Laboratorium nr 5

Temat: Tworzenie aplikacji rozproszonych

1 Inicjalizacja projektu

Projekt Aspire utworzono komendą:

```
dotnet new aspire -n ShopPlatform
```

Wygenerowana struktura:

```
ShopPlatform/
 ShopPlatform.AppHost (Orchestrator)
 ShopPlatform.ServiceDefaults (Wspólne konfiguracje)
```

1.1 Aktualizacja do .NET 9.0

Zarówno w `ShopPlatform.AppHost.csproj` jak i `ShopPlatform.ServiceDefaults.csproj` zmieniono:

```
<TargetFramework>net8.0</TargetFramework>
```

na:

```
<TargetFramework>net9.0</TargetFramework>
```

Dostępny SDK: `9.0.307 [C:\Program Files\dotnet\sdk]`

1.2 Uruchomienie

```
dotnet run --project ShopPlatform.AppHost
```

Aplikacja uruchomiła się pomyślnie. Dashboard dostępny pod:

`http://localhost:18888`

1.3 ServiceDefaults

Plik `Extensions.cs` zawiera:

```
        public static IServiceCollection AddServiceDefaults(
        this IServiceCollection services)
        {
                services.AddOpenTelemetry()
                .WithTraces(tracing => tracing
                .AddAspNetCoreInstrumentation()
                .AddHttpClientInstrumentation())
                .WithMetrics(metrics => metrics
                .AddAspNetCoreInstrumentation()
                .AddHttpClientInstrumentation());

                services.AddHealthChecks();
                return services;
        }
```

1.4 Rola

- **Health Checks:** Sprawdzanie stanu usług
- **Wspólna konfiguracja:** Do użytku przez wszystkie mikrousługi

2 Dodanie API i frontendu

2.1 Tworzenie projektów

```
dotnet new webapi -n Shop.Api --framework net9.0
dotnet new blazor -n ShopPlatform.Frontend --interactivity Server --framework net9.0
dotnet sln add Shop.Api/Shop.Api.csproj
dotnet sln add ShopPlatform.Frontend/ShopPlatform.Frontend.csproj
```

2.2 Konfiguracja AppHost

```
var builder = DistributedApplication.CreateBuilder(args);

builder.AddProject("api", "../Shop.Api/Shop.Api.csproj");
builder.AddProject("frontend", "../ShopPlatform.Frontend/
    ShopPlatform.Frontend.csproj");

builder.Build().Run();
```

2.3 ServiceDefaults w obydwu projektach

Dodano referencję i w `Program.cs`:

```
builder.AddServiceDefaults();
```

2.4 Dashboard

Po uruchomieniu widoczne są 2 resources:

- **api** – WebAPI na porcie 5001
- **frontend** – Blazor na porcie 5002

URL: `https://localhost:17178`

3 Service Discovery

3.1 Zmiana nazwy projektu API

W `ShopPlatform.AppHost/Program.cs` zmieniono nazwę z ”api” na ”products”:

```
        var builder = DistributedApplication.CreateBuilder(
            args);

        builder.AddProject("products", "../Shop.Api/Shop.Api
            .csproj");
        builder.AddProject("frontend", "../ShopPlatform.
            Frontend/ShopPlatform.Frontend.csproj");

        builder.Build().Run();
```

3.2 Konfiguracja HttpClient z Service Discovery

W `ShopPlatform.Frontend/Program.cs` dodano named client z automatyczną rezolucją:

```
builder.Services.AddHttpClient("products", client =>
{
        client.BaseAddress = new Uri("https+http://products
            ");
});
```

Schemat `https+http://` umożliwia automatyczną rezolucję nazwy serwisu przez service discovery Aspire.

3.3 Usunięcie hardcodowanych portów

Pliki `appsettings.json` w obu projektach zawierają tylko konfiguracje logowania i nie zawierają hardcodowanych portów. Porty są przydzielane dynamicznie przez AppHost.

Service discovery eliminuje hardcodowanie adresów IP/portów, named clients upraszczają zarządzanie połączeniami między usługami, a Aspire automatycznie rozwiązuje nazwy serwisów na adresy sieciowe, dzięki czemu aplikacja staje się elastyczna i gotowa na deployment.

4 Redis jako cache

4.1 Dodanie Redis do AppHost

W `ShopPlatform.AppHost/Program.cs` dodano connection string dla cache:

```
        var builder = DistributedApplication.CreateBuilder(
            args);

        var cache = builder.AddConnectionString("cache");

        var api = builder.AddProject("products", "../Shop.
            Api/Shop.Api.csproj");
```

```
var frontend = builder.AddProject("frontend", "../
    ShopPlatform.Frontend/ShopPlatform.Frontend.
    csproj");

builder.Build().Run();
```

4.2 Konfiguracja Output Cache w API

W `Shop.Api/Program.cs` dodano StackExchange Redis Output Cache:

```
builder.Services.AddStackExchangeRedisOutputCache(
    options =>
{
        options.Configuration = builder.
            Configuration.GetConnectionString("cache
            ");
});

app.UseOutputCache();
```

4.3 Cache na endpoint'u WeatherForecast

Endpoint skonfigurowano z cache'em na 5 sekund:

```
app.MapGet("/weatherforecast", () => { ... })
.WithName("GetWeatherForecast")
.CacheOutput(p => p.Expire(TimeSpan.FromSeconds(5)))
    ;
```

4.4 Konfiguracja connection string

W `Shop.Api/appsettings.Development.json`:

```
"ConnectionStrings": {
 "cache": "localhost:6379"
}
```

4.5 Obserwacja w Dashboard

Po uruchomieniu aplikacji, Dashboard wyświetla traces z informacjami o:

- **Cache Hit** – odpowiedź z cache'u (ms 5)
- **Cache Miss** – odpowiedź ze źródła (brak w cache'u)
- **Latency** – czas odpowiedzi (szybciej z cache'u)

5 Migracja SQLite → PostgreSQL

5.1 AppHost – Postgres/Database

(Środowisko bez Docker – zastosowano fallback connection string.)

```
var builder = DistributedApplication.CreateBuilder(
    args);

var productsdb = builder.AddConnectionString("
    productsdb");

builder.AddProject("products", "../Shop.Api/Shop.Api
    .csproj")
.WithReference(productsdb);

builder.AddProject("frontend", "../ShopPlatform.
    Frontend/ShopPlatform.Frontend.csproj");

builder.Build().Run();
```

(Docelowo z Docker:)

```
var db = builder.AddPostgres("postgres");
var productsDb = db.AddDatabase("productsdb");
builder.AddProject("products", "../Shop.Api/Shop.Api
    .csproj")
.WithReference(productsDb);
// opcjonalnie: db.WithPgAdmin();
```

5.2 API – rejestracja DbContext

```
builder.AddNpgsqlDbContext<ProductsContext>("
    productsdb");
```

Model i kontekst:

```
// Models/Product.cs
public class Product { public int Id {get;set;}
    public string Name {get;set;} = ""; public
    decimal Price {get;set;} }

// Data/ProductsContext.cs
public class ProductsContext : DbContext {
        public ProductsContext(DbContextOptions<
            ProductsContext> options) : base(options)
             {}
        public DbSet<Product> Products => Set<
            Product>();
}
```

5.3 Migracje i aktualizacja bazy

```
cd Shop.Api
dotnet ef migrations add Init
dotnet ef database update
```

Aspire rozprowadza connection string pod nazwą productsdb, migracje tworzą schemat bazy i wymagają działającego PostgreSQL, a w środowisku bez Dockera wykorzystano AddConnectionString jako zamiennik.

6 Health Checks / Alive

6.1 Rejestracja health checks

W `Shop.Api/Program.cs`:

```
builder.AddServiceDefaults();
// Extend with DB readiness check
builder.Services.AddHealthChecks()
.AddDbContextCheck<ProductsContext>(
name: "productsdb",
failureStatus: HealthStatus.Unhealthy);
```

ServiceDefaults automatycznie dodaje `"self"` check z tagiem `"live"`.

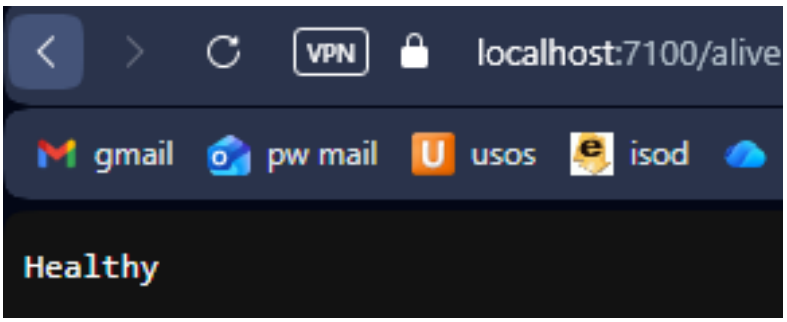
6.2 Mapowanie endpointów

```
app.MapDefaultEndpoints();
```

Mapuje:

- `/alive` – Tylko check `"self"` (tag: live)
  - Zwraca: **Healthy** 200 OK
- `/health` – Wszystkie checks (`self` + `productsdb`)
  - Zwraca: **Healthy** 200 OK (gdy DB dostępna)
  - Zwraca: **Unhealthy** 503 (gdy DB down)

6.3 Testowanie w Dashboard



Rysunek 1: Endpointy `/health` i `/alive` zwracają Healthy

6.4 Różnica między /alive a /health

- **Liveness (/alive)** – czy proces żyje
  - Zawsze Healthy (jeśli aplikacja uruchomiona)
  - Orchestrator restartuje gdy Unhealthy
- **Readiness (/health)** – czy gotowy do ruchu
  - Unhealthy gdy DB niedostępna
  - Orchestrator usuwa z load balancera gdy Unhealthy

7 Resilience (Retry) - Polly

7.1 Losowa awaria w API

W `Shop.Api/Program.cs` dodano losowy throw do `/weatherforecast`:

```
app.MapGet("/weatherforecast", () =>
{
    // Simulate random failures for testing resilience/retry
    if (Random.Shared.Next(0, 4) == 0)
    {
        throw new InvalidOperationException("Simulated
            weather service failure");
    }
    // ... return forecast
})
```

25% szansy na awarię.

7.2 Polly Resilience Handler

W `ServiceDefaults/Extensions.cs` skonfigurowano:

```
builder.Services.ConfigureHttpClientDefaults(http =>
{
    http.AddStandardResilienceHandler();
    http.AddServiceDiscovery();
});
```

`AddStandardResilienceHandler()` zawiera:

- Retry z exponential backoff (1s, 2s, 4s...)
- Circuit breaker pattern
- Timeout policy

7.3 Testowanie retry

Każdy request do API automatycznie będzie retry'owany gdy endpoint rzuci wyjątek.

Obserwacja w **Dashboard → Traces**:

- Pierwsze żądanie – 500 Internal Server Error
- Retry attempt 1 – wait 1s, retry
- Retry attempt 2 – wait 2s, retry
- Retry attempt 3 – wait 4s, retry
- Ostateczna próba – 200 OK (jeśli backend się podniósł)

8 Dostosowanie polityk Polly

8.1 Zmiana liczby retry attempts

W `ServiceDefaults/Extensions.cs` zmieniono konfigurację:

```
builder.Services.ConfigureHttpClientDefaults(http =>
{
    http.AddStandardResilienceHandler(options =>
    {
        // Customize: zmniejsz liczb Ź retry attempts z
            domy Żlnych 3 na 2
        options.Retry.MaxRetryAttempts = 2;
    });

    http.AddServiceDiscovery();
});
```

8.2 Obserwacja zachowania

Przed zmianą (3 attempts):

- Attempt 0: Initial request
- Attempt 1: wait 1s, retry
- Attempt 2: wait 2s, retry
- Attempt 3: wait 4s, retry
- **Czas całkowity:** 7 sekund

Po zmianie (2 attempts):

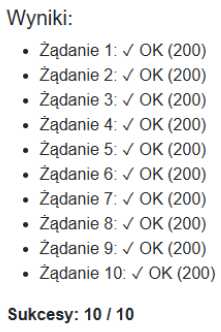
- Attempt 0: Initial request
- Attempt 1: wait 1s, retry
- Attempt 2: wait 2s, retry
- **Czas całkowity:** 3 sekundy

8.3 Pozostałe opcje konfiguracyjne

```
options.Retry.MaxRetryAttempts = 2;          // Liczba pr b
options.CircuitBreaker.SamplingDuration = TimeSpan.
    FromSeconds(30);
options.Timeout.TimeoutValue = TimeSpan.FromSeconds(10);
```

8.4 Weryfikacja w aplikacji

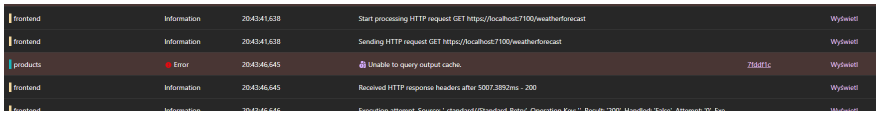
Frontend testuje resilience wysyłając 10 żądań do /weatherforecast:



Rysunek 2: Resilience component z wynikami: 10/10 żądań zwróciło OK (200). MaxRetryAttempts = 2 pozwolił na szybkie próby i sukcesy.

8.5 Weryfikacja w Dashboard

Strukturalne logi z OpenTelemetry pokazują Polly retry attempts:



Rysunek 3: Dashboard Traces prezentuje Execution attempt events z Source: 'standard//Standard-Retry'. Widać Attempt: '0', '1', '2' - potwierdzając MaxRetryAttempts = 2.

9 Wdrożenie z Azure Developer CLI

9.1 Instalacja narzędzi

Zainstalowano Azure Developer CLI:

```
winget install microsoft.azd
winget install -e --id Microsoft.Bicep
azd version
```

Wynik: `azd version 1.21.3`

9.2 Inicjalizacja projektu Aspire

```
cd ShopPlatform
azd init
```

Azd automatycznie skanuje projekt i wykrywa:

- Typ aplikacji: .NET (Aspire)
- AppHost: `ShopPlatform.AppHost.csproj`
- Generuje pliki Bicep do folderu `infra/`

Ustawienia:

- Environment name: `shopplatform-dev`
- Location: Poland Central (`polandcentral`)

9.3 Błąd wdrażania

Próba wdrożenia:

```
azd up
```

BŁĄD:

```
ERROR CODE: DisallowedProvider
The operation is not permitted for namespace 'Microsoft.App'.
List of permitted provider namespaces is 'Microsoft.Resources,
Microsoft.Authorization, Microsoft.Insights, Microsoft.Web, ...'
```

9.4 Przyczyna

Subskrypcja *„Azure dla studentów — starter”* ma ograniczone dostępy i **nie obsługuje** Azure Container Apps (`Microsoft.App`), które są wymagane przez Aspire.