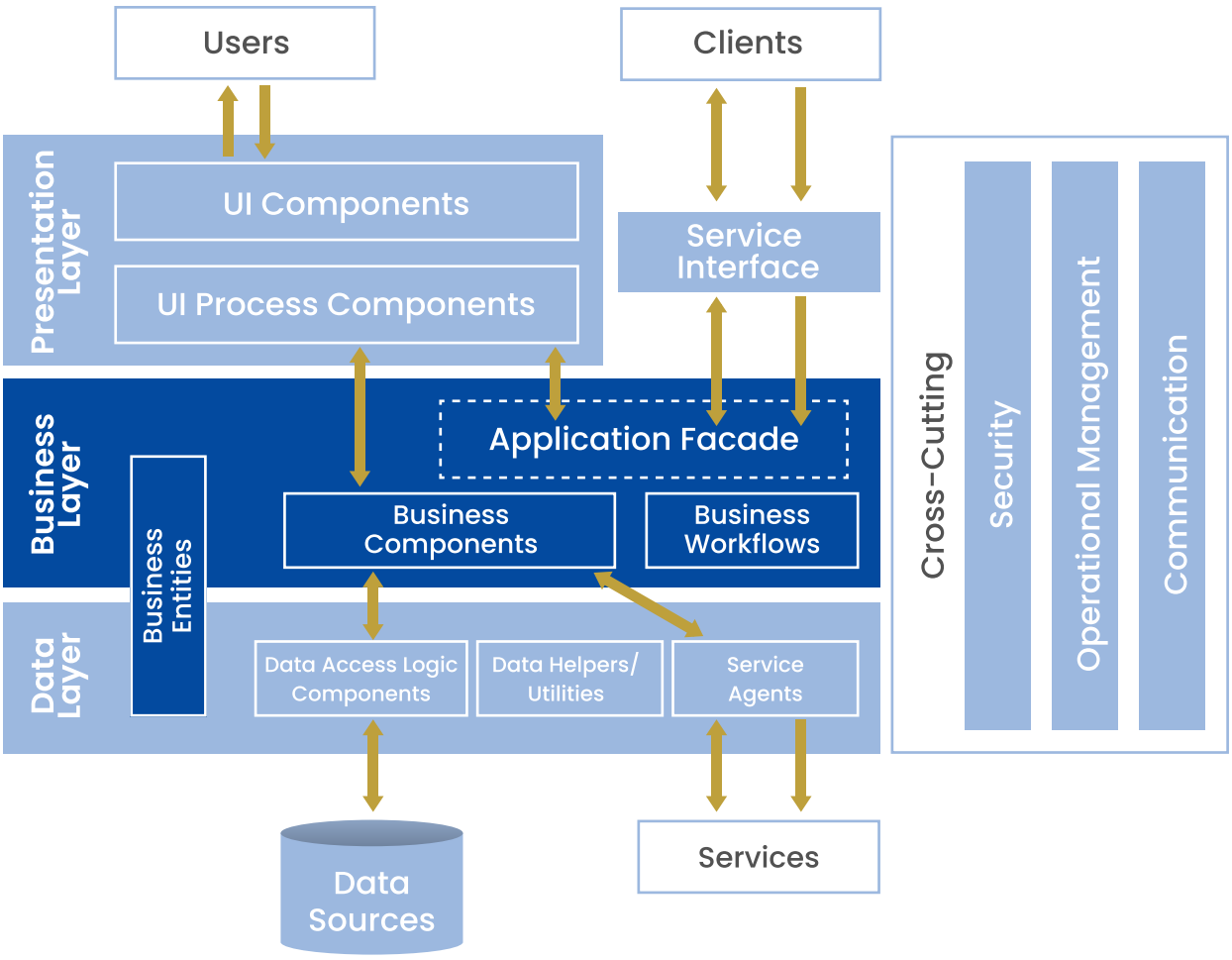


Considerations for the Business Logic Layer

From a modernization standpoint, the business logic layer is probably the most complicated area of your system. If your legacy Web Forms app follows the n-tier paradigm, you've done yourself a great favor, as your new ASP.NET application will likely get a lot of your original code organization.

On the other hand, if you're starting with n-tier architectures now, your business layer (also called Application Core) should generally be divided into multiple sub-areas:

- Entities – plain model classes and data containers representing your domain entities
- Business services – your domain-specific business logic classes
- Interfaces – the public API facade of your application core, facilitating decoupled system components through dependency injection
- Other system objects – domain events, exception classes, aggregates, value objects – all players in your domain model and required artifacts in an enterprise service architecture



Migrating your business logic from .NET Framework to modern .NET is a well-documented exercise, but one that also requires critical thinking. First, the business logic layer should be migrated to .NET Standard, which would produce interoperable compiled assemblies that can be safely referenced in multi-platform systems. When mapping between legacy and modern .NET versions, this table can help you choose the correct .NET Standard to target. Start migrating from the bottom of your dependency tree (usually a project named Entities, Model, or BLL) and work your way up through the dependency graph. Migrating projects between .NET Framework and .NET Standard is sometimes super easy and other times hard, depending on the framework APIs and NuGet packages you're using.

Most NuGet packages are either .NET Standard assemblies already or have .NET Standard counterparts. If you find yourself unable to use a .NET Standard version of a package, you can still build your .NET Standard project against a .NET framework dependency. You should expect compiler warnings, and not everything may work properly at runtime.

Typically, our advice in these situations is to either try to replace the package with a .NET Standard equivalent or similar or, if this is not possible, create unit tests for sufficient test coverage.

Testing potentially incompatible dependencies is usually problematic since developers have to cover all business logic cases calling an API in a dependent package to catch incompatible APIs.

Considerations for the User Interface Layer

The user interface of your application has a significant impact on the overall usability of the system. It is the biggest deciding factor between users sticking with your application versus leaving. In the extreme worst case, a poor user experience drives users to abandon your product or business altogether. Therefore, software modernization projects are often the time to improve the user interface, make it more intuitive, performant and thus increase user engagement and retention.

Our experience shows that most modernization projects turn into major UI overhaul initiatives, introducing modern user interaction paradigms like touch and voice, multi-device and on-the-go access through a responsive user interface, and better security and encryption. Picking the right choice of technology, tools, and patterns for your user interface is key to achieving success in a world of ever-increasing user expectations for fast, engaging, and intuitive applications. Developers call these **"consumer-grade applications"** – a term denoting the trend that massively popular consumer apps like Facebook, Twitter, and Google apps have created – shaping user expectations for all app experiences, including business apps. The modern business application faces the exact same user expectations for high-quality, interactive, easy-to-use visual interfaces.

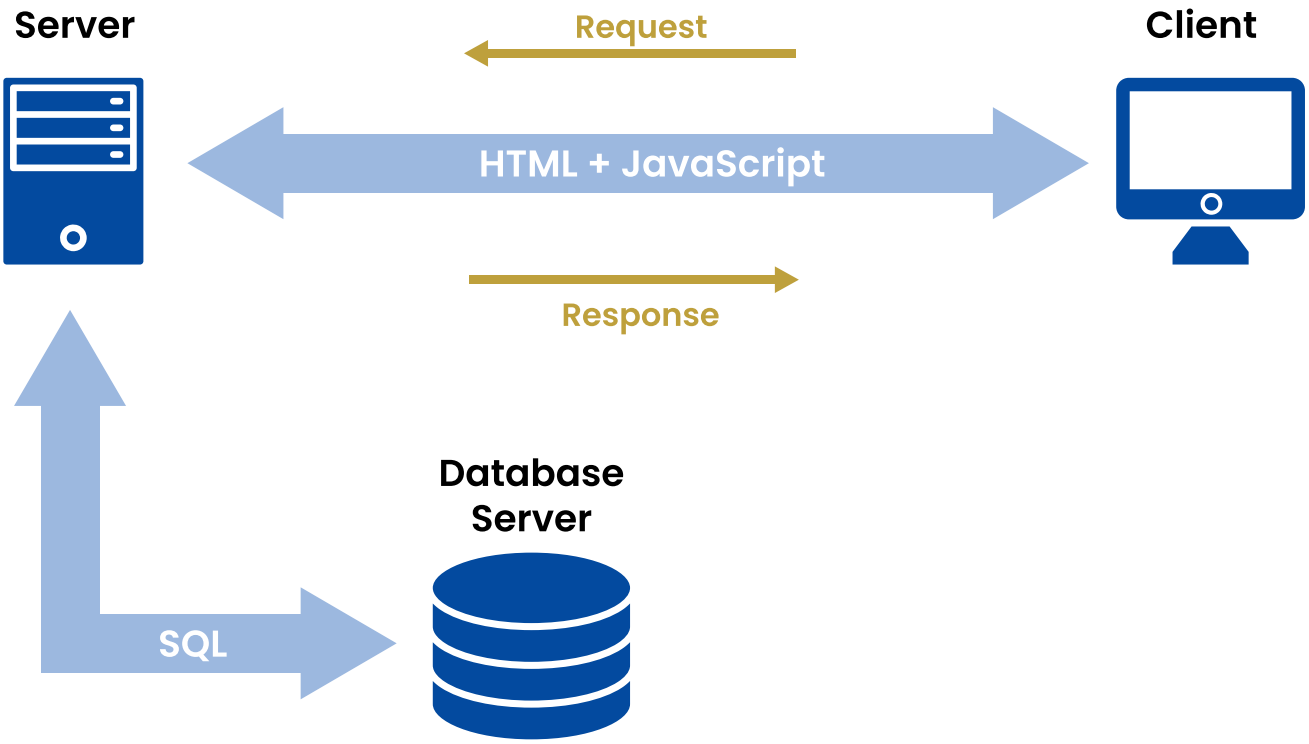
Why ASP.NET Web Forms is Not Suited for Consumer-Grade Applications

ASP.NET Web Forms is a “server pages” paradigm at its core – it presents fully rendered HTML pages from the web server. When the page loads, users can interact with it – click on buttons, expand sections, navigate further into detail pages. Such changes in the visual state require a roundtrip to the server, where the server generates an updated page.

This interaction pattern is, by design, bound to suffer from the network overhead of making requests between the browser and the server. Not only is the user waiting, but they will see the flickering between the old pages being dismissed and the new pages being rendered, creating a jarring user experience.

Advancements in browser technology and the introduction of AJAX back in 2007 helped improve the situation by reducing the page flicker, and the amount of data passing between the browser and the server but could not alleviate the server roundtrip overhead altogether. And while many capable web developers have used the power of JavaScript to create dynamic and interactive user interfaces on any web framework, the original Web Forms technology alone is highly insufficient to meet the requirements of the rich, modern browser applications.

Web Page Round-Trip

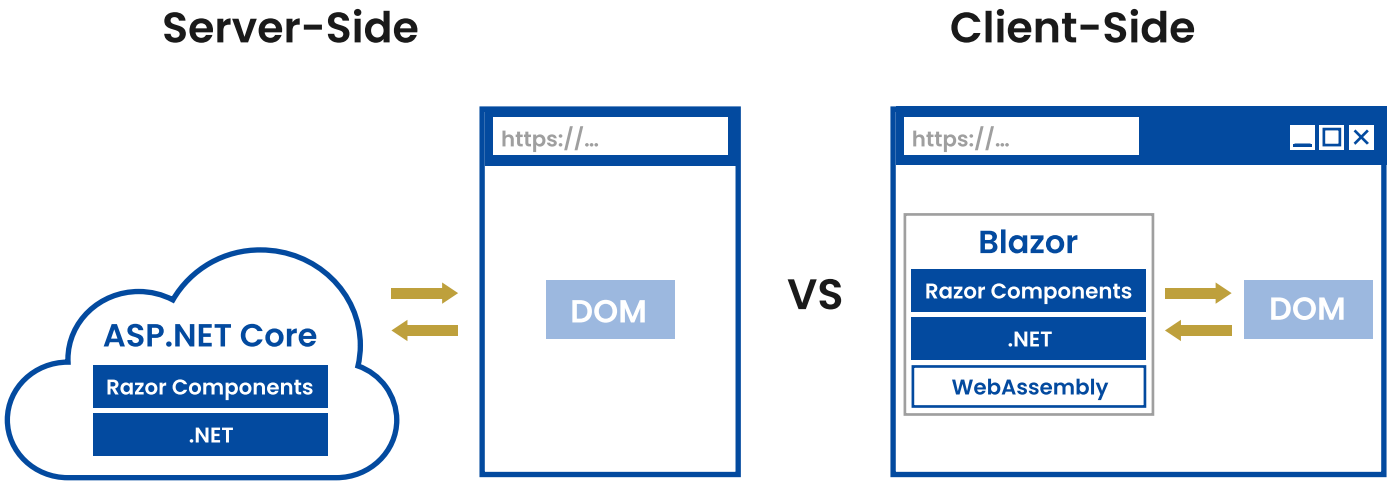


New Alternatives are Gaining Popularity

Web technologies have evolved significantly since the days of Web Forms. Today, snappy, responsive user interfaces are created using modern JavaScript frameworks. The Single-Page Application (SPA) paradigm shifted the responsibility for building and updating the web page to the browser with the help of JavaScript APIs and browser capabilities. Frameworks like Angular and React help create a clean and maintainable layered architecture for the application front-end and help scale the server by offloading more and more of the data processing to the browser.

A recent and well-hyped alternative by Microsoft – Blazor – promises to combine the best of modern JavaScript MVC patterns with the ability to write C# running in the browser for maximum .NET skill reuse. Through a capability in modern browsers to run compiled code, Blazor bootstraps a slimmed-down version of the .NET runtime in the browser process and allows .NET developers to build web interfaces and share code in .NET Standard assemblies – all in their favorite programming language.

Of course, not all business applications require robust and dynamic client-side user interfaces that change state rapidly in a highly interactive manner. For Server pages, their equivalent in the modern .NET framework is Razor pages, which provide a simple, elegant, and intuitive programming model for developers who need to write page-focused app scenarios.



Having options is a great thing, but making the right decision requires a careful assessment of the core system requirements and dealing with the impact of that choice. The decisions we make in this area affect the rest of the application layers. The frameworks mentioned above have their strengths and weaknesses, learning curves, ecosystem strength, and varying degrees of complexity (framework complexity and system complexity).

Considerations for Single-Page Application (SPA) Frameworks

Going with a modern SPA framework like Angular or React requires significant development effort in JavaScript. The client application codebase is typically separate from the rest of the .NET solution and is developed, tested, and deployed as a standalone system component. This separation increases the deployment complexity in a typical .NET and Visual Studio-based deployment scenario for ASP.NET applications.

Picking the Right Framework for Your Project

JavaScript is a popular language, and the JavaScript ecosystem is abundant with choice – tools, libraries, components… Anything a developer could want can probably be found in the JavaScript world. The most popular JavaScript web frameworks today – Angular, React, and Vue – all come with their own very active communities, rich tools, and third-party components. From a functional perspective, they typically vary on a scale between "just an MVC library" to an "all-in-one framework for web applications." The position of a particular framework on that scale describes the degree of "completeness" of the core library or framework.

Angular, for example, is an all-in-one framework that comes bundled with a component model, routing, forms, and localization in addition to the core MVC framework. On the other hand, React is considered more an MVC library that deals mainly with presenting the UI and tracking changes on the page, where additional application components must be brought in as external dependencies and third-party components from the ecosystem. The level of support components get from the engineering team behind the framework should also impact your choice.

Both Angular and React are considered mature and well-supported, but picking the proper framework for your project still requires additional research on framework capabilities, the richness of third-party components, and their long-term roadmap.

Angular		React
Google	Company	Facebook
2010(Angular JS), 2016(Angular 2+)	Released Year	2013
JavaScript, TypeScript	Code	JavaScript, HTLM, JSX
NativeScript (Web, iOS, Android)	Portability	ReactNative for mobile version (Android, iOS)
Real	DOM	Virtual
High	Performance	High
41.871	Github Stars	113.719
Open-source	Price	Open-source
Steep	Learning Curve	Moderate
Two-way	Data Binding	One-way
Client/Server Side	UI Rendering	Client/Server Side
Relatively Small	App size	Relatively Small

Possible Complexities with Blazor

Blazor is an exciting case since its fundamental promise is that developers can use C# to create web applications typically coded in JavaScript. Our take on Blazor is that it certainly has its place in the list of viable choices for modern web applications running on .NET, particularly where developers need to share and run the same code on both the client and the server. However, we don't consider the notion of "C# instead of JavaScript" true. Instead, we can generalize our thoughts as "C# and JavaScript together in the browser," as Blazor alone cannot replace all cases where JavaScript is required, particularly when applications need more advanced visual components like maps and charts, or modern browser capabilities such as local storage, IndexedDB, web workers and encryption.

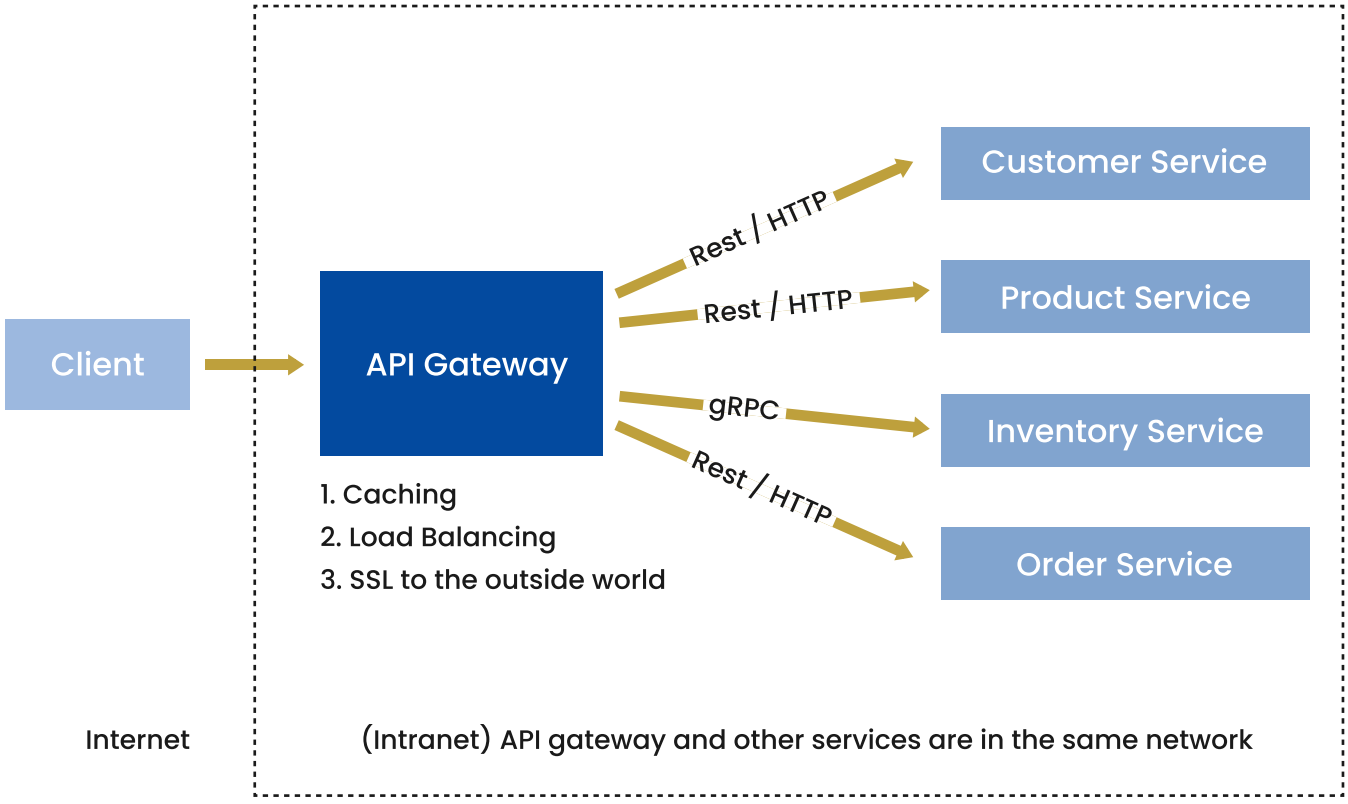
We continue to observe the need to write significant amounts of JavaScript in non-trivial Blazor applications, and Blazor apps still require the same level of JavaScript code organization and discipline as pure JavaScript SPA frameworks.

REST API Requirements

If leaning towards a SPA framework, one aspect to consider is that SPAs require a robust server API (typically a RESTful JSON API) that provides CRUD and other business layer capabilities. When a Web Forms application is the subject of modernization, a REST API may be a new system requirement since the typical Web Forms application has its client and server code tightly coupled through page code-behind files (.aspx.cs files) instead of a decoupled, UI-agnostic public API. When migrating to a modern .NET application, the need to access data and business logic on the server through a standalone API interface has a significant implication on your overall system design. It creates a shift from a page-focused to an endpoint-focused design. Since modern APIs are modeled around the notion of a resource (an entity or an aggregate) in your domain model that exposes data and operations through a uniform interface, this often constitutes a change of abstraction compared to Web Forms, where you expose functionality local to a specific server page, aggregating multiple entities and actions. This shift from a function-based to a resource-based system design affects the data pathways (how data is exchanged) and the "chattiness" (how frequently is data exchanged) between the client side and the server side of your application.

Considerations for Server Applications

The modern .NET application architecture promotes well-defined segregation between the client layer (a JavaScript SPA, a Blazor application, or a set of Razor server pages) and the server layer – a collection of data entities and business functions grouped logically in functional modules and exposed through an API interface. Note that in server pages with Razor, the Razor markup files and the underlying controller classes are still considered part of the client-side, even though they run on the server. The client-server segregation principle mandates that the client communicates with the server through a well-defined API surface that exposes data and business functions through a uniform interface. When the client is a standalone browser app (e.g., a JavaScript SPA or a Blazor app), this interface is typically a REST-ful JSON API provided over the HTTPS protocol. Server-side API controllers then tap into the pool of business functions and data access classes (the so-called Business Logic Layer – BLL – in an n-tier architecture) and provide useful business functionality to the client.



When your client is a set of Razor pages, the client-server API interface may not be as well-defined as a separate JSON API. Razor provides a mix of server functions (Razor controller methods executed on HTTP GET and POST requests) executed before the page is rendered to the browser and can call controller actions asynchronously through JavaScript. In the latter case, specific controller methods (actions) are exposed as a JSON API local to that particular Razor page. The two approaches are different in execution but similar from an architecture and component coupling perspective.

Regardless of whether you use Razor actions or API controller actions, your client application calls into server code and expects the operation result.

The above comparison between standalone SPA apps and Razor apps enables us to introduce an architecture principle that applies to both development approaches – the principle of "thin controllers." It states that you must keep your controllers thin and simple, limiting their responsibility to user authorization, data validation, sanitization, and formatting. Furthermore, any business logic, data queries, or modification must happen within the dedicated business layer (BLL). Thus, API or Razor controllers must inject and call into the required business classes, returning the computation result to the client through JSON results or rendered Razor views.

The principle of thin controllers constitutes a significant architectural benefit in your modern .NET application, enabling app developers to quickly refactor, enhance and extend the client application without introducing change to the domain-specific business layer. Further, decoupling your business functions from your presentation (client app) layer facilitates code reuse, preventing the repetition of identical data entry and manipulation of code we typically see in long-maintained legacy systems.

Testing, Deployment, and Operational Considerations

Most Web Forms applications are monoliths. They started as a single solution in Visual Studio, may have grown to multiple projects within that solution over time, but are usually built and deployed as a single chunk.

Modern web applications are modular both on the server (microservices) and the client (micro-frontends).

This creates a deployment mismatch between the legacy and the new, with implications to consider early on. No major system is replaced overnight.

The legacy system continues to live and receive updates and enhancements while parts are replaced with modern equivalents. Software teams must adopt Agile software deployment and operation practices while using legacy monolithic deployment procedures to minimize disruption in the software delivery to production. Replacing parts of an existing system always bears the risk of system regressions and disruptions for end users. Catching bugs in production is frustrating for both developers and users and can be expensive. To minimize risks, a precise and effective testing strategy must accompany any software modernization effort.

A healthy testing strategy includes a combination of unit tests for the domain model and business services, and a set of automated end-to-end tests that verify at least the critical use cases in the system, optimally all use cases.

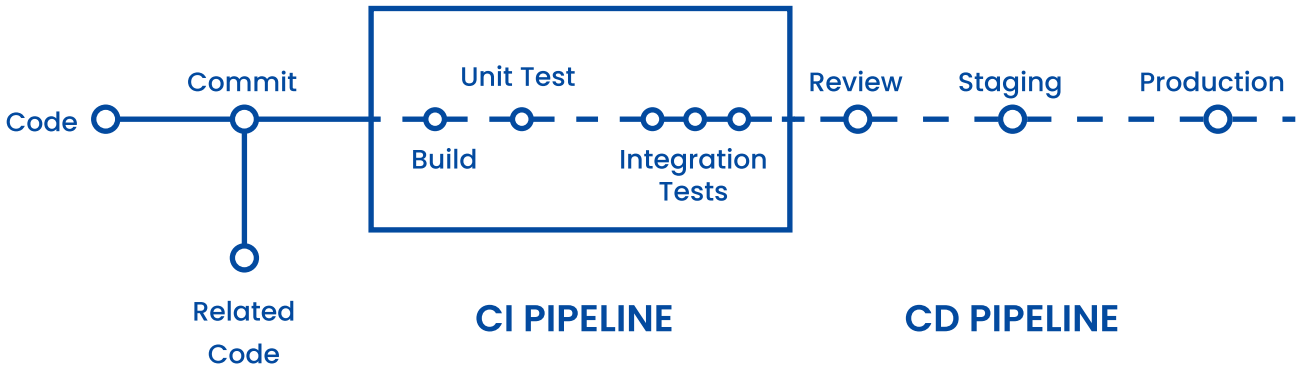
Of course, the distribution of your testing efforts across various test types depends on your business domain, system complexity, and user journeys. You may decide to create more unit tests for a complex domain model, such as financial analysis. Or you may choose to write integration tests that verify the correct interaction between different microservices in your server layer in a heavily distributed system. Architectures that promote heavy reliance on a REST API layer may require more API tests. Your pie chart of test types can be different, so long as you have more than one type of test in your solution.

Testing technologies and tools will also vary depending on your framework of choice. For .NET-based server applications, we love using NUnit and xUnit for unit testing, Postman and SoapUI for API testing. For JavaScript frameworks, some of the tools we use include Vitest, Jasmine and Jest for writing test specs, Cypress for end-to-end tests running in the browser, and Jenkins for tying everything together in nicely-orchestrated, build, test, and deploy workflows.

With the increasing adoption of highly independent, loosely coupled components in a microservice architecture, agile teams are increasingly adopting continuous delivery practices like automated testing, automated environment provisioning, source control-based build and deployment, quality gates, deployment promotion, and zero downtime deployment in production.

Combining these practices, collectively termed Continuous Integration and Delivery (CI/CD), reduces the cost of deployment and operation of a live system and increases the rate of delivering new value to end-users. The principles of CI/CD are increasingly considered an essential part of the modern software development life cycle. They lead to a DevOps culture that builds significant deployment and operational efficiency – a positive outcome not just for the software development team but for the entire business. We use CI/CD from day one of a software project. It's a great approach that allows us to deliver value quickly, receive feedback often, and fix production issues cheaply. We also help other engineering teams adopt CI/CD practices through training, environment.

Continuous Integration & Delivery



Final Words from Our CTO

Picking the right architecture for your modern .NET-based system is a challenging task. The sheer amount of technical, business, and user requirements to consider can be overwhelming. Add to that potential decision fatigue when picking from a multitude of JavaScript frameworks, application models, and architectural patterns, and you can quickly lose the light in the proverbial tunnel of your software modernization project.

At Resolute Software, we specialize in solving these challenges. We help businesses pick the right choice of technology, tools, and practices for successful modernization. We believe in "the right tool for the right job" philosophy, assess project requirements from different angles and help pick a feasible set of technologies for a successful software project.

> **"From our experience as software people, investing the time and effort in a carefully crafted system architecture is worth every minute."**

Just as Brian Foote and Joseph Yoder comment in their famous 1997 paper "Big Ball of Mud" – **"If you think good architecture is expensive, try bad architecture."**

Veli Pehlivanov
Co-Founder & CTO
Resolute Software

Need help to modernize your ASP.NET Web Forms applications?

Modern .NET is the new step forward with .NET, the platform which resonates with modern application development principles and where all future Microsoft investments in .NET will converge. While your ASP.NET Web Forms applications will remain functional if you have the talent to support them, there are many good reasons for you to consider migrating to the latest version. By undertaking this step, you will achieve simpler development models, better syntax in newer C# versions, fewer security risks and vulnerabilities and improved application stability overall. It takes a certain level of experience and resources to do such a migration smoothly without compromising your legacy application's uptime and performance. At Resolute Software, we assess your application's architecture, analyze interdependencies with other applications and databases, and design the separation of the business logic from the UI, before proposing a plan of action.

See how we at Resolute would go about modernizing your ASP.NET Web Forms application.

Resolute

Let's talk about your technology requirements.

Get in touch

USA

MA 01701, Framingham,
945 Concord St,

+1-617 386-9697

sales@resolutesoftware.com

Clutch ★★★★★

People First Company Award
2019 — 2023

Empowering Leadership Award
2024

Deloitte. Company to watch | Tech Fast 50

ISO 9001:2015
Quality Management
2025

ISO/IEC 27001:2022
Information Security Management
2025

ISO/IEC 20000-1:2018
IT - Service Management
2025

EcoVadis Commitment
2025