Desktop 2.0

Przewodnik po transformacji aplikacji .NET 4.8 do ekosystemu chmurowego

Marek Jasiński

2026-05-01

Architektura ASP.NET Core i Ekosystem Blazor

Wstęp merytoryczny: Ewolucja od .NET 4.8 do ASP.NET Core i Blazor w .NET 10

Przejście ze środowiska desktopowego, opartego o **.NET Framework 4.8** (WinForms, WPF), do nowoczesnego ekosystemu **ASP.NET Core** i frameworka **Blazor** reprezentuje fundamentalną zmianę paradygmatu architektonicznego. Ekosystem .NET ewoluował z platformy zamkniętej w systemie Windows (zależnej od ciężkiego API systemu operacyjnego) do wydajnego, wieloplatformowego rozwiązania open-source. ASP.NET Core wprowadza koncepcję zwinnego, modularnego potoku żądań HTTP obsługiwanego przez serwer **Kestrel**, podczas gdy Blazor pozwala na tworzenie interfejsów użytkownika bezpośrednio w języku C#, eliminując historyczną konieczność dzielenia logiki pomiędzy C# a JavaScript.

Transformacja z tradycyjnego oprogramowania typu “gruby klient” (Stateful) na środowisko webowe w .NET 10 wymaga zrozumienia mechanizmów asynchroniczności, nowoczesnego wstrzykiwania zależności (DI) oraz odmiennego cyklu życia aplikacji.

Analiza Porównawcza: Desktop vs Nowoczesny Web (.NET 10)

Wiedza zdobyta przy tworzeniu aplikacji WPF i WinForms stanowi solidny fundament, jednak wymaga mapowania na wzorce używane w aplikacjach ASP.NET Core i komponentach Blazor:

Koncepcja Desktop (WPF/ WinForms)	Odpowiednik w .NET 10 (ASP.NET Core / Blazor)	Charakter zmiany architektonicznej
Window / UserControl	**Komponent Razor (.razor)**	Przejście z XAML/projektanta WinForms na deklaratywny, oparty na C# i HTML komponent UI działający w przeglądarce lub na serwerze.

Koncepcja Desktop (WPF/ WinForms)	Odpowiednik w .NET 10 (ASP.NET Core / Blazor)	Charakter zmiany architektonicznej
App.config / Web.config	**appsettings.json**	Rezygnacja ze statycznego, ciężkiego pliku XML na rzecz hierarchicznego pliku JSON, z wbudowanym wsparciem wstrzykiwania zależności i konfiguracji środowiskowej.
Zdarzenia UI (Click, Loaded)	**EventCallback / OnInitializedAsync**	Zdarzenia systemowe i cykl życia stają się w pełni asynchroniczne i zintegrowane z cyklem renderowania frameworka Blazor.
Globalne instancje (Static/ Singleton)	**Dependency Injection (Wbudowany kontener DI)**	Stan w ASP.NET Core jest zarządzany poprzez usługi o ściśle określonym cyklu życia (Transient, Scoped, Singleton) rejestrowane w klasie bazowej.
Stan w pamięci RAM aplikacji	**Obwód (Circuit) / Stateless API**	Przejście z grubego klienta przechowującego dane lokalnie, na stan rozproszony lub utrzymywany na serwerze za pomocą połączenia SignalR w modelu Blazor Server.

Mechanika potoku ASP.NET Core i Modele Blazor

Hosting i Middleware

Tradycyjna architektura .NET 4.8 często opierała się na potężnej i zmonolityzowanej bibliotece `System.Windows`. W nowoczesnym ASP.NET Core sercem aplikacji jest **Generic Host** oraz modularny rurociąg żądań HTTP (Request Pipeline) ułożony z komponentów **Middleware**. Żądanie przechodzi przez kolejne filtry, w których każde oprogramowanie pośredniczące może zmodyfikować kontekst, zanim zostanie przekazane do odpowiedniego sterownika lub komponentu renderującego.

Modele uruchomieniowe Blazor

Blazor nie jest pojedynczą technologią, ale modelem komponentowym, który można osadzić w różnych konfiguracjach hostingowych: * **Blazor Server:** Komponenty wykonywane są po stronie serwera wewnątrz aplikacji ASP.NET Core. Komunikacja UI odbywa się poprzez protokół WebSockets z użyciem SignalR. Stan powiązany z połączonym klientem nazywany jest “obwodem” (circuit). Pozwala to na pełny dostęp do zasobów serwera bez wysyłania kodu C# do przeglądarki. * **Blazor**

WebAssembly (Wasm): Aplikacja (wraz z runtime'em .NET) pobierana jest do przeglądarki klienta i wykonywana w piaskownicy przeglądarki (sandbox). Model ten umożliwia pełną pracę w trybie offline, w architekturze PWA (Progressive Web App) i zdejmuje obciążenie obliczeniowe z serwera. * **Blazor Hybrid:** Łączy nowoczesne technologie webowe z oprogramowaniem natywnym. ## Dobre Praktyki i Antywzorce

W procesie przenoszenia logiki z .NET Framework 4.8 do architektury .NET 10 łatwo popełnić błędy polegające na przenoszeniu dawnych nawyków w nowe ramy technologiczne.

- **Antywzorzec (ZASADA ZERO HYBRYDY W 4.8):** Choć w oficjalnej dokumentacji pojawiają się informacje o integracji kontrolek `BlazorWebView` w celu modernizacji istniejących aplikacji WPF czy Windows Forms, należy to kategorycznie sprostować w odniesieniu do środowisk współdzielonych z .NET 4.8. Użycie natywnego Blazora w "starej" aplikacji Windows Forms w architekturze Framework 4.8 jest niemożliwe; mechanika hybrydowa z `BlazorWebView` wymaga zastosowania nowoczesnego środowiska uruchomieniowego w projektach natywnych .NET 6+. Migracja musi polegać na wydzieleniu logiki biznesowej do .NET Standard 2.0 i sukcesywnym podpinaniu nowoczesnego front-endu w architekturze .NET 10.
- **Antywzorzec:** Blokowanie wątków systemowych (np. `Thread.Sleep` czy synchroniczne `Wait()` na zadaniach). Aplikacje ASP.NET Core i serwer Kestrel zostały zoptymalizowane pod kątem tysięcy współbieżnych połączeń dzięki zastosowaniu mechanizmów asynchronicznych. Złamanie tej zasady w architekturze webowej zdestabilizuje serwer. Programowanie musi być w całości oparte o wzorce `async/await`.
- **Dobra praktyka:** Pełna separacja logiki prezentacji (Blazor/API) od logiki biznesowej i dostępu do danych (Entity Framework Core) za sprawą Dependency Injection. W ASP.NET Core wstrzykiwanie zależności nie jest zewnętrznym dodatkiem – to wbudowany filar frameworka udostępniany poprzez interfejs `WebApplicationBuilder.Services`. Umożliwia to zjawisko całkowicie luźnego powiązania (loose coupling) i oddzielenie kodu renderującego UI od operacji I/O.

Laboratorium kodu: Rejestracja usług i budowa komponentu

Poniżej przedstawiono implementację ilustrującą współdziałanie nowej mechaniki konfiguracyjnej oraz architektury asynchronicznej. Rejestracja zależności odbywa się w pliku punktu startowego `Program.cs`, co zastępuje w całości dawne konstrukcje z wykorzystaniem `Global.asax` i ciężkich konstruktorów.

```
// Program.cs w .NET 10 (Uproszczony model Generic Host)
using Microsoft.EntityFrameworkCore;
using ModernApp.Data;

var builder = WebApplication.CreateBuilder(args);

// Rejestracja puli logiki poprzez wbudowany kontener DI
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();

// Użycie appsettings.json do odczytu Connection Stringa wstrzykiwanego do EF Core
builder.Services.AddDbContextFactory<AppDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

var app = builder.Build();

// Konfiguracja potoku żądań Middleware
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseAntiforgery();

app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();

app.Run();
```

Mechanika po stronie klienta również ulega uproszczeniu, przechodząc z obiektów `Dispatcher` na zautomatyzowany cykl odświeżania cyklu życia. Poniżej przedstawiony jest w pełni asynchroniczny komponent odczytujący dane z systemu, respektujący zasady modelu bezstanowego po stronie UI:

```
// TasksList.razor
@page "/tasks"
@inject ITaskService TaskService

<div class="task-board">
    @if (_tasks == null)
    {
        <p><em>Ładowanie strumieniowe, proszę czekać...</em></p>
    }
    else
    {
        <table class="table">
            @foreach (var task in _tasks)
            {
                <tr>
                    <td>@task.Title</td>
                    <td>@task.AssignedTo</td>
                </tr>
            }
        </table>
    }
```

```
</div>

@code {
    private List<TaskDto>? _tasks;

    // Asynchroniczny odpowiednik zdarzenia Form_Load / UserControl.Loaded
    protected override async Task OnInitializedAsync()
    {
        // Wywołanie usługi wstrzykniętej z DI - brak blokowania wątku interfejsu
        (UI Thread)
        _tasks = await TaskService.GetActiveTasksAsync();
    }
}
```

Wnioski architektoniczne

Porzucenie zmonolityzowanego modelu *Stateful* (aplikacji stanowych Desktop w .NET 4.8) na rzecz ekosystemu **.NET 10**, gdzie dominuje bezstanowa architektura potoku żądań (Middleware) i elastyczne hostowanie interfejsu (Blazor Server/ WebAssembly), wymaga transformacji sposobu projektowania pamięci i stanu aplikacji. Blazor upodabnia pisanie nowoczesnego, responsywnego interfejsu klienta do modelu podobnego pod względem mentalnym do wzorca znanej abstrakcji klas c#, jednak eliminuje bezpośredni, procesowy dostęp do hardware'u stacji roboczej z poziomu interfejsu, wymuszając w pełni asynchroniczną i bezpieczną integrację poprzez komunikację sieciową (API i potoki wstrzykiwania zależności).

Fundamenty Architektury

Wstęp merytoryczny: Fundamenty nowoczesnej architektury webowej

W niniejszym rozdziale przeanalizujemy fundamenty architektury **ASP.NET Core**, które stanowią niezbędną infrastrukturę dla frameworka **Blazor** oraz nowoczesnych interfejsów API. Przejście z aplikacji desktopowych opartych o **.NET Framework 4.8** na nowoczesny ekosystem sieciowy wymusza odrzucenie paradygmatu aplikacji stanowych (Stateful), w których system operacyjny zarządzał cyklem życia okna, na rzecz architektury bezstanowej (Stateless) opartej o rurociąg przetwarzania żądań HTTP.

Środowisko uruchomieniowe w .NET 10 jest budowane w oparciu o modułowy mechanizm **Generic Host**, który integruje wbudowane wstrzykiwanie zależności (DI), konfigurację środowiskową oraz potok oprogramowania pośredniczącego (Middleware). Zrozumienie tych filarów jest krytyczne dla inżynierów migrujących systemy desktopowe, ponieważ w nowym modelu to infrastruktura serwerowa odpowiada za orkiestrację żądań, a UI jest jedynie odseparowaną warstwą prezentacji.

Analiza Porównawcza: Desktop (4.8) vs ASP.NET Core (.NET 10)

Koncepcja Desktop (.NET 4.8)	Odpowiednik w ASP.NET Core (Blazor/.NET 10)	Charakter zmiany architektonicznej
App.xaml.cs / Program.cs (Statyczne metody startowe)	**Minimal API / WebApplicationBuilder (`Program.cs`)**	Inicjalizacja aplikacji została ujednolicona w jednym pliku używającym wzorca *Builder*, inicjalizującym Host, DI i rurociąg żądań.
Globalne instancje (Wzorce Singleton / Zmienne statyczne)	**Wbudowane Wstrzykiwanie Zależności (DI)**	Fundamentem współdzielenia stanu jest kontener DI wstrzykujący obiekty o zdefiniowanym cyklu życia (Transient, Scoped, Singleton) prosto do konstruktorów lub komponentów.

Koncepcja Desktop (.NET 4.8)	Odpowiednik w ASP.NET Core (Blazor/.NET 10)	Charakter zmiany architektonicznej
Pliki App.config / Web.config (XML)	**appsettings.json oraz Zmienne Środowiskowe**	Zastąpienie ciężkich plików XML elastyczną, hierarchiczną konfiguracją JSON, która automatycznie reaguje na zmianę środowiska wykonawczego.
Ścisłe powiązanie z systemem Windows / IIS	**Serwer Kestrel**	Aplikacja webowa uruchamia własny, wydajny i wieloplatformowy serwer Kestrel, uwalniając proces od konieczności hostowania przez systemowy IIS.
Zdarzenia UI (Click, Load) w kodzie głównym	**Modułowe Middleware**	Zamiast monolitycznej obsługi cyklu życia używa się filtrów i *Middleware*, w którym każde żądanie przechodzi przez kaskadowy potok (np. autoryzacja, routing, logowanie).

Głębokie Nurkowanie (Deep Dive): Kluczowe filary architektury

Middleware (Rurociąg żądań HTTP)

Zamiast korzystać ze zmonolityzowanej biblioteki `System.Web`, na której w dużej mierze opierał się dawny ASP.NET, ASP.NET Core przetwarza żądania za pomocą sekwencji **Middleware**. Każdy komponent rurociągu analizuje przychodzący obiekt `HttpContext` i może podjąć decyzję o modyfikacji żądania, obsłużeniu go i zakończeniu procesu (tzw. zwarcie rurociągu - np. odesłanie pliku statycznego), lub przekazaniu wykonania do kolejnego modułu za pomocą asynchronicznego delegata. Właściwa kolejność rejestracji oprogramowania pośredniczącego determinuje całe zachowanie aplikacji; na przykład middleware obsługujący wyjątki musi zostać dodany jako pierwszy, aby mógł przechwycić błędy z każdego kolejnego poziomu wykonania.

Wstrzykiwanie Zależności (Dependency Injection)

W .NET 4.8 wstrzykiwanie zależności było opcjonalne i wymagało użycia zewnętrznych bibliotek. W środowisku ASP.NET Core DI jest pierwszoplanowym, wbudowanym rozwiązaniem strukturalnym, udostępniającym instancje na terenie całej aplikacji. Komponenty i kontrolery otrzymują wymagane serwisy (np. dostęp do bazy danych, mechanizmy uwierzytelniania) wyłącznie poprzez swoje konstruktory. Ten wzorzec radykalnie zwiększa testowalność oraz egzekwuje luźne powiązanie (loose coupling) klas.

Serwer Kestrel i Hosting

Silnik ASP.NET Core tworzy instancję *Host*, hermetyzując implementację serwera HTTP, rurociąg middleware oraz mechanizmy takie jak logowanie i wstrzykiwanie konfiguracji. Odrzucono uzależnienie od ciężkiego oprogramowania OS, a głównym mechanizmem odbierania ruchu sieciowego stał się **Kestrel** – lekki, wieloplatformowy serwer sieciowy, optymalizowany pod kątem ogromnej przepustowości.

Dobre Praktyki i Antywzorce

Proces adaptacji inżynierów desktopowych do technologii webowych często wiąże się z nawykami, które w ASP.NET Core są krytycznymi błędami.

- **Antywzorzec: Przechowywanie tajemnic w kodzie:** Praktyka kodowania parametrów połączeń (connection strings) czy kluczy prywatnych w plikach z kodem źródłowym (bądź w lokalnym konfiguratorze dystrybuowanym z repozytorium) jest poważną luką bezpieczeństwa. W ASP.NET Core niejawne dane developerskie powiny być przetrzymywane przy użyciu narzędzia Secret Manager w środowisku programistycznym, a na produkcji w usługach typu Azure Key Vault.
- **Dobra Praktyka: Hierarchiczna Konfiguracja `appsettings.json`:** Prawidłowo zaprojektowany system używa silnie typowanych klas dla ustawień konfiguracyjnych poprzez “Options Pattern”, co integruje strukturę pliku konfiguracyjnego `appsettings.json` wprost do kontenera wstrzykiwania zależności.

Laboratorium kodu: Architektura Punktu Wejścia (Program.cs)

Poniższy kod przedstawia ewolucję logiki inicjalizacji z monolitycznego pliku znanego z dawnego desktopu (czy WebForms), do minimalistycznego i deklaratywnego API w nowoczesnym środowisku .NET. Prezentuje powołanie

instancji `WebApplicationBuilder`, rejestrację serwisów do kontenera DI, a ostatecznie skonfigurowanie asynchronicznego rurociągu Middleware.

```
using Microsoft.EntityFrameworkCore;
using ModernApp.Data;

// Inicjalizacja budowniczego Host'a ze wbudowaną obsługą appsettings.json
var builder = WebApplication.CreateBuilder(args);

// --- REJESTRACJA SERWISÓW (DEPENDENCY INJECTION) ---
// Rejestracja bazy danych ze ścisłym wstrzykiwaniem parametru z konfiguracji
builder.Services.AddDbContextFactory<AppDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")));

// Rejestracja obsługi środowiska Blazor
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();

// Budowa instancji aplikacji
var app = builder.Build();

// --- KONFIGURACJA RUROCIĄGU (MIDDLEWARE PIPELINE) ---
// Kolejność wywołań determinuje cykl przetwarzania HTTP

// 1. Obsługa wyjątków - na samym początku rurociągu
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error");
    app.UseHsts();
}

// 2. Przekierowanie ruchu na protokół szyfrowany HTTPS
app.UseHttpsRedirection();

// 3. Obsługa plików statycznych (np. CSS/JS) z katalogu wwwroot
// Jeśli żądanie dotyczy pliku, rurociąg ulega zwarciu (short-circuit)
app.UseStaticFiles();

// 4. Mechanizmy zabezpieczeń przed atakami
app.UseAntiforgery();

// 5. Mapowanie tras i renderowanie interfejsu (Ostatni punkt węzłowy Middleware)
app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();

// Uruchomienie pętli nasłuchującej (Kestrel Server)
app.Run();
```

Wnioski architektoniczne

Architektura ASP.NET Core oferuje modularność na poziomie wcześniej niedostępnym dla aplikacji desktopowych korzystających ze sztywnego ekosystemu Windows. Dzięki przejściu na jednolity standard `Generic Host`, elastycznemu strumieniowi `Middleware` i potężnemu wstrzykiwaniu zależności, programiści tworzący nowoczesne rozwiązania za pomocą **Blazor** otrzymują aplikacje maksymalnie skalowalne.

Zrozumienie fundamentów takich jak bezstanowość żądań HTTP, cykl życia wstrzykiwanych instancji oraz separacja warstwy konfiguracji pozwala inżynierom .NET 4.8 uniknąć kopiowania złych wzorców technicznych do nowego ekosystemu, otwierając drogę nie tylko dla zaawansowanego front-endu serwowanego w przeglądarce, ale i wieloplatformowych mikrouUsług rozwijanych ramię w ramię z logiką UI.

Koncepcja Generic Host

Wstęp merytoryczny: Generic Host jako serce aplikacji .NET 10

W architekturze nowoczesnych systemów webowych opartych na .NET 10, sercem każdej aplikacji jest mechanizm **Generic Host** (Host Ogólny). W przeciwieństwie do aplikacji desktopowych w środowisku **.NET Framework 4.8**, gdzie cyklem życia zarządzał system operacyjny Windows (np. poprzez proces pętli komunikatów w funkcji `Application.Run`), w środowisku ASP.NET Core kontrolę nad procesem przejmuje wyspecjalizowany kontener. Host ten na etapie uruchamiania hermetyzuje wszystkie zasoby aplikacji, w tym: implementację serwera HTTP (Kestrel), potok oprogramowania pośredniczącego (Middleware), system wstrzykiwania zależności (DI), konfigurację oraz logowanie. Zrozumienie roli Hosta jest kluczowe dla programistów migrujących systemy z modelu *Stateful*, ponieważ w webowym rurociągu żądań to Host dyktuje architekturę i granice życia obiektów.

Analiza Porównawcza: Inicjalizacja Aplikacji

Koncepcja Desktop (.NET 4.8)	Odpowiednik w ASP.NET Core (.NET 10)	Charakter zmiany architektonicznej
Klasa `Application` (App.xaml.cs)	`WebApplicationBuilder` / `Generic Host`	Zastąpienie mechanizmów systemu operacyjnego abstrakcją odpowiedzialną za orkiestrację serwera i wbudowanych systemów cross-cutting (logowanie, DI).
Pętla `Application.Run()`	`app.Run()` **(Kestrel Server)**	Aplikacja webowa nie czeka na zdarzenia klawiatury/myszy, lecz włącza wielowątkowy nasłuch żądań sieciowych na serwerze wieloplatformowym.

Koncepcja Desktop (.NET 4.8)	Odpowiednik w ASP.NET Core (.NET 10)	Charakter zmiany architektonicznej
Klasa statyczna `ConfigurationManager`	**Wbudowana integracja** `appsettings.json`	W .NET 10 konfiguracja jest automatycznie ładowana i wstrzykiwana przez Hosta z wielu źródeł (pliki JSON, zmienne środowiskowe) w trakcie jego budowy.
Brak wbudowanego ujednoliconego cyklu życia usług	**Interfejsy Hosta (Worker Services)**	Generic Host umożliwia współdzielenie logiki DI, konfiguracji i zarządzania cyklem życia nie tylko w aplikacjach webowych, ale również w usługach tła (tzw. scenariusze non-web).

Głębokie Nurkowanie (Deep Dive): Anatomia Generic Hosta

W początkowych wersjach frameworka ASP.NET Core architektura opierała się ściśle na obiekcie `WebHost`. Obecnie środowisko ewoluowało w stronę uniwersalnego **Generic Hosta**, który jest zalecanym podejściem dla każdego typu aplikacji serwerowej. Nowoczesne szablony projektów posługują się tzw. Minimal Hostem opartym na klasach `WebApplication` oraz `WebApplicationBuilder`.

Obiekt `WebApplicationBuilder` podczas inicjalizacji wykonuje ogromną pracę infrastrukturalną bez widocznego nakładu kodu: 1. Konfiguruje wieloplatformowy serwer sieciowy **Kestrel** jako główny silnik do przetwarzania żądań (często integrowany z IIS w scenariuszach reverse proxy). 2. Ładuje i scala konfigurację ze źródeł lokalnych (`appsettings.json`), środowiskowych i z wiersza poleceń. 3. Uruchamia zintegrowany kontener Dependency Injection i rejestruje w nim kluczowe podsystemy frameworka. 4. Ustawia standardowe dostarczycieli logów (np. konsola, debug), dzięki czemu diagnostyka działa natychmiast po uruchomieniu aplikacji.

Architektura systemowa: Cykl życia Hosta

W projektach bazujących na technologiach desktopowych lub w środowisku `System.Web` cykl życia był w dużej mierze monolityczny, a logika startowa często rozproszona w strukturach takich jak `Global.asax`. Generic Host implementuje jasny podział na dwie fazy: 1. **Fazę budowy (Builder):** Rejestrowanie usług (DI) i ładowanie konfiguracji. 2. **Fazę wykonawczą (App):** Konstruowanie sekwencyjnego rurociągu potoku Middleware, który decyduje, jak traktować przychodzące wywołania, kończącego się ostatecznie uruchomieniem serwera.

To podejście sprawia, że przeniesienie starych systemów korzystających z klas abstrakcji do środowiska zorientowanego na wstrzykiwanie zależności staje się naturalne i przewidywalne.

Dobre Praktyki i Antywzorce

Proces migracji do infrastruktury .NET 10 niesie ze sobą ryzyko przenoszenia starych nawyków. W kontekście uruchamiania i hostowania aplikacji inżynierowie powinni zwrócić uwagę na następujące aspekty:

- **Antywzorzec:** Ręczne tworzenie i instancjonowanie "zależności" w konstruktorach obiektów klasycznych (tzw. "new is glue"). W architekturze opartej na Generic Host, obiekty takie jak klasy logiki czy kontekst bazy danych (np. Entity Framework) są konfigurowane raz w klasie `WebApplicationBuilder` i obsługiwane wyłącznie za pomocą natywnego kontenera Dependency Injection.
- **Dobra Praktyka:** Używanie `WebApplication` do zarządzania infrastrukturą zamiast utrzymywania globalnego stanu. Starsze środowiska zorientowane na kompatybilność, takie jak klasyczny `WebHost`, nie są już zalecane w nowo powstających aplikacjach .NET 10.

Laboratorium kodu: Inicjalizacja aplikacji webowej

Poniższy fragment kodu z wykorzystaniem koncepcji Top-Level Statements ukazuje, jak Generic Host łączy się w całość w nowoczesnej aplikacji z Blazorem. Eliminuje to zbędne otoczki formalne starych aplikacji (jak statyczne klasy `Program` i procedury `Main`).

```
// Program.cs
using ModernApp.Data;
using Microsoft.EntityFrameworkCore;

// 1. Utworzenie Minimal Hosta (Generic Host) - hermetyzuje serwer Kestrel
var builder = WebApplication.CreateBuilder(args);
```

```
// 2. Faza BUDOWY (Rejestracja usług i konfiguracji w kontenerze DI zarządzanym
        przez Hosta)
// Pobranie Connection String wstrzykniętego do appsettings.json poprzez wbudowany
        mechanizm Hosta
builder.Services.AddDbContextFactory<AppDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")
    ?? throw new InvalidOperationException("Brak konfiguracji bazy danych.")));

builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();

// 3. Budowa instancji aplikacji i zatrzaśnięcie kontenera DI
var app = builder.Build();

// 4. Faza WYKONAWCZA: Konfiguracja potoku żądań Middleware w obrębie Hosta
if (!app.Environment.IsDevelopment()) // Środowisko kontrolowane przez Hosta
{
    app.UseExceptionHandler("/Error", createScopeForErrors: true);
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseAntiforgery();

app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();

// 5. Rozpoczęcie nasłuchu przez serwer Kestrel – asynchroniczne utrzymanie procesu
        aplikacji
app.Run();
```

Wnioski architektoniczne

Mechanizm **Generic Host** stanowi nieodłączny fundament stabilności ekosystemu **.NET 10**. Odciąża on programistę z manualnego zestawiania potoków autoryzacyjnych, ładowania konfiguracji XML (jak dawne `app.config`) oraz wiązania procesów aplikacyjnych z niskopoziomowym kodem systemu operacyjnego. Skalowalność serwera Kestrel, w połączeniu z modułowym wstrzykiwaniem zależności zintegrowanym na etapie budowania Hosta, tworzy elastyczny, bezstanowy i wysoko wydajny szkielet. Właśnie na nim inżynierowie mogą opierać nowoczesne implementacje logiki biznesowej przeniesionej z wycofywanych aplikacji WinForms i WPF.

Zarządzanie cyklem życia w architekturze Generic Host

Wstęp merytoryczny: Zarządzanie cyklem życia w ekosystemie Generic Host

W procesie transformacji aplikacji z klasycznego środowiska **.NET Framework 4.8** (WPF, WinForms) do nowoczesnego **.NET 10**, kluczowym wyzwaniem architektonicznym jest zrozumienie zmiany w zarządzaniu cyklem życia aplikacji. W środowisku desktopowym cykl życia był ściśle powiązany z pętlą komunikatów systemu operacyjnego Windows (np. za pośrednictwem `Application.Run` lub zdarzeń okna głównego). W ASP.NET Core odpowiedzialność tę przejmuje scentralizowany mechanizm **Generic Host** (Host Ogólny).

Generic Host hermetyzuje wszystkie zasoby aplikacji, integrując wbudowane wstrzykiwanie zależności (DI), konfigurację środowiskową, logowanie oraz właśnie zarządzanie cyklem życia (App Lifetime Management). Co więcej, abstrakcja ta pozwala na przeniesienie nowoczesnych wzorców nie tylko do aplikacji webowych, ale również do procesów działających w tle (tzw. scenariusze non-web).

Analiza Porównawcza: Desktop vs Nowoczesny Web (.NET 10)

Przejście z modelu “grubego klienta” (Stateful) na infrastrukturę zarządzaną przez Hosta (Stateless) drastycznie zmienia podejście do czasu życia obiektów:

Koncepcja Desktop (WPF/ WinForms)	Odpowiednik w .NET 10 (Generic Host)	Charakter zmiany architektonicznej
Pętla zdarzeń systemu Windows	**Zarządzanie cyklem przez `IHost`**	Zależność od OS zostaje zastąpiona przez niezależny, wieloplatformowy proces hostujący (często używający serwera Kestrel).
Globalne instancje w pamięci RAM	**Cykle życia w kontenerze DI**	W .NET 10 zarządzanie życiem instancji jest ściśle kontrolowane przez DI (cykle: *Transient*, *Scoped*, *Singleton*) pod nadzorem Hosta.

Koncepcja Desktop (WPF/ WinForms)	Odpowiednik w .NET 10 (Generic Host)	Charakter zmiany architektonicznej
`BackgroundWorker` / Ręczne wątki	`IHostedService` (Usługi tła)	Generic Host dostarcza natywny interfejs do obsługi długotrwałych procesów w tle, integrując je z bezpiecznym uruchamianiem i wyłączaniem.
Zdarzenia `App.Startup` / `App.Exit`	`IHostApplicationLifetime`	Delegacja nasłuchiwania na zamknięcie aplikacji (np. sygnał `SIGTERM` z kontenera Docker) do natywnego mechanizmu platformy.

Głębokie Nurkowanie: Generic Host i mechanika cyklu życia

Koncepcja Generic Hosta opiera się na stworzeniu jednolitego środowiska startowego, w którym aplikacja inicjalizuje swoje podsystemy zanim zacznie przyjmować jakiekolwiek żądania. Proces uruchamiania jest dwufazowy: najpierw konfigurowane są zależności (poprzez `WebApplicationBuilder`), a następnie budowany jest sam Host, który orkiestruje wykonaniem oprogramowania pośredniczącego (Middleware) oraz uruchamia usługi w tle.

Zarządzanie cyklem życia za pomocą Hosta wprowadza spójność we frameworku. Host odpowiada m.in. za: 1. **Inicjalizację serwera HTTP:** Uruchomienie Kestrel i otwarcie portów sieciowych. 2. **Uruchamianie usług tła:** Asynchroniczne wywoływanie implementacji `IHostedService` (klasycznych *Workerów*), co z punktu widzenia inżynierów stanowi bezpieczną alternatywę dla niestabilnych obiektów `Thread` znanych z .NET 4.8. 3. **Płynne wyłączanie (Graceful Shutdown):** Po otrzymaniu sygnału o zamknięciu, Generic Host wstrzymuje przyjmowanie nowego ruchu, pozwala aktualnym żądaniom na dokończenie pracy w ograniczonym czasie i bezpiecznie zamyka usługi w tle oraz połączenia z bazą danych.

Dobre Praktyki i Antywzorce

- **Antywzorzec:** Ręczne tworzenie wątków za pomocą `new Thread()` lub `Task.Run()` dla procesów długotrwających w tle (np. synchronizacji danych).
  - *Zagrożenie:* Wątki te nie są świadome cyklu życia Generic Hosta. Podczas wdrażania nowej wersji aplikacji, Host zakończy pracę, a niekontrolowane wątki mogą zostać natychmiast ubite (hard kill), co prowadzi do uszkodzenia danych.

- **Dobra praktyka:** Implementacja procesów pracujących w tle wyłącznie poprzez dziedziczenie po klasie `BackgroundService` (implementującej `IHostedService`). Dzięki temu Host Ogólny poprawnie zarządza ich asynchronicznym startem i bezpiecznym zamknięciem przy użyciu struktury `CancellationToken`.

Laboratorium kodu: Cykl życia usługi tła pod kontrolą Generic Hosta

Poniższy fragment kodu z wykorzystaniem C# 10+ demonstruje nowoczesną metodę inicjalizacji Generic Hosta (`WebApplication.CreateBuilder`), który zarządza nie tylko interfejsem Blazor, ale również dedykowaną usługą w tle. Takie podejście całkowicie eliminuje potrzebę ręcznego kontrolowania cyklu życia wątków.

```
using Microsoft.Extensions.Hosting;
using Microsoft.Extensions.Logging;

// 1. Inicjalizacja budowniczego Generic Hosta w .NET 10
var builder = WebApplication.CreateBuilder(args);

// Rejestracja komponentów UI (Blazor)
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();

// 2. Rejestracja usługi tła zarządzanej przez cykl życia Hosta
builder.Services.AddHostedService<DataSynchronizationWorker>();

var app = builder.Build();

// 3. Konfiguracja bezstanowego potoku żądań (Middleware)
if (!app.Environment.IsDevelopment())
{
    app.UseExceptionHandler("/Error", createScopeForErrors: true);
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles();
app.UseAntiforgery();

app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();

// 4. Uruchomienie Hosta - Kestrel zaczyna nasłuch, a usługi w tle zostają
        zainicjalizowane
app.Run();


// --- Definicja Worker'a kontrolowanego przez cykl życia Hosta ---
public class DataSynchronizationWorker : BackgroundService
{
    private readonly ILogger<DataSynchronizationWorker> _logger;

    // Primary Constructor injects dependencies
    public DataSynchronizationWorker(ILogger<DataSynchronizationWorker> logger)
```

```
    {
        _logger = logger;
    }

    // Metoda wykonywana automatycznie podczas startu Hosta
    protected override async Task ExecuteAsync(CancellationToken stoppingToken)
    {
        _logger.LogInformation("Usługa tła uruchomiona. Integracja z cyklem życia
        powiodła się.");

        // Pętla szanująca Graceful Shutdown Hosta
        while (!stoppingToken.IsCancellationRequested)
        {
            _logger.LogInformation("Synchronizacja danych w tle...");

            // Asynchroniczne opóźnienie - brak blokowania puli wątków!
            await Task.Delay(TimeSpan.FromMinutes(5), stoppingToken);
        }

        _logger.LogInformation("Host zasygnalizował zakończenie pracy. Usługa tła
        bezpiecznie wstrzymana.");
    }
}
```

Wnioski architektoniczne

Zintegrowanie mechanizmu **Generic Host** stanowi absolutny przełom dla inżynierów wywodzących się ze starszych rozwiązań desktopowych. Rezygnacja z ukrytych, natywnych procesów Windows na rzecz jawnej, opartej na interfejsach kontroli czasu życia (App Lifetime Management) radykalnie upraszcza obsługę operacji takich jak płynne uruchamianie i zatrzymywanie serwera sieciowego, zwalnianie pamięci w obrębie kontenera wstrzykiwania zależności (DI) czy logowanie błędów.

W .NET 10 to środowisko serwerowe dyktuje zasady bytu obiektów w aplikacji bezstanowej (Stateless). Zastosowanie wzorca `IHostedService` do obsługi operacji zaplecza umożliwia konstruowanie wysoce stabilnych systemów, w których komponenty Blazor oraz usługi logiczne wspólnie i bezpiecznie dzielą ten sam cykl życia, nadzorowany asynchronicznie przez jeden węzeł Hosta.

Wbudowane wstrzykiwanie zależności (DI) w architekturze Generic Host

Wstęp merytoryczny: Wbudowane wstrzykiwanie zależności pod kontrolą Hosta

Przejście z architektury **.NET Framework 4.8** na nowoczesny ekosystem **.NET 10** wymusza zmianę sposobu zarządzania cyklem życia obiektów i ich zależnościami. W klasycznych aplikacjach desktopowych (WPF, WinForms) wstrzykiwanie zależności (Dependency Injection, DI) było mechanizmem opcjonalnym, często dodawanym za pomocą zewnętrznych bibliotek (np. Autofac, Ninject). W architekturze ASP.NET Core wbudowane wstrzykiwanie zależności stanowi nierozerwalny fundament platformy.

Centralnym punktem zarządzającym wstrzykiwaniem zależności jest **Generic Host** (Host Ogólny), najczęściej powoływany za pomocą instancji `WebApplicationBuilder`. Host hermetyzuje wszystkie zasoby aplikacji: serwer HTTP, konfigurację, potok oprogramowania pośredniczącego (Middleware) oraz centralny kontener DI. Oznacza to, że w środowisku webowym to nie system operacyjny Windows ani kod interfejsu użytkownika zarządza pamięcią obiektów, lecz wysoce zoptymalizowana infrastruktura Hosta.

Analiza Porównawcza: Zarządzanie zależnościami (Desktop vs Web)

Koncepcja Desktop (.NET 4.8)	Odpowiednik w .NET 10 (Generic Host)	Charakter zmiany architektonicznej
Ręczne powoływanie obiektów (`new`)	**Kontener DI (`builder.Services`)**	Odrzucenie ścisłego powiązania (tight coupling). Aplikacja polega na odwróceniu kontroli (IoC), a Host wstrzykuje instancje do konstruktorów.

Koncepcja Desktop (.NET 4.8)	Odpowiednik w .NET 10 (Generic Host)	Charakter zmiany architektonicznej
Zmienne statyczne / Wzorzec Singleton	**Rejestracja o cyklu życia (Singleton/Scoped)**	Eliminacja globalnego stanu aplikacji (Stateful) na rzecz zarządzania bytem obiektów w obrębie cyklu życia bezstanowego żądania HTTP (Stateless).
Zewnętrzne biblioteki DI	**Wbudowane** `Microsoft.Extensions.DependencyInjection`	Standaryzacja. Mechanizm DI jest zintegrowany bezpośrednio wewnątrz Hosta i dostarczany natywnie przez framework.
Ścisłe zależności między warstwami	**Wstrzykiwanie przez konstruktor lub** `@inject`	Logika biznesowa jest przekazywana do nowoczesnych kontrolerów lub komponentów Blazor wyłącznie przez mechanizmy wbudowane w platformę.

Głębokie Nurkowanie (Deep Dive): Anatomia DI w Generic Host

W modelu ASP.NET Core proces inicjalizacji i orkiestracji działania aplikacji opiera się na dwóch fazach budowy Hosta. W fazie początkowej obiekt `WebApplicationBuilder` udostępnia właściwość `Services` reprezentującą kolekcję usług (`IServiceCollection`). To na tym etapie infrastruktura dodaje logikę dostępu do bazy danych, logowania, autoryzacji oraz renderowania komponentów interfejsu.

Kiedy programista wywołuje metodę `builder.Build()`, Host zatrzaskuje konfigurację, generuje docelowy kontener DI (`IServiceProvider`) i konfiguruje środowisko uruchomieniowe wraz z wieloplatformowym serwerem HTTP Kestrel. Od tego momentu każda składowa potoku Middleware, każdy kontroler oraz każdy

komponent Blazor otrzymuje swoje zależności wyłącznie w sposób automatyczny na etapie wykonania. Model ten eliminuje narzut związany z manualnym przekazywaniem parametrów systemowych w głąb drzewa wywołań.

Dobre Praktyki i Antywzorce

- **Antywzorzec: Ręczna rezolucja usług:** Pobieranie instancji poprzez anty-wzorzec *Service Locator* (np. bezpośrednie odpytywanie `ServiceProvider` w warstwie logiki) łamie spójność kontroli sprawowanej przez Hosta i ogranicza przejrzystość.
- **Dobra Praktyka: Wstrzykiwanie przez konstruktor (Constructor Injection):** Zależności powinny być zawsze deklarowane jawnie jako parametry konstruktora (lub poprzez dyrektywę `@inject` w plikach `.razor`). Gwarantuje to testowalność i przejrzystość wymogów danej klasy, co ułatwia zarządzanie dostarczanymi zależnościami.

Laboratorium kodu: Rejestracja usług i ich użycie w architekturze Hosta

Poniższy przykład demonstruje ewolucję inicjalizacji na poziomie punktu wejścia aplikacji. Użyto w nim wzorca Minimal API, gdzie `Generic Host` ładuje mechanizmy infrastruktury z jednego punktu, pozwalając na precyzyjne odizolowanie bazy danych od interfejsu UI.

```
// Program.cs w .NET 10 (Konfiguracja Hosta)
using Microsoft.EntityFrameworkCore;
using ModernApp.Data;

// 1. Inicjalizacja Hosta i infrastruktury
var builder = WebApplication.CreateBuilder(args);

// 2. Rejestracja wbudowanego mechanizmu wstrzykiwania zależności (DI)
// Usługi środowiskowe (np. konfiguracja JSON) są dostępne od razu z poziomu Hosta
builder.Services.AddDbContextFactory<AppDbContext>(options =>
    options.UseSqlServer(builder.Configuration.GetConnectionString("DefaultConnection")
    ?? throw new InvalidOperationException("Brak konfiguracji bazy danych."))); //

// Rejestracja frameworka komponentów i obsługi cyklu życia UI
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents(); //

// 3. Zbudowanie Hosta — zatrzaśnięcie kontenera DI
var app = builder.Build(); //

// (Konfiguracja bezstanowego potoku Middleware...)
app.UseHttpsRedirection(); //
app.UseAntiforgery(); //

app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode(); //
```

```
// 4. Start Hosta i serwera Kestrel
app.Run(); //
```

Zastosowanie wstrzykniętych zasobów w asynchronicznym środowisku webowym za pomocą dedykowanej dyrektywy frameworka:

```
@* TasksList.razor *@
@page "/tasks"
@using ModernApp.Data
@using Microsoft.EntityFrameworkCore

@* Deklaratywne żądanie wstrzyknięcia usługi utrzymywanej przez Generic Host *@
@inject IDbContextFactory<AppDbContext> DbFactory

<div class="task-board">
    @if (_tasks == null)
    {
        <p><em>Pobieranie danych ze strumienia DI...</em></p>
    }
    else
    {
        <table class="table">
            @foreach (var task in _tasks)
            {
                <tr>
                    <td>@task.Title</td>
                </tr>
            }
        </table>
    }
</div>

@code {
    private List<TaskDto>? _tasks;

    // Metoda cyklu życia, w której asynchronicznie zużywamy zasoby dostarczone z
        Hosta
    protected override async Task OnInitializedAsync()
    {
        await using var context = DbFactory.CreateDbContext();
        _tasks = await context.Tasks.ToListAsync();
    }
}
```

Wnioski architektoniczne

Przesunięcie zarządzania bytem i instancjonowaniem obiektów z poziomu kodu sterowanego zdarzeniami w architekturze Desktop (np. powiązanej z cyklem życia kontrolki w WPF), bezpośrednio do wbudowanego, centralnego mechanizmu kontenera DI zintegrowanego z modelem **Generic Host**, pozwala na radykalne podniesienie testowalności i skalowalności. Architektura w .NET 10 zapobiega wyciekom pamięci dzięki standaryzowanym cyklom życia obiektów (Transient, Scoped, Singleton) zarządzanym i powiązanym z ramami czasowymi żądań HTTP

przetwarzanych przez Middleware. Umożliwia to zwinne budowanie i wdrażanie odseparowanej, ustandaryzowanej logiki aplikacji.

Zarządzanie Konfiguracją (appsettings.json) w Architekturze Generic Host

Wstęp merytoryczny: Ewolucja zarządzania konfiguracją w ekosystemie .NET

W procesie transformacji aplikacji desktopowych z **.NET Framework 4.8** do nowoczesnego środowiska **.NET 10**, zmianie ulega nie tylko sposób renderowania interfejsu, ale również mechanika zarządzania ustawieniami aplikacji. W starym modelu architektonicznym parametry konfiguracyjne przechowywano w ciężkich, płaskich plikach XML (`App.config` dla WPF/WinForms lub `Web.config`), z których odczytywano je najczęściej za pomocą statycznej klasy `ConfigurationManager`.

W ASP.NET Core oraz nowym modelu uruchomieniowym odpowiedzialność za konfigurację przejmuje **Generic Host** (Host Ogólny). Infrastruktura Hosta domyślnie ładuje i agreguje ustawienia z wielu źródeł, w tym przede wszystkim z nowoczesnego, hierarchicznego pliku `appsettings.json`. Podejście to całkowicie rezygnuje ze statycznego, globalnego dostępu do stanu aplikacji na rzecz wbudowanego wstrzykiwania zależności (DI).

Analiza Porównawcza: Zarządzanie ustawieniami (Desktop 4.8 vs Web .NET 10)

Koncepcja Desktop (.NET 4.8)	Odpowiednik w .NET 10 (Generic Host)	Charakter zmiany architektonicznej
Plik `App.config` / `Web.config` (XML)	**Plik `appsettings.json` (JSON)**	Przejście z płaskich struktur XML z atrybutami, na elastyczne i czytelne struktury hierarchiczne (zagnieżdżone obiekty JSON).

Koncepcja Desktop (.NET 4.8)	Odpowiednik w .NET 10 (Generic Host)	Charakter zmiany architektonicznej
Statyczna klasa `ConfigurationManager`	**Interfejs `IConfiguration` i Kontener DI**	Wyeliminowanie wzorca Service Locator oraz globalnego stanu na rzecz wstrzykiwania zależności, ściśle zarządzanego przez Generic Host.
Brak wbudowanych profili środowiskowych	**`appsettings.Development.json` / Zmienne Środowiskowe**	Host automatycznie podmienia wartości konfiguracji bazując na aktywnym środowisku (np. Development, Production) bez potrzeby stosowania transformacji XML.
Ręczne rzutowanie typów (`int.Parse`)	**Options Pattern (`IOptions<T>`)**	Automatyczne mapowanie sekcji plików JSON na silnie typowane klasy (POCO), wstrzykiwane bezpośrednio do logiki biznesowej.

Głębokie Nurkowanie (Deep Dive): Integracja konfiguracji z Generic Host

W architekturze ASP.NET Core, plik `appsettings.json` nie jest samodzielnym bytem, lecz integralną częścią procesu budowania Hosta. Gdy w pliku startowym wywołujemy konstrukcję `WebApplication.CreateBuilder(args)`, Generic Host pod spodem konfiguruje całą infrastrukturę ładowania ustawień.

Proces ten odbywa się warstwowo. Host odczytuje konfigurację z określonego łańcucha dostawców (Configuration Providers) w ustalonej kolejności: 1. Podstawowy plik `appsettings.json`. 2. Plik środowiskowy, np. `appsettings.Development.json` lub `appsettings.Production.json`. 3. Zmienne środowiskowe (Environment Variables) systemu operacyjnego, które zawsze nadpisują wartości z plików JSON. 4. Argumenty wiersza poleceń.

Dzięki temu system jest naturalnie przystosowany do orkiestracji kontenerów (np. Docker, Kubernetes), w których zmienne środowiskowe sterują zachowaniem Hosta bez konieczności re-kompilacji aplikacji czy podmiany plików konfiguracyjnych.

Architektura systemowa: Wzorzec Opcji (Options Pattern)

Bezpośrednie używanie interfejsu `IConfiguration` w klasach biznesowych, polegające na odpytywaniu go ciągami znaków (np. `config["Smtp:Port"]`), jest w nowoczesnym .NET antywzorcem prowadzącym do tzw. *magic strings*.

Zamiast tego, framework promuje użycie **Options Pattern**. Architektura ta polega na stworzeniu klasy C# idealnie odzwierciedlającej strukturę wycinka pliku `appsettings.json`. Następnie, za pomocą kontenera usług Hosta (DI), wiążemy (bindujemy) sekcję pliku JSON z tą klasą. Wymusza to silne typowanie konfiguracji w całym rurociągu żądań HTTP i eliminuje rozproszenie danych aplikacyjnych.

Dobre Praktyki i Antywzorce

Proces migracji wymusza rewizję dawnych nawyków konfiguracji środowisk deweloperskich i produkcyjnych:

- **Antywzorzec: Przechowywanie tajemnic w `appsettings.json`.** Przechowywanie kluczy do zewnętrznych API, haseł do bazy danych czy certyfikatów produkcyjnych jawnym tekstem w konfiguracji zapisanej w repozytorium źródłowym.
- **Dobra praktyka:** Parametry połączeń (Connection Strings) na maszynie programisty należy umieszczać w specjalnym lokalnym mechanizmie wbudowanym w Hosta – *Secret Manager* (User Secrets). Na docelowym środowisku produkcyjnym korzystamy natomiast z usług do bezpiecznego przechowywania tajemnic, np. **Azure Key Vault**. Generic Host automatycznie podepnie te źródła jako bezpieczne nadpisania dla pliku konfiguracyjnego.

Laboratorium kodu: Konfiguracja i Wzorzec Opcji

Poniższe bloki demonstrują prawidłową konfigurację w .NET 10, która na etapie startu Hosta wiąże właściwości z sekcją pliku JSON, wstrzykując je jako silnie typowaną klasę do logiki aplikacji.

1. Plik `appsettings.json`: Hierarchiczna struktura zastępująca zawiłe bloki znane z dawnego `<appSettings>`.

```
{
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  },
  "SystemSettings": {
```

```
    "FeatureToggle": true,
    "MaxRetryAttempts": 3
  }
}
```

2. Klasa Opcji (POCO): Obiekt modelujący ustawienia biznesowe.

```
namespace ModernApp.Configuration;

public class SystemSettingsOptions
{
    // Konwencja nazewnicza ściśle odpowiada polom z pliku JSON
    public bool FeatureToggle { get; set; }
    public int MaxRetryAttempts { get; set; }
}
```

3. Inicjalizacja Hosta (`Program.cs`): Rejestracja opcji bezpośrednio z poziomu kontenera DI budowniczego. Wzorzec ten izoluje `IConfiguration` tylko do momentu startu aplikacji.

```
using ModernApp.Configuration;

var builder = WebApplication.CreateBuilder(args);

// Rejestracja sekcji JSON i wstrzyknięcie jako IOptions<SystemSettingsOptions>
builder.Services.Configure<SystemSettingsOptions>(
    builder.Configuration.GetSection("SystemSettings"));

builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();

var app = builder.Build();
// (Rurociąg Middleware...)
app.Run();
```

4. Wykorzystanie w serwisie lub komponencie Blazor: Logika biznesowa otrzymuje gotowy obiekt przy użyciu `IOptions<T>`.

```
// SystemService.cs - rejestrowany w DI
using Microsoft.Extensions.Options;

public class SystemService
{
    private readonly SystemSettingsOptions _settings;

    // Klasyczne, poprawne wstrzykiwanie w architekturze .NET 10
    public SystemService(IOptions<SystemSettingsOptions> options)
    {
        _settings = options.Value;
    }

    public void PerformOperation()
    {
        if (_settings.FeatureToggle)
        {
```

```
            // Nowa funkcjonalność uruchomiona z konfiguracji Hosta
            for (int i = 0; i < _settings.MaxRetryAttempts; i++) { /*...*/ }
        }
    }
}
```

Wnioski architektoniczne

Porzucenie pliku `App.config` i interfejsów systemu Windows na rzecz **Generic Hosta** przetwarzającego `appsettings.json` to milowy krok w kierunku budowania skalowalnych i bezpiecznych aplikacji rozproszonych. W środowisku **.NET 10** konfiguracja jest mechanizmem odseparowanym, który naturalnie współpracuje z kontenerem *Dependency Injection* oraz systemami wdrażania (CI/CD). Programiści, zamiast siłować się z wczytywaniem i parsowaniem typów prostych (często rzucającymi wyjątkami podczas startu), mogą swobodnie pracować w warstwach biznesowych z użyciem walidowanych i bezstanowych klas wzorca Options, nad którymi pieczę operacyjną sprawuje potok uruchomieniowy.

Serwery Web

Wstęp merytoryczny: Serwery Web jako silnik aplikacji .NET 10

W architekturze oprogramowania desktopowego opartego na **.NET Framework 4.8**, środowiskiem wykonawczym i zarządcą pętli komunikatów był bezpośrednio system operacyjny Windows. Przejście do ekosystemu **ASP.NET Core** oznacza całkowitą zmianę tego paradygmatu. W architekturze webowej punktem styku między światem zewnętrznym a logiką biznesową staje się zintegrowany **serwer HTTP**. Działa on jako główny nasłuchiwacz, który odbiera przychodzące pakiety sieciowe i tłumaczy je na abstrakcyjny obiekt `HttpContext`, przekazywany następnie do potoku oprogramowania pośredniczącego (Middleware) zarządzanego przez Generic Host.

Zrozumienie mechaniki wbudowanych serwerów jest krytyczne dla inżynierów migrujących systemy z modelu grubego klienta. Rozwiązania webowe nie posiadają pętli zdarzeń interfejsu graficznego (jak `Dispatcher` w WPF) – ich wydajność i responsywność opiera się w całości na asynchronicznej obsłudze tysięcy współbieżnych żądań HTTP.

Analiza Porównawcza: Pętla komunikatów vs Serwer HTTP

Koncepcja Desktop (.NET 4.8)	Odpowiednik w ASP.NET Core (.NET 10)	Charakter zmiany architektonicznej
`Application.Run()` / Pętla Win32	**Serwer Kestrel (`app.Run()`)**	Porzucenie systemowej pętli zdarzeń GUI na rzecz wysoce zoptymalizowanego, sieciowego nasłuchu na portach TCP/HTTP.
Ścisłe powiązanie z OS (Windows)	**Wieloplatformowość (Cross-platform)**	Silnik aplikacji (np. Kestrel) działa natywnie pod kontrolą systemów Windows, macOS oraz Linux.

Koncepcja Desktop (.NET 4.8)	Odpowiednik w ASP.NET Core (.NET 10)	Charakter zmiany architektonicznej
Pamięć współdzielona procesu	**Obiekty `HttpContext`**	Każda interakcja użytkownika to osobne żądanie sieciowe z własnym stanem, hermetyzowanym w kontekście HTTP.
Hosting przez systemowy IIS (dla WCF/Web)	**Samodzielny (Self-hosted) Generic Host**	Aplikacja webowa zawiera własny serwer (Kestrel) i może działać niezależnie od serwerów aplikacyjnych, często za tzw. *Reverse Proxy*.

Głębokie Nurkowanie (Deep Dive): Implementacje serwerów w ASP.NET Core

Aplikacje oparte na .NET 10 wykorzystują interfejsy serwera HTTP do obsługi ruchu sieciowego. Architektura ASP.NET Core dostarcza kilka wbudowanych implementacji, które programista może wybrać w zależności od specyfiki wdrożenia:

1. Serwer Kestrel

Kestrel to domyślny, wieloplatformowy serwer sieciowy o potężnej wydajności. W przeciwieństwie do ciężkich serwerów aplikacyjnych z przeszłości, Kestrel został zaprojektowany z myślą o maksymalnej przepustowości. Może on działać jako publicznie dostępny serwer brzegowy (edge server), wystawiony bezpośrednio na dostęp do Internetu. W zastosowaniach korporacyjnych Kestrel najczęściej operuje w konfiguracji odwrotnego proxy (Reverse Proxy), działając za serwerami takimi jak IIS, Nginx lub Apache. Zapewnia to dodatkową warstwę bezpieczeństwa i równoważenia obciążenia (Load Balancing).

2. IIS HTTP Server

Jest to specjalistyczny serwer przeznaczony wyłącznie dla środowisk Windows korzystających z usług IIS (Internet Information Services). W tym modelu aplikacja ASP.NET Core oraz proces roboczy IIS działają w tej samej przestrzeni pamięci (in-process), co eliminuje narzut komunikacji sieciowej między serwerem proxy a instancją Kestrel.

3. Serwer HTTP.sys

Implementacja przeznaczona dla systemu Windows, która nie wykorzystuje IIS. Opiera się bezpośrednio na sterowniku jądra systemu Windows (`Http.sys`). Jest używana głównie w scenariuszach wymagających współdzielenia tego samego portu IP przez wiele procesów lub przy uwierzytelnianiu zintegrowanym na poziomie systemu operacyjnego.

Dobre Praktyki i Antywzorce

- **Antywzorzec: Blokowanie wątków serwera (Synchronous Blocking):** Używanie konstrukcji takich jak `Thread.Sleep()` lub wywoływanie `Task.Result` w obrębie obsługi żądania HTTP. Kestrel opiera się na relatywnie małej puli wątków asynchronicznych (Thread Pool). Zablokowanie wątku w oczekiwaniu na operację wejścia/wyjścia (I/O) powoduje zjawisko “Thread Starvation”, które błyskawicznie paraliżuje serwer.
- **Dobra Praktyka: Wdrożenia kontenerowe:** Kestrel sprawdza się idealnie w nowoczesnej architekturze mikrousług. Powszechną praktyką w .NET 10 jest pakowanie aplikacji wraz z jej zintegrowanym serwerem Kestrel do kontenera Docker (opartego na systemie Linux), co zapewnia pełną powtarzalność środowiska między deweloperem a produkcją.

Laboratorium kodu: Inicjalizacja serwera Kestrel

W nowoczesnym środowisku uruchomieniowym .NET instancjonowanie i konfiguracja serwera odbywa się niemal przezroczyście dzięki zastosowaniu mechanizmu `WebApplicationBuilder`. Poniższy fragment ukazuje, jak Host automatycznie konfiguruje Kestrel jako wbudowany serwer HTTP i otwiera pętlę nasłuchującą na żądania, wprowadzając obiekt `HttpContext` do potoku:

```
using Microsoft.AspNetCore.Builder;
using Microsoft.Extensions.Hosting;

// Inicjalizacja budowniczego konfiguruje m.in. serwer Kestrel oraz ładuje
        appsettings.json
var builder = WebApplication.CreateBuilder(args);

// Rejestracja usług w kontenerze Dependency Injection (Fundament architektury
        Stateless)
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();

// Budowa instancji Hosta i uformowanie serwera HTTP
var app = builder.Build();

// Konfiguracja potoku żądań Middleware - tu Kestrel przekazuje obiekt HttpContext
if (!app.Environment.IsDevelopment())
{
    // Globalne przechwytywanie błędów serwera
```

```
    app.UseExceptionHandler("/Error", createScopeForErrors: true);
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles(); // Serwowanie plików statycznych (.js, .css)
app.UseAntiforgery();

app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();

// Start serwera Kestrel. Zastępuje to pętlę "Application.Run()" z Windows Forms.
// Aplikacja przechodzi w asynchroniczny stan nasłuchiwania HTTP na określonych
        portach.
app.Run();
```

Wnioski architektoniczne

Przejście z aplikacji okienkowych na architekturę ASP.NET Core uwalnia oprogramowanie z restrykcji operacyjnych systemu operacyjnego klienta, przenosząc ciężar na wydajną infrastrukturę serwerową opartą o Kestrel. Dzięki temu inżynierowie mogą wdrażać logikę w dowolnym środowisku (Windows, Linux, kontenery Docker) z nieosiągalną wcześniej w .NET Framework skalowalnością.

Należy jednak pamiętać, że serwer webowy przetwarza setki lub tysiące bezstanowych żądań niemal jednocześnie. Wymaga to od programistów wywodzących się ze starszych technologii Microsoftu bezwzględnego stosowania wstrzykiwania zależności (DI) oraz programowania w pełni asynchronicznego, gwarantującego optymalne wykorzystanie zasobów przydzielonych dla serwera HTTP.

Rurociąg Middleware

Wstęp merytoryczny: Rurociąg Middleware jako kręgosłup aplikacji webowej

W procesie transformacji z aplikacji desktopowych opartych na **.NET Framework 4.8** do nowoczesnego ekosystemu **ASP.NET Core** i frameworka **Blazor**, jednym z najważniejszych fundamentów architektonicznych jest zrozumienie sposobu przetwarzania żądań. W klasycznych aplikacjach WPF czy WinForms interakcja opierała się na systemowej pętli komunikatów (Message Loop) oraz zdarzeniach cyklu życia okna. Z kolei starsze aplikacje webowe korzystały z ciężkich bibliotek `System.Web.dll`, w których przetwarzanie zależało od modułów `HttpModules` i `HttpHandlers`.

W środowisku **.NET 10** architektura ta ulega całkowitemu spłaszczeniu i ujednoliceniu. Kestrel, wieloplatformowy serwer HTTP, odbiera pakiety sieciowe i natychmiast przekazuje obiekt `HttpContext` do **rurociągu oprogramowania pośredniczącego (Middleware Pipeline)**. Rurociąg ten to sekwencja wysoce zoptymalizowanych komponentów. Każdy z nich wykonuje operacje na kontekście żądania, a następnie podejmuje decyzję o przekazaniu sterowania do kolejnego elementu lub o natychmiastowym zakończeniu przetwarzania i zwróceniu odpowiedzi klientowi. Zrozumienie tej bezstanowej i kaskadowej architektury jest kluczowe dla inżynierów migrujących z systemów stanowych.

Analiza Porównawcza: Obsługa cyklu życia i żądań

Koncepcja Desktop / Legacy Web (4.8)	Odpowiednik w ASP.NET Core (.NET 10)	Charakter zmiany architektonicznej
`Application_DispatcherUnhandledException`	**Middleware Obsługi Błędów (`UseExceptionHandler`)**	Globalne przechwytywanie wyjątków przeniesione na początek rurociągu, chroniące serwer przed awarią i standaryzujące format odpowiedzi błędu.

Koncepcja Desktop / Legacy Web (4.8)	Odpowiednik w ASP.NET Core (.NET 10)	Charakter zmiany architektonicznej
`HttpModules` i `HttpHandlers` (`System.Web`)	**Komponenty Middleware**	Rezygnacja ze zmonolityzowanej biblioteki na rzecz modularnych, lekkich delegatów `RequestDelegate` tworzących łańcuch wywołań.
Pętla komunikatów systemu Windows	**Asynchroniczny potok** `next()`	Każde żądanie jest izolowanym procesem bezstanowym (Stateless). Komponent może przerwać ten potok (zwarcie rurociągu), np. serwując plik statyczny.
Atrybuty i zdarzenia globalne kontrolki	**Filtry (np.** `IAsyncResultFilter`**) vs Middleware**	Middleware dotyczy każdego żądania HTTP w aplikacji. Filtry (Filters) są specyficzne dla cyklu życia warstwy kontrolerów (MVC) lub stron Razor.

Głębokie Nurkowanie (Deep Dive): Anatomia Rurociągu

Kaskadowe przekazywanie sterowania (RequestDelegate)

Każdy komponent Middleware to w rzeczywistości delegat `RequestDelegate`, który przyjmuje `HttpContext` i zwraca zadanie (`Task`). Kiedy żądanie trafia do rurociągu, komponent wykonuje swoją logikę przed asynchronicznym wywołaniem delegata `_next(context)`. Oznacza to, że Middleware może działać dwufazowo: modyfikować żądanie przed przekazaniem go dalej, a następnie manipulować odpowiedzią, gdy wywołanie `next` wróci z dalszych części systemu.

Zwarcie rurociągu (Short-Circuiting)

Jeśli dany komponent zdecyduje, że sam w pełni potrafi obsłużyć żądanie, nie wywołuje metody `next()`. Taka sytuacja nazywana jest zwarciem rurociągu.

Doskonałym przykładem jest wbudowane oprogramowanie `StaticFilesMiddleware` (`app.UseStaticFiles()`). Jeśli klient poprosi o plik obrazu czy arkusz CSS z folderu `wwwroot`, middleware odsyła ten plik w odpowiedzi i przerywa potok – żądanie nigdy nie dotrze do mechanizmów autoryzacji czy renderowania Blazora.

Globalny zasięg a Filtry (Filters)

Należy stanowczo rozróżnić Middleware od Filtrów w .NET 10. Middleware wykonuje się absolutnie dla każdego żądania (np. śledzenie, CORS, statyczne pliki, przechwytywanie błędów). Filtry (takie jak `ActionFilter` czy `ExceptionFilter`) działają znacznie głębiej – aktywują się dopiero, gdy proces wejdzie w kontekst warstwy prezentacji, czyli kontrolerów lub stron Razor, zapewniając granulację zachowań na poziomie konkretnych metod lub klas.

Architektura systemowa: Rejestracja i implementacja

W nowoczesnym podejściu preferowane jest tworzenie Middleware za pomocą interfejsu `IMiddleware` zamiast rozwiązań konwencyjnych (historycznie opierających się na konstruktorach bez interfejsu).

W podejściu opartym na konwencji komponent Middleware jest rejestrowany jako tzw. Singleton, przez co wstrzykiwane przez niego w konstruktorze zależności również muszą być Singletonami. Z kolei wykorzystanie natywnego interfejsu `IMiddleware` (wprowadzonego od wczesnych wersji ASP.NET Core 2.0) pozwala fabryce kontenera **Dependency Injection** na utworzenie dedykowanej instancji dla każdego pojedynczego żądania (Scoped lifetime). Oznacza to, że jest w 100% bezpieczne wstrzykiwanie do takiego Middleware usług o krótkim cyklu życia (np. kontekstu dostępu do danych).

Dobre Praktyki i Antywzorce

Proces definicji rurociągu to etap, na którym inżynierowie przyzwyczajeni do aplikacji monolitycznych nierzadko popełniają błędy związane z kolejnością operacji.

- **Antywzorzec: Błędna kolejność rejestracji.** Komponenty rejestrowane w obiekcie budowniczego w pliku startowym determinują kolejność wywołań. Dodanie obsługi wyjątków (`UseExceptionHandler`) na końcu rurociągu lub dodanie uwierzytelnienia (`UseAuthorization`) po warstwie renderowania tras (`MapRazorComponents`) złamie architekturę i narazi aplikację na luki bezpieczeństwa i wycieki błędów.

- **Dobra praktyka:** Pełna asynchroniczność w ciele Middleware. Wątki serwera Kestrel obsługują żądania. Jakiekolwiek blokowanie wykonania (np. za pomocą sygnatur `Wait()` lub instrukcji bez słowa kluczowego `await`) doprowadzi do wyczerpania puli wątków przy większym obciążeniu.

Laboratorium kodu: Wdrożenie silnie typowanego Middleware

Poniższy przykład przedstawia konstrukcję i rejestrację nowoczesnego Middleware służącego do logowania czasu wykonania żądania, w pełni zgodnego ze specyfikacją .NET 10 oraz opartą na wstrzykiwaniu zależności za pomocą interfejsu `IMiddleware`.

```
// 1. Definicja silnie typowanego Middleware o żywotności Scoped (per żądanie)
using System.Diagnostics;
using Microsoft.AspNetCore.Http;
using Microsoft.Extensions.Logging;

public class PerformanceTrackingMiddleware : IMiddleware
{
    private readonly ILogger<PerformanceTrackingMiddleware> _logger;

    // Usługi wstrzykiwane bezpiecznie z zachowaniem cyklu życia Scoped
    public PerformanceTrackingMiddleware(ILogger<PerformanceTrackingMiddleware>
        logger)
    {
        _logger = logger;
    }

    public async Task InvokeAsync(HttpContext context, RequestDelegate next)
    {
        var stopwatch = Stopwatch.StartNew();

        // Przekazanie sterowania w głąb rurociągu (do Blazora lub API)
        await next(context);

        stopwatch.Stop();

        // Logika powrotna wykonująca się po wygenerowaniu odpowiedzi przez system
        var isHtml = context.Response.ContentType?.ToLower().Contains("text/html");
        if (context.Response.StatusCode == 200 && isHtml.GetValueOrDefault())
        {
            _logger.LogInformation($"Ścieżka {context.Request.Path} wygenerowana w
        {stopwatch.ElapsedMilliseconds} ms.");
        }
    }
}
```

```
// 2. Punkt startowy aplikacji (Program.cs) - Integracja z Hosterem i rurociągiem
var builder = WebApplication.CreateBuilder(args);

// Rejestracja klas Middleware jako serwisów DI, aby fabryka mogła powołać
        instancje
builder.Services.AddScoped<PerformanceTrackingMiddleware>();
```

```
builder.Services.AddRazorComponents()
    .AddInteractiveServerComponents();

var app = builder.Build();

// --- FORMOWANIE RUROCIĄGU (Kolejność determinuje kaskadowość wykonania) ---

if (!app.Environment.IsDevelopment())
{
    // Obsługa błędów rejestrowana NAJPIERW
    app.UseExceptionHandler("/Error", createScopeForErrors: true);
    app.UseHsts();
}

app.UseHttpsRedirection();
app.UseStaticFiles(); // Ewentualne zwarcie rurociągu dla zasobów graficznych / CSS

// Nasz autorski Middleware dodany przed procesem routingu
app.UseMiddleware<PerformanceTrackingMiddleware>();

app.UseAntiforgery();

app.MapRazorComponents<App>()
    .AddInteractiveServerRenderMode();

app.Run(); // Kestrel zaczyna pętlę nasłuchującą
```

Wnioski architektoniczne

Porzucenie pętli zdarzeń znanej z desktopowych technologii Frameworka 4.8 na rzecz asynchronicznego Rurociągu Middleware narzuca na deweloperów zmianę paradygmatu postrzegania cyklu życia instancji. Aplikacje .NET 10 cechują się absolutną bezstanowością logiki HTTP. Oznacza to, że potok jest niezwykle wysoce skalowalny, a izolacja kodu w postaci dedykowanych komponentów przechwytujących `HttpContext` pozwala na konstrukcję lekkiej i bardzo czytelnej infrastruktury systemu. Granica odpowiedzialności staje się tu krystalicznie jasna: to, co ma charakter globalny, zamyka się w rurociągu (Middleware), a logika kontroli pojedynczych operacji przypisywana jest Filtrom (Filters) lub bezpośrednio deklaratywnym komponentom UI z rodziny Blazor.