**Slide 1**

გამარჯობა მეგობრებო, დღეს მინდა წარმოგიდგინოთ .NET Aspire სიახლე cloud dev ში. იგი მიზნად ისახავს ქლაუდ დეველოპმენტის პროცესის გაუმჯობესებას/ასწრაფებას.

**Slide 2**

.NET Aspire ახალი თულია მაიკროსოფტისგან, რომელიც ქლაუდ აპლიკაციების კეთებაში დაგვეხმარება. შეგვიძლია მას ისე შევხედოდ როგორც თულებისა და მიდგომების კომბინაციას, რომელიც ქლაუდ აპლიკაციების მარტივად კეთება/შენახვაში გვეხმარება. აქვე ხაზი გავუსვათ იმას რომ .NET Aspire ერთი შეხედვით შეიძლება ჰგავდეს K8 ს ან Openshift ს მაგრამ არც ერთთან ახლოსაც არაა, სრულიად სხვა დანიშნულება აქვს. Aspire არის მიდგომების და დეველოპერის ცხოვრების გამმარტივებელი თულების ერთობლიობა, როცა K8 არის ორკესტრაციის თული რომელიც სხვადასხვა მანქანებზე გაშვებული ჩვენი ფუნქციონალის მართვაში გვეხმარება.

როგორც უკვე აღვნიშნეთ Aspire მიზნად ისახავს ქლაუდ დეველოპმენტის დროს ხშირად წარმოქმნილი პრობლემების გადაჭრას. მისი ძირითადი ფოკუსია აპლიკაციები გადააქციოს უფრო სანდოდ, მარტივად მონიტორირებად და ადვილად დასაკონფიგურირებლად. Aspire ის მიდგომა ეხება იმ საერთო გამოწვევებს, რომლებსაც ვეჩეხებით ქლაუდ დევში.

Aspire ში ჩადებული იდეები და ფუნქციონალები, როგორც მისი შემქმნელები ამბობენ, მოდის მაიკროსოფტის დიდი პროექტების გამოცდილებიდან, რომლებსაც მილიონობით ადამიანი იყენებს. ესენია სერვისები, რომლებიც უნდა იყვნენ სანდო და რომლებსაც უნდა სეეძლოთ დიდი ლოადის ატანა. გამომდინარე აქედან, როგორც თავად გვეუბნებიან, .NET Aspire შექმნილია პრაქტიკაზე დაფუძვნებით რეალური ჩელენჯების გასამკლავებლად მზად.

რომ შევაჯამოთ, Aspire გვაძლევს ხელსაწყოებისა და პრაქტიკების ნაზავს, რომელიც გამოცდილია დატვირთულ და მოთხოვნად გარემოებში.

**Slide 3**

გავეცნოთ ძირითად ტექნოლოგიებს რომლებიც .NET Aspire ში გამოიყენება.

დავიწყოთ Health Check ებით, YARP ით, HTTP Client Factory ითა და GRPC ით. ეს ტექნოლოგიები ძირითადია. Health Check ები ამონიტორინგებენ აპლიკაციების სიცოცხლისუნარიანობას. YARP (Yet Another Reverse Proxy) გვეხმარება ტრაფიკის განაწილებაში რომ სწორმა აპლიკაციამ მიიღოს რექვესთი. HTTP Client Factoy ამარტივებს http client ებთან მუშაობას და მათ მენეჯმენტს. საბოლოოდ gRPC რომელიც მნიშვნელოვანია სერვისებს შორის ინფორმაციის სწრაფი მიმოცვლისათვის.

სიაში შემდეგი გვყავს AOT (Ahead Of Time Compilation). თუ ამის შესახებ უკვე არ გსმენიათ, მარტივად რომ ვთქვათ AOT საშუალებას გვაძლევს ჩვენი აპლიკაციები გავხადოთ უფრო სწრაფები და შევამციროთ მათი ზომა. ეს შესაძლებელი ხდება აპლიკაციის პირდაპირ ბაინერიზე დამუშავებით, სანამ იგი გაეშვება, რათა რანთაიმში დრო არ დაიხარჯოს low level კოდიდან ბაინარიზე თარგმანში.

საბოლოოდ ასევე აღსანიშნავია SDK შემცველი ბილდები, რომლებიც გვიმარტივებენ deployment ის ამბავს.

**Slide 4**

ვისაუბროთ .NET Aspire ის შესახებ. იგი შეიცავს საკმაო რაოდენობის გამოსადეგ ხელსაწყოებს და კომპონენტებს. რეალურად Aspire ხომ ინტეგრაციების გასამარტივებლად შეიქმნა. იგი გვაწვდის გამზადებულ თულებს ქლაუდ დეველოპმენტისთვის.

.NET Aspire ში ჩაშენებული ფუნქციონალები შეიცავს Service Discovery ს, რომელიც ეხმარება ჩვენს აპლიკცაიებს მარტივ კომუნიკაციაში, და Telemetry ს, რომელიც გვეხმარება აპლიკაციების მონიტორინგში. Aspire ში ასევე ყურადღება გამახვილებულია ამტანობაზე ?? და აპლიკაციის მდგრადობაზე რათა 24/7 სერვისი მივიღოთ.

აქვე ყურადსაღები საკითხია რომ Aspire ცალკე გამოშვებული ფუნქციონალი არაა. იგი .NET 8 ის ნაწილია და მასთან ერთად განვითარებას განაგრძობს.

**Slide 5**

მოდი დასაწყისისთვის .NET Aspire ის Starter შაბლონით შევქმნათ აპლიკაცია და თვალი გადვავლოთ თუ რა ხდება შიგ.   
შევქმნათ ახალი სოლუშენი vs 2022 preview ში, შაბლონად ავირჩიოთ .NET Aspire. შექმნილ სოლუშენში დავინახავთ რომ გვაქვს როგორც ჩვენთვის კარგად ნაცნობი WeatherForecast Api და Blazor ფრონტენდი, ასევე გვაქვს ორი ახალი პროექტი: ServiceDefaults და AppHost. რეალურად ეს AppHost იქნება რომელიც ნებისმიერ .NET აპლიკაციას გაუშვებს, ანუ ჩვენი სტარტერი პროექტიც ეგაა. მისი გაშვების შემდეგ დებაგერი ავტომატურად მიებმება პროცესებს VS ში ხოლო თუ Rider ზე ვართ მაშ ხელით მოგვიწევს ჯერ ჯერობით.

ServiceDefaults პროექტი შეიცავს საერთო გაზიარებულ ლოგიკას, რომელიც ეხება ყველა Aspire ში მოქცეულ პროექტს. აქვეა კონფიგურირებული Health Check ები Service Discovery, Telemetry და სხვა საჭირო გაზიარებული ფუნქციონალი.

მოდი ახლა გავუშვათ ჩვენი AppHost და ვნახოთ რა აპლიკაციები დაისტარტება. როგორც მოველოდით, გაეშვა როგოც API ასევე Blazor ის პროექტი და მათთან ერთად Developer Dashboard. დეშბორდი არის ჩვენი საწყისი წერტილი პრობლემების კვლევისა და მონიტორინგისთვის ლოკალზე. დეველოპმენტისას. აქ ჩანს თუ რა აპლიკაციები გვაქვს გაშვებული, თუ როგორი კომუნიკაცია აქვთ მათ ერთმანეთთან და არის თუ არა რამე პრობლემა მათში. აქვე უნდა გავითვალისწინოთ რომ Dev Dashboard გათვლილია მხოლოდ დეველოპმენტზე, ანუ ეს დეშბორდი ვერ გვექნება სატესტოზე ან პროდზე. ამ თემას კიდევ მივუბრუნდებით.

მოდი დავკვირდეთ ჩვენთვის ნაცნობ პროექტებში თუა რამე ცვლილება შეტანილი. ფრონტის პროექტს თუ შევხედავთ აღმოვაჩენთ რომ გვაქვს პაკეტი რომლის სახელიც Aspire ით იწყება. ამ ნაგეტ პაკეტს ეწოდება Aspire Component. კომპონენტები ერთგვარი შემაერთებელი ბიბლიოთეკებია, რომლებიც აკონფიგურირებენ SDK ს ქლაუდზე გასაშვებად. უფრო ადამიანურ ენაზე რომ ვთქვათ Component ები წინასწარ დეფოლტად გაწიკწიკებული კონფიგურაციებით მოდიან ნებისმიერ ინფრასტრუქტურაზე რომელსაც დააჯდებიან და cloud ready არიან. იმისათვის რომ პაკეტი ჩაითვალოს კომპონენტად მან უნდა:

1. მოგვაწოდოს json სქემა appsetting თვის
2. უნდა დააკონფიგურიროს (თუ ამის საშუალებაა) retry, timeout და circuit breaker ები ხელმისაწვდომობის გასაზრდელად
3. უნდა გამოდგას ჰელს ჩეკები
4. უნდა შემოგვთავაზოს ჩაშენებული ლოგირება, მეტრიკები და ტრეისინგის საშუალებები .NET ის თანამედროვე ფუნქციონალის გამოყენებით (ILogger, Metric, Activity)
5. შემოგვთავაზოს ექსტენშენ მეთოდი DI ში ამ ყველაფრის ერთად შესაწებებლად

კოდშიც ჩავიჭყიტოთ ცოტა. მოდი დავაკვირდეთ

**Slide 6**

მოდი გავეცნოთ Aspire კომპონენტებს. კომპონენტები Lego სავითაა Aspire ში ჩამატებული და შეგვიძლია ნებისმიერი საჭირო კომპონენტი მარტივად დავამატოთ ჩვენს აპლიკაციაში.

მაგალითად Aspire ის ერთ-ერთი კომპონენტია Aspire.StackExchane.Redis რედისის ქეში. ჩვენ შეგვიძლია დავამატოთ საჭირო ბიბლიოთეკები Aspire პროექტში და თვალის დახამხამებაში გვექნება გამართული Redis. შეგვიძლია დავამატოთ როგორც უბრალოდ ბიბლიოთეკა არსებულ რედისის სერვერთან სამუშაოდ, ასევე კონტეინერი რომელიც სრულიად ახალ კონტეინერს გადმოწერს და გაუშვებს აპლიკაციასთან ერთად (საჭიროა გვეყენოს დოკერი).

We’re going to start our deep dive with components. .NET Aspire Components are designed to solve the pain that we heard from customers getting started with cloud-native development, that there was a lot of techniques/configuration you had to get right and that it wasn’t obvious what path to start with. We help this by being *opinionated* about what a component needs to provide, mandating that all components at a minimum provide resiliency defaults, health checks, setup telemetry, and integrate with DI. To highlight that, let’s look at what an app ready for production might do to configure Redis in their app:

1. Add the Redis package with the Redis client library.
2. Discover and add a health checks library so your app can respond to the Redis being unavailable. This is frequently missed but useful in practice.
3. Add Redis to DI and configure connection strings. This is tricky because you need to know what lifetime the Redis client library types should have. Which requires research.
4. Configure Redis client library to send log output to your telemetry system.
5. Logs and Metrics are different and require different plumbing.
6. Decide what resiliency policy and logic is needed and configure Redis or wrap calls with a library like Poly that can implement resiliency policies. This again requires research into the capabilities of Redis and knowledge of what resiliency policy you should have, which is frequently not something you know starting out and results in people shipping without it until something breaks in production that could’ve been avoided with a retry policy with exponential backoff.

If we contrast that with using .NET Aspire:

1. Add the .NET Aspire Redis package.
2. Call AddRedis on the builder.
3. Optionally override default config in *appSettings.json* (which is now schematized so you have completion to discover what can be set).

The .NET Aspire Components are crafted to provide you with an optimal production-ready configuration without concealing the underlying SDK. In both examples mentioned, your code for utilizing Redis will consistently employ the same Redis client library and types.

A component must do the following to be considered ready for use:

* Provide detailed, schematized, configuration.
* Setup Health Checks to track and respond to the remote services health.
* Provide a default, configurable, resiliency pattern (retries, timeouts, etc) to maximize availability.
* Offer integrated logging, metrics, and tracing to make the component observable.

Our initial set of components are listed below, and more documentation can be found at [.NET Aspire components overview | Microsoft Learn](https://learn.microsoft.com/dotnet/aspire/components-overview).

**Slide 7**

AppHost პროექტი .NET Aspire ში საშუალებას გვაძლევს ავსახოთ ჩვენი აპლიკაციის საჭიროებები ჩვენთვის საყვარელ ენაზე C# ზე. ასევე AppHost პროექტია პასუხისმგებელი ჩვენი აპლიკაციების ლოკალურ ორკესტრაციაზე. Aspire orchestration ორ ძირითად პრობლემას გვიგვარებს:

App Composition : მასში ჩვენ აღვწერთ თუ რისგან შედგება ჩვენი აპლიკაცია. (.NET პროექტები, exe ფაილები, კონტეინერები და ა.შ.)

Service Discovery: AppHost ში დგინდება თუ როგორ უკავშირდებიან სხვადასხვა რესურსები ერთმანეთს.

მოდი program.cs ს შევხედოთ და მის მიხედვით ვიმსჯელოთ.