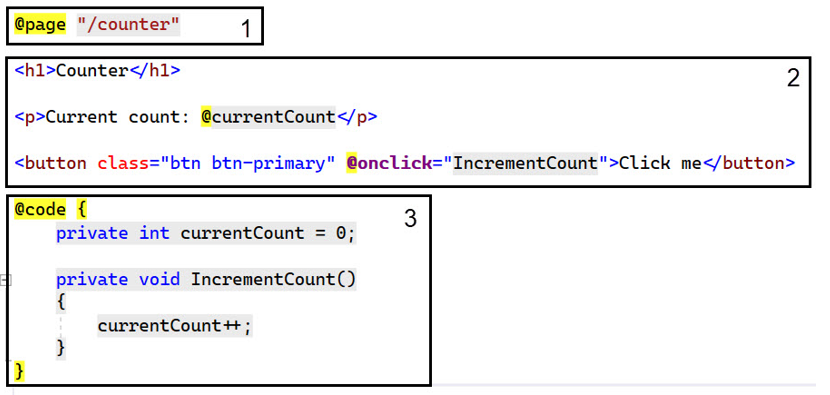
**Blazor nedir?**  
Blazor, Single Page Application geliştirme çevçevesidir.  
Blazor açık kaynaktır.  
  
**Neden Blazoru Seçmelisin?**  
1.Modern ve zengin özelliklere sahip bir dil olan C#:  
Güçlü, öğrenmesi kolay ve çok yönlüdür. C# nesne yönelimli bir dil olsa da, isterseniz daha işlevsel bir yaklaşım sağlamak için giderek daha fazla yetenek benimsiyor. Statik yazma, geliştiricilerin derleme sırasında hataları yakalamasına yardımcı olarak geliştirme yaşam döngüsünü daha hızlı ve daha verimli hale getirir. Ayrıca uzun süredir piyasada, şu anda 10. versiyonunda . Kararlı, iyi tasarlanmış ve iyi destekleniyor.  
  
**2.Harika araçlar:**  
.NET topluluğu bazı harika araçlara sahip olduğu için şanslı. Visual Studio, son derece güçlü, zengin özelliklere sahip ve genişletilebilir bir IDE'dir - ayrıca bireyler, açık kaynaklı çalışma veya 5 kişiye kadar kurumsal olmayan ekipler için %100 ücretsizdir. Daha hafif bir şey tercih ederseniz, günümüzün en popüler kod düzenleyicilerinden biri olan Visual Studio Code (VS Code) var. Hem Visual Studio hem de VS Code platformlar arası kullanılabilir. Visual Studio, Windows ve Mac'te, VS Code ise Windows, Mac ve Linux'ta mevcuttur. AyrıcaJetBrains'in Rider adında harika bir 3. parti IDE'si var. Bu aynı zamanda Windows, Mac ve Linux üzerinde çalışan tamamen çapraz platformdur.  
  
**3. .NET Ekosistemi :**  
 Pek çok yeni çerçevenin etraflarında bir ekosistemin oluşmasını beklemesi gerekirken, Blazor mevcut .NET ekosisteminden faydalanabilir. Yazma sırasında, Blazor uygulamaları .NET 6'yı hedefler ve teoride herhangi bir uyumlu NuGet paketini kullanabilir. Teoride, bazı paketlerin dosya sistemini değiştirmek gibi WebAssembly senaryosunda izin verilmeyen eylemleri gerçekleştirdiğini söylüyorum  
  
**4.Yorumsuz:**  
 Diğer çerçeveler uygulamaların ne kadar yazılacağını şart koşarken, Blazor bunu yapmaz. Blazor geliştirme için tercih edilen kalıplar veya uygulamalar yoktur, aşina olduğunuz ve rahat olduğunuz uygulamaları kullanarak uygulamalar yazabilirsiniz. MVVM'yi seviyorsanız, bunun için gidin. Redux kullanmayı tercih ediyorsanız, sahip olun. Seçim senin.  
   
 **5.Sığ öğrenme eğrisi :**  
 Mevcut bir .NET geliştiricisiyseniz, Blazor için öğrenme eğrisi oldukça sığdır. Razor, C#, bağımlılık enjeksiyonu, proje yapısı size tanıdık gelecek ve Blazor kalıplar hakkında fikir sahibi olmadığı için, sadece aşina olduğunuz ve üretken olduğunuz şeyleri kullanabilirsiniz. Tüm bunlar, çerçeveyi öğrenmek yerine özellikleri daha hızlı yazmaya odaklanabileceğiniz anlamına gelir.  
   
 **6.Kod paylaşımı :**  
 Sunucuda C# kullanıyorsanız Blazor mükemmel bir eşleştirme yapar. Farklı istemci ve sunucu dilleriyle ilgili en sinir bozucu sorunlardan biri, kodun yeniden kullanılamamasıdır. Modeller veya DTO'lar, sunucu ve istemci arasında çoğaltılmalıdır, senkronize olarak güncel tutulmaları gerekir. Bu, manuel bir süreç veya bir tür kod oluşturma kullanılarak otomatikleştirilmiş olabilir, ancak bu, kurulumu ve bakımı başka bir şeydir. Blazor ile her şey C#'tır. Paylaşılan herhangi bir kod, ortak bir .NET Standard sınıf kitaplığına yerleştirilebilir ve sunucu ile istemci arasında kolayca paylaşılabilir.  
   
 **7.Açık kaynak :**

Microsoft'taki birçok projede olduğu gibi, Blazor tamamen açık kaynaklıdır ve kod, kendi kopyanıza göz atmanız, indirmeniz veya çatallamanız için GitHub'da ücretsiz olarak bulunur. Açıktaki ekip çalışması, geliştirici istekleri ve geri bildirimleri tarafından yönlendirilir. Hatta isterseniz katkıda bulunabilirsiniz  
 **Bileşenler, kullanıcı arayüzü oluşturmanın daha iyi bir yolu:**  
 Blazor, birçok modern ön uç çerçevesinde olduğu gibi, kullanıcı arabirimini oluşturmak için bileşen kavramını kullanır. Her şey bir bileşendir; sayfalar, sayfanın bölümleri, düzenler—hepsi birer bileşendir. Blazor'da çeşitli bileşen türleri ve bunları tasarlamanın birden çok yolu vardır ve bunların tümü ileriki bölümlerde incelenecektir. Ancak bileşenler açısından düşünmeyi öğrenmek, Blazor uygulamaları yazmak için çok önemlidir.  
  **Bileşen nedir?**  
 Bir bileşeni yapı taşı olarak düşünebilirsiniz. Uygulamanızı oluşturmak için bu yapı taşlarını bir araya getiriyorsunuz. Bu yapı taşları, karar verdiğiniz kadar büyük veya küçük olabilir, ancak tüm bir kullanıcı arayüzünü tek bir bileşen olarak oluşturmak iyi bir fikir olmaz. Bileşenler, bir kullanıcı arayüzünün mantıksal alanlarını bölmenin bir yolu olarak düşündüğünüzde gerçekten faydalarını gösterirler. Bileşenler olarak yapılandırılmış bir kullanıcı arayüzü örneğine bakalım.  
 Arayüzün her alanı bir bileşendir ve her birinin belirli bir sorumluluğu vardır. Ayrıca bir hiyerarşi oluşumu olduğunu da fark edebilirsiniz. Düzen bileşeni, ağacın en üstünde yer alır; menü, üst bilgi, ana sayfa ve alt bilgi, düzen bileşeninin tüm alt bileşenleridir. Bu alt bileşenler, kendilerine ait alt bileşenlere sahip olabilir ve muhtemelen olacaktır. Örneğin, başlık bileşeni bir logo bileşeni ve bir arama bileşeni içerebilir  
   
 **Bileşen tabanlı bir kullanıcı arayüzünün faydaları:**  
 Birçok kullanıcı arabiriminde yinelenen öğeler bulunur, bileşenleri kullanmanın büyük bir avantajı, bir bileşende bir öğe tanımlayabilmeniz ve ardından bileşeni, öğenin tekrar ettiği her yerde yeniden kullanabilmenizdir. Bu, bir uygulamada tekrarlanan kod miktarını büyük ölçüde azaltabilir. Ayrıca, uygulamanın sürdürülebilirliğini çok daha iyi hale getirir, sanki o öğenin tasarımı değişiyormuş gibi, yalnızca tek bir yerde güncellemeniz gerekiyor.  
   
 Daha gelişmiş senaryoları karşılamak için bileşenler, veri ve olayların içeri ve dışarı iletilmesine izin veren kendi API'lerini tanımlayabilir. Bir iş kolu uygulaması düşünün, bu uygulama içinde çok sayıda yerin verisinin tablo formatında görüntüleneceğini varsaymak muhtemelen güvenlidir. Bir yaklaşım, her tabloyu kendi bileşeni olarak oluşturmak olabilir; ancak bu, verileri bir tabloda görüntüleyen birçok bileşenle sonuçlanacağı anlamına gelir. Bir veri kümesini parametre olarak alan tek bir bileşeni tanımlamak daha iyi bir yaklaşım olacaktır.ve sonra bir tabloda gösterdi. Artık tüm uygulamada yeniden kullanabileceğimiz bir tabloda verileri görüntülemek için tek bir bileşenimiz var. Bu bileşene sıralama veya sayfalama gibi özellikler de ekleyebiliriz. Yaptığımız gibi, bu işlevsellik, aynı bileşeni yeniden kullandıkları için uygulamadaki tüm tablolar için otomatik olarak kullanılabilir.  
   
 Genellikle bağımsız olsa da, daha karmaşık bir kullanıcı arayüzü oluşturmak için bileşenlerin birlikte çalışması mümkündür. Örneğin, az önce bahsettiğimiz veri tablosu senaryosunu ele alalım, bu tek bir bileşen olabilir, ancak potansiyel olarak oldukça büyük olabilir. Başka bir yaklaşım, onu her biri belirli bir işi yapan birkaç küçük bileşene bölmek olacaktır. Bir tablo başlığı bileşenine, bir tablo gövdesi bileşenine, hatta bir tablo hücresi bileşenine sahip olabiliriz. Bu bileşenlerin her biri belirli bir işi gerçekleştirir ancak yine de genel tablo bileşeninin bir parçasıdır.  
   
   
 **Blazor bileşenini anatomisi:**



Kodu anlamak bu noktada önemli olmasa da, bileşenin yapısını anlayabiliriz. Şekil 1.3, her birinin belirli bir sorumluluğu olan üç bölüme ayrılmıştır.

* **Bölüm 1** , yönergeleri tanımlamak, kullanım ifadeleri eklemek, bağımlılıkları enjekte etmek veya tüm bileşen için geçerli olan diğer genel konfigürasyonları tanımlamak için kullanılır.
* **Bölüm 2** , bileşenin işaretlemesini tanımlar; bu, C# ve HTML'nin bir karışımı olan Razor dili kullanılarak yazılmıştır. Burada bileşeni oluşturan görsel öğeleri tanımlıyoruz.
* **Bölüm 3** kod bloğudur. Bu, bileşenin mantığını tanımlamak için kullanılır. Bu bölüme herhangi bir geçerli C# kodu yazılabilir. İsterseniz alanları, özellikleri, hatta tüm sınıfları tanımlayabilirsiniz.

**Blazor, C# ile modern kullanıcı arayüzü oluşturmaya yönelik bir platform**

Blazor, C# ve .NET'in gücünü kullanarak modern istemci tarafı uygulamaları oluşturmaya yönelik tam özellikli bir çerçevedir. Geliştiricilerin web, mobil ve masaüstü dahil olmak üzere neredeyse tüm platformlarda çalışan ilgi çekici uygulamalar oluşturmasına olanak tanır.

Blazor, Angular, Vue ve React gibi JavaScript çerçevelerine ve kitaplıklarına bir alternatiftir. Bunlardan herhangi biriyle çalışma deneyiminiz varsa, muhtemelen tanıdık kavramları tespit etmeye başlayacaksınız. En dikkate değer etki, bileşenlerle UI'ler oluşturma fikri, tüm bu teknolojilerin paylaştığı bir kavram ve bu bölümün ilerleyen kısımlarında daha ayrıntılı olarak keşfedeceğimiz bir şeydir.

Blazor web standartlarının üzerine kurulduğu için; son kullanıcının makinelerinde .NET veya herhangi bir tarayıcı eklentisi veya uzantısı yüklü olmasını gerektirmez. Aslında Blazor WebAssembly uygulamaları ile sunucu üzerinde çalışan .NET'e bile ihtiyacımız yok, Blazor'un bu lezzeti basit statik dosyalar olarak barındırılabilir.

.NET üzerine kurulu olmak, NuGet'te bulunan canlı paket ekosistemine erişimimiz olduğu anlamına gelir. Visual Studio, Visual Studio Code ve JetBrains Rider ile sınıfının en iyisi araçlara sahibiz. Ayrıca, .NET çapraz platform olduğundan, Blazor uygulamalarımızı Windows, Mac veya Linux olsun, tercih ettiğimiz platform ne olursa olsun geliştirebiliriz.

Bu kitap web uygulaması geliştirme için Blazor'a odaklanacak olsa da, Blazor'un programlama modelinin platformlar arası masaüstü uygulamaları oluşturmak için de kullanılabileceğini vurgulamak istiyorum. .NET 6 ile Blazor Hybrid tanıtıldı. Yeni .NET Multi-platform App UI çerçevesi (diğer adıyla MAUI) üzerine inşa edilmiş olup, Electron uygulamalarına benzer şekilde çalışır. Bir Blazor uygulamasından gelen içerik, bir BlazorWebView denetimi aracılığıyla işlenir. Bu, bu uygulamaların nasıl yapılandırıldığı konusunda birçok seçenek sunar. Geliştiriciler, başlık çubuğunu içeren uygulama için en dıştaki kapsayıcı olan krom hariç, tüm kullanıcı arayüzünü oluşturmak için Blazor ve web teknolojilerini kullanmayı seçebilir. Veya Blazor ile yazılacak arayüzün yalnızca belirli bir parçasını hedefleyebilir ve yerel kontrollerin yanında barındırabilirler.

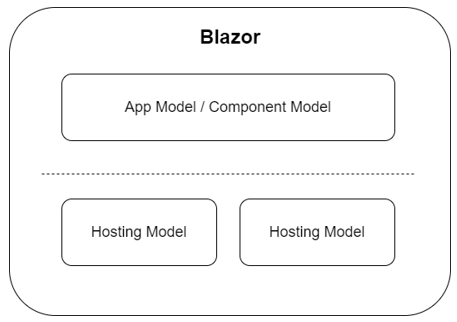
Orada bitmiyor. [Mobile Blazor Bindings ( https://docs.microsoft.com/en-us/mobile-blazor-bindings/](https://docs.microsoft.com/en-us/mobile-blazor-bindings/) ) adında uzun süredir devam eden bir deneysel proje var . Bu, yerel mobil uygulamalar oluşturmak için Blazor'un programlama modelini kullanma potansiyelini ve talebini araştırmak için ASP.NET Core ekibi ile .NET MAUI ekibi arasında bir işbirliğidir!

Bu, Blazor'u bir kez anlaşıldığı gibi öğrenmesi gerçekten zorlayıcı bir teknoloji haline getiriyor ve geliştiricilerin hemen hemen her platform veya cihaz için UI'ler oluşturmasına izin verebilir.

**1.3.1 Barındırma modellerini anlama**

Blazor'u kullanmaya başladığınızda hemen *hosting modelleri* kavramıyla karşılaşacaksınız . Esasen, barındırma modelleri bir Blazor uygulamasının çalıştırıldığı yerdir. *Şu anda Blazor, Blazor WebAssembly* ve *Blazor Server* adlı web'e özel iki barındırma modeline sahiptir . Uygulamanız için bu modellerden hangisini seçerseniz seçin, bileşen modeli aynıdır, yani bileşenler aynı şekilde yazılır ve her iki barındırma modeli arasında değiştirilebilir (şekil 1.4).

**Şekil 1.4 Blazor, barındırma modelleri ile uygulama/bileşen modeli arasında bir ayrım yapar. Yani bir hosting modeli için yazılan bileşenler bir diğeriyle birlikte kullanılabilir.**

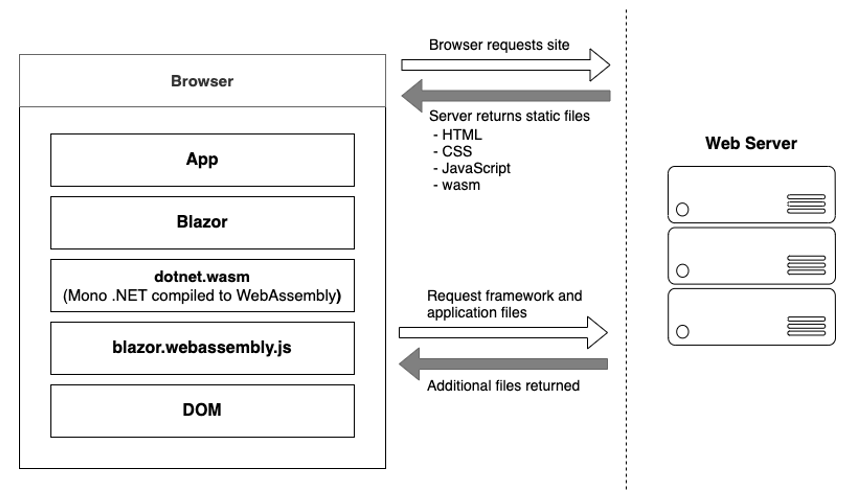


Şekil 1.4, uygulama ve bileşen modeli ile çeşitli barındırma modelleri arasındaki ayrım ile Blazor mimarisinin soyut bir temsilini göstermektedir. Blazor'un ilginç yönlerinden biri, zaman içinde kullanıma sunulan diğer barındırma modellerinin potansiyelidir. Bu, Blazor'un daha fazla yerde çalışmasına ve daha fazla UI türü oluşturmak için kullanılmasına olanak tanır.

**Blazor WebAssembly**

Blazor WebAssembly'yi seçmek, uygulamanızın tamamen müşterinin tarayıcısında çalışacağı ve onu JavaScript SPA (Tek Sayfa Uygulaması) çerçevelerine doğrudan bir alternatif haline getireceği anlamına gelir. Bu barındırma modelinin nasıl çalıştığını anlamak için, şekil 1.5'te gösterilen bir Blazor WebAssembly uygulamasını başlatma sürecini gözden geçireceğiz.

**Şekil 1.5 İstemcinin tarayıcısı ile web sunucusu arasındaki etkileşimleri gösteren bir Blazor WebAssembly uygulamasının önyüklemesi.**



İşlem, tarayıcı tarafından web sunucusuna bir istek yapıldığında başlar. Web sunucusu, uygulamayı yüklemek için gereken bir dizi dosyayı döndürür; bunlar, genellikle index.html olarak adlandırılan uygulamanın ana bilgisayar sayfasını, görüntüler, CSS ve JavaScript gibi uygulamanın gerektirdiği tüm statik varlıkları içerir. *blazor.webassembly.js* adlı özel bir JavaScript dosyasının yanı sıra .

Blazor WebAssembly barındırma modelinde, Blazor çerçevesinin bir kısmı JavaScript'te bulunur ve *blazor.webassembly.js* dosyasında bulunur. Çerçevenin bu kısmı üç ana şey yapar:

1. Tarayıcıda Blazor uygulamasını yükler ve başlatır.
2. Blazor'un UI güncellemelerini gerçekleştirebilmesi için doğrudan DOM manipülasyonu sağlar.
3. Daha sonraki bölümlerde ayrıntılı olarak tartışacağımız JavaScript birlikte çalışma senaryoları için API'ler sağlar.

Bu noktada, neden bir JavaScript dosyamız olduğunu merak ediyor olabilirsiniz, Blazor'un en büyük satış noktalarından biri, JavaScript yerine C# kullanarak UI mantığı yazabilme yeteneğidir, değil mi? Evet bu doğru. Ancak şu an itibariyle WebAssembly'nin büyük bir sınırlaması var, DOM'yi değiştiremez veya Web API'lerini doğrudan çağıramaz. Bu özellikler, WebAssembly'nin sonraki aşaması için planlanmış ve üzerinde çalışılmaktadır, ancak JavaScript'i indirene kadar bu görevleri gerçekleştirmenin tek yolu budur.

Gelecekte bu dosyanın artık gerekli olmaması mümkündür, bu, özelliklerin WebAssembly'ye ne kadar hızlı eklendiğine ve tarayıcılar tarafından ne kadar hızlı benimsendiğine bağlı olacaktır. Ama şimdilik, çerçevenin önemli bir parçası.

Şimdi bunu temizledik, hadi açılış Blazor uygulamamıza geri dönelim. Sunucudan dönen dosyaların hepsinin statik dosyalar olduğunu, herhangi bir sunucu tarafı derleme veya manipülasyon gerektirmediğini belirtmek isterim. Bu, statik barındırma sunan herhangi bir hizmette barındırılabileceği anlamına gelir, sunucuda bir .NET çalışma zamanının bulunmasına gerek yoktur. İlk defa bu, GitHub sayfaları gibi ücretsiz barındırma seçeneklerini .NET geliştiricilerine açar (yalnızca bağımsız Blazor WebAssembly uygulamaları için geçerlidir).

Tarayıcı web sunucusundan tüm ilk dosyaları aldığında, bunları işleyebilir ve Belge Nesne Modelini (DOM) oluşturabilir. Ardından, *blazor.webassembly.js yürütülür, bu birçok eylemi gerçekleştirir, ancak bir Blazor WebAssembly uygulamasını başlatma bağlamında blazor.boot.json* adlı bir dosyayı indirir . Bu dosya, uygulamayı çalıştırmak için gereken tüm çerçeve ve uygulama dosyalarının bir envanterini içerir. İndirildikten sonra, uygulamayı çalıştırmak için gereken kalan dosyaları indirmek için kullanılır.

Bu dosyaların çoğu normal .NET derlemeleridir, özel bir şeyleri yoktur ve uyumlu herhangi bir .NET çalışma zamanında çalıştırılabilirler. Ancak indirilen *dotnet.wasm* adında başka bir dosya türü daha vardır . *dotnet.wasm* dosyası , WebAssembly için derlenmiş eksiksiz bir .NET çalışma zamanıdır.

**WEBASSEMBLY**

WebAssembly, yerel performansa yakın modern web tarayıcılarında çalıştırılabilen, düşük seviyeli, derleme benzeri bir dildir. WebAssembly'yi doğrudan yazmak mümkün olsa da, daha yaygın olarak C/C++ ve Rust gibi daha yüksek seviyeli diller için bir derleme hedefi olarak kullanılır. JavaScript ile birlikte çalışacak ve JavaScript'in WebAssembly'ye çağrı yapmasına izin verecek şekilde tasarlanmıştır ve bunun tersi de geçerlidir. WebAssembly, JavaScript uygulamalarıyla aynı güvenlik sanal alanında da çalışır.

WebAssembly hakkında ayrıntılı bilgi için [https://webassembly.org adresini](https://webassembly.org/) ziyaret edin.

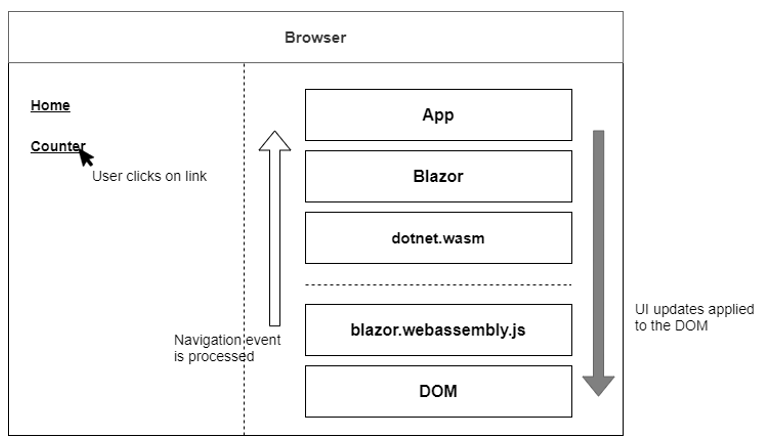
Varsayılan olarak, WebAssembly'ye yalnızca .NET çalışma zamanı derlenir; çerçeve ve uygulama dosyaları standart .NET derlemeleridir. Ancak, .NET 6'da geliştiricilerin uygulamalarını WebAssembly'de derlemelerine olanak tanıyan AOT modu tanıtıldı. Bunun yararı, CPU yoğun kod için çok daha iyi performanstır. AOT kullanarak, WebAssembly'de derlenen CPU yoğun kod, varsayılan olarak kullanılan yorumlanmış yaklaşımdan çok daha fazla performans gösterecektir. Ancak, bir ödünleşim var ve bu boyut. AOT derlenmiş kodu, uygulama için çok daha büyük bir genel indirme boyutu anlamına gelen standart derlemelerden yaklaşık 2 kat daha büyüktür.

Blazor.boot.json dosyası indirildikten ve içinde listelenen dosyalar indirildikten sonra, uygulamanın çalıştırılma zamanı gelmiştir *.*WebAssembly .NET çalışma zamanı başlatılır ve bu da Blazor çerçevesini ve son olarak uygulamanın kendisini yükler. Bu noktada, tamamen müşterinin tarayıcısında bulunan çalışan bir Blazor uygulamamız var. Ek veri istemenin (varsa) dışında, sunucuya başka bir güven yoktur.

**UI Güncellemelerini Hesaplama**

Artık bir Blazor WebAssembly uygulamasının nasıl başlatıldığını anlıyoruz. Ancak UI güncellemeleri nasıl hesaplanır? Başlatma işlemi için yaptığımız gibi, bunun nasıl olduğunu ve Blazor'un ne yaptığını anlamak için bir senaryo izleyeceğiz (şekil 1.6).

**Şekil 1.6 Blazor WebAssembly'de UI güncellemelerinin uygulanmasına giden bir bağlantıya tıklayarak istemci tarafında gezinme süreci.**



Senaryomuz için *Home* ve *Counter* olmak üzere iki sayfalı bir Blazor WebAssembly uygulamamız var . Bu sayfaların hiçbirinde sırasıyla “Ev” veya “Sayaç” yazan bir başlık dışında hiçbir şey yoktur. Kullanıcı, uygulamanın ana sayfasındadır ve Sayaç sayfasına gitmek için bağlantıya tıklayacaktır. Ana sayfadan Sayaç sayfasına giderken kullanıcı arayüzünü güncellemek için Blazor'un geçtiği süreci takip edeceğiz.

Kullanıcı sayaç bağlantısını tıkladığında, navigasyon olayı Blazor'un JavaScript çalışma zamanı ( *blazor.webassembly.js)* tarafından yakalanır . *Bu olay daha sonra WebAssembly çalışma zamanında ( dotnet.wasm* ) çalışan Blazor çerçevesine aktarılır ve Blazor'un yönlendirici bileşeni tarafından işlenir.

Yönlendirici, kullanıcının gitmeye çalıştığı bağlantıyla eşleşen herhangi bir yönlendirilebilir bileşen için yönlendirme tablosunu kontrol eder. Bizim durumumuzda, Counter bileşeni ile bir eşleşme bulacaktır ve bu bileşenin yeni bir örneği oluşturulacaktır; ve ilgili yaşam döngüsü yöntemleri yürütülecektir.

Tamamlandığında, Blazor, DOM'yi Counter bileşenininkiyle eşleşecek şekilde güncellemek için gereken minimum değişiklik sayısını hesaplayacaktır. Bu tamamlandığında, bu değişiklikler Blazor JavaScript çalışma zamanına geri aktarılacak ve bu da sırayla bu değişiklikleri fiziksel DOM'ye uygulayacaktır. Bu noktada UI güncellenecek ve kullanıcı Sayaç sayfasında olacaktır.

Tüm bunlar, kullanıcı tarayıcısında istemci tarafında oldu. Bu süreçte herhangi bir noktada sunucuya ihtiyaç duyulmadı. Gerçek dünyadaki bir uygulamada, muhtemelen bu süreçte bir sunucuya bir çağrı yapacağınızı söylemek doğru olur. Bu genellikle, bileşen için bazı başlangıç ​​verilerini yüklemek için gezinilen bileşenin yaşam döngüsü yöntemlerinin yürütülmesi sırasında olur. Ancak bu, bireysel uygulamaya bağlı olacaktır.

**Avantajlar ve ödünler**

Artık Blazor WebAssembly barındırma modelinin nasıl çalıştığı hakkında biraz daha fazla şey biliyoruz, bu modeli seçmenin yararları ve avantajları hakkında konuşalım. Faydalarıyla başlayalım.

* ***Uygulamalar istemcide çalışır.***Bu, sunucuda çok daha az yük olduğu anlamına gelir, işin çoğunu istemciye devredebilirsiniz. Bu, sunucu altyapısında önemli ölçüde maliyet tasarrufu sağlayabilir ve bir uygulamanın ölçeklenebilirliğini iyileştirebilir.
* ***Çevrimdışı senaryolarda çalışabilir.***Uygulama tamamen tarayıcının içinde çalıştığından, sunucuya kalıcı bir bağlantıya gerek yoktur; bu da uygulamaları kararsız ağ bağlantılarına karşı daha toleranslı hale getirir. Aşamalı Web Uygulaması (PWA) işlevini etkinleştirmek de önemsizdir. Aslında, Blazor WebAssembly, uygulamanızı oluştururken seçebileceğiniz bir seçenek olarak buna sahiptir.
* ***Statik dosyalar olarak dağıtılır.***Blazor WebAssembly uygulamaları yalnızca statik dosyalar olduğundan, statik barındırmanın olduğu her yere dağıtılabilirler. Bu, geçmişte .NET geliştiricileri için hiç mevcut olmayan bazı seçenekleri açar. GitHub sayfaları, Netlify, Azure Blob Storage, AWS S3 Buckets, Azure Statik Web Siteleri gibi hizmetlerin tümü, bağımsız Blazor WebAssembly uygulamalarını barındırmaya yönelik seçeneklerdir. Statik dosyaları dağıtmanın maliyeti, önde gelen Bulut sağlayıcılarının her birinde Web Uygulamalarını barındırmaya kıyasla nispeten daha düşüktür.
* ***Kod paylaşımı.***Blazor WebAssembly'nin potansiyel olarak en büyük avantajlarından biri, sunucuda C# kullanmanızdır. Artık sunucuda kullandığınız C# nesnelerinin aynısını istemcinizde de kullanabilirsiniz. TypeScript modellerini C# eşdeğerleriyle senkronize tutma ve bunun tersini yapma günleri sona erdi.

Tabii ki, hiçbir şey gümüş bir kurşun değildir, bu yüzden bu modelin bazı takaslarını anlayalım.

**Yük.** Bazı JavaScript uygulamalarıyla karşılaştırıldığında, Blazor uygulamalarının ilk indirme boyutu çok daha büyük olabilir, ancak bu her sürümde gelişiyor. Yayınlandığında yaklaşık 1 MB ağırlığında minimal bir Blazor uygulaması üretilebilir, ancak diğer uygulamalar önemli ölçüde daha büyük olabilir. Her uygulama farklıdır ve bir Blazor uygulaması için standart bir boyut yoktur. Ancak bu, çalışma zamanı ve çerçeve derlemelerinin çoğu ilk yüklemede önbelleğe alındığından tek seferlik bir maliyettir. Yani sonraki yükler birkaç kb kadar küçük olabilir.

**Yükleme zamanı.** Yük boyutunun zincirleme etkisi yükleme süresi olabilir. Kullanıcıların internet bağlantısı zayıfsa, ilk dosyaları indirmek için gereken süre daha uzun olacaktır, bu da uygulamanın başlamasını geciktirecek ve kullanıcıya bir tür yükleme mesajı bırakacaktır. Bu, sunucu tarafı önceden oluşturma kullanılarak biraz dengelenebilir, ancak bu, kullanıcıya başlangıçta bakmak için daha ilginç bir şey verirken, tüm dosyalar indirilip başlatılıncaya kadar uygulama yine de etkileşimli olmayacaktır. Blazor WebAssembly uygulamaları için sunucu tarafı önceden oluşturma, sunucuda herhangi bir ücretsiz barındırma seçeneğini reddeden bir ASP.NET Core öğesi de gerektirir.

**Kısıtlı çalışma zamanı.** Bu, tartışmasız bir değiş tokuş değil, ancak uygulamalarının çalıştığı makine üzerinde nispeten özgür bir dizgine sahip olmaya alışmış mevcut .NET geliştiricileri için, farkında olunması gereken bir şey. WebAssembly uygulamaları, JavaScript uygulamalarıyla aynı tarayıcı sanal alanında çalışır. Bu, örneğin, kullanıcıların makinesine erişmenize ve yerel dosya sistemine erişim gibi şeyler yapmanıza izin verilmeyeceği anlamına gelir.

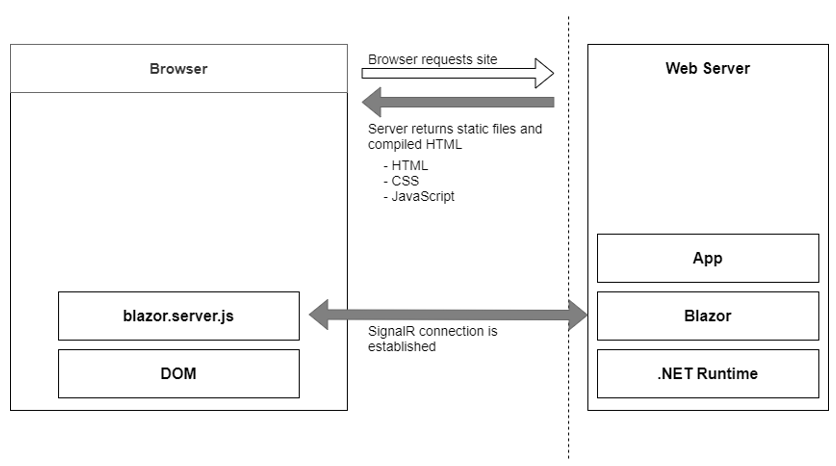
**Kod güvenliği.** JavaScript uygulamalarında olduğu gibi, kodunuz indirilir ve tarayıcıda çalıştırılır. Bu nedenle, kullanıcının uygulama DLL'lerinize erişimi vardır. Bu, bir Blazor WebAssembly uygulamasına fikri mülkiyet içeren herhangi bir kod eklememeniz gerektiği anlamına gelir. Herhangi bir değerli kod, bir API'nin parçası olarak sunucuda tutulmalıdır.

Özetlemek gerekirse, Blazor WebAssembly, Angular, React veya Vue gibi bir JavaScript SPA çerçevesi için doğrudan bir yedek arıyorsanız seçebileceğiniz barındırma modelidir. Dikkate alınması gereken birkaç ödün olsa da, bu modeli seçmenin bazı önemli faydaları vardır.

**Blazor Sunucusu**

Şimdi Blazor WebAssembly'nin nasıl çalıştığını gördük, şimdi dikkatimizi Sunucu barındırma modeline çevirelim ve nasıl farklı olduğunu görelim. Blazor Server, WebAssembly sürümünden yaklaşık 8 ay önce piyasaya sürülen Blazor için üretim destekli ilk barındırma modeliydi. Önceki modelde yaptığımız gibi, işlerin nasıl yürüdüğünü anlamamıza yardımcı olması için bir Blazor Server uygulamasını başlatma adımlarını inceleyeceğiz (şekil 1.7).

**Şekil 1.7 Blazor Server uygulamasının açılış süreci**



İşlem, siteyi tarayıcıdan yükleme isteğiyle başlar. Bu istek web sunucusuna ulaştığında iki şey olabilir, uygulama başlatılır veya uygulama zaten çalışıyorsa yeni bir oturum kurulur. Uygulama neden zaten çalışıyor olabilir? Geleneksel SPA modelini izleyen ve tamamen tarayıcıda çalışan Blazor WebAssembly ile karşılaştırıldığında, temelde onu bir masaüstü uygulaması gibi yapar. Her kullanıcının, kendi makinesinde yerel olarak çalışan kendi uygulama örneği vardır. Blazor Server farklıdır, uygulamanın yalnızca bir örneği sunucuda çalışır, ancak birçok istemciyi destekleyebilir. Bu nedenle, uygulama zaten çalışıyor olabilir ve yeni istek yalnızca yeni bir oturum oluşturur.

İstek daha sonra uygulama tarafından işlenir ve ilk yük tarayıcıya geri gönderilir. Buna CSS ve JavaScript dosyaları gibi statik varlıklar ve resimler dahildir. Başlangıç ​​HTML'si de vardır, ancak bu, Blazor WebAssembly'de gördüğümüz statik HTML yerine derlenmiştir. Bunun nedeni, bir Blazor Server uygulamasının barındırma sayfasının WebAssembly modelinde statik bir HTML sayfası yerine bir *Razor Sayfası olmasıdır.*Bunun avantajı, Blazor Server uygulamalarının kutudan çıktığı gibi sunucu tarafı önceden oluşturmayı kullanmasına izin vermesidir. Aslında, bu tür bir Blazor uygulaması oluşturduğunuzda bu özellik varsayılan olarak etkinleştirilir.

İlk yük tarayıcıya döndürüldüğünde, dosyalar işlenir ve DOM oluşturulur - ardından *blazor.server.js* adlı bir dosya yürütülür. Bu çalışma zamanının görevi, sunucuda çalışan Blazor uygulamasına bir SignalR bağlantısı kurmaktır. Bu noktada uygulama tamamen önyüklenir ve kullanıcı etkileşimi için hazırdır.

**SinalR**

SignalR, geliştiricilerin uygulamalarına gerçek zamanlı işlevsellik eklemelerine olanak tanıyan, Microsoft'un açık kaynaklı bir kitaplığıdır. İstemciler bir sunucuya bir hub aracılığıyla bağlanır, sunucu daha sonra WebSockets kullanarak istemcilere güncellemeleri gerçek zamanlı olarak gönderir (gerektiğinde diğer teknolojilere geri döner). SignalR kullanmanın yaygın bir örneği, gerçek zamanlı sohbet uygulamaları oluşturmaktır.

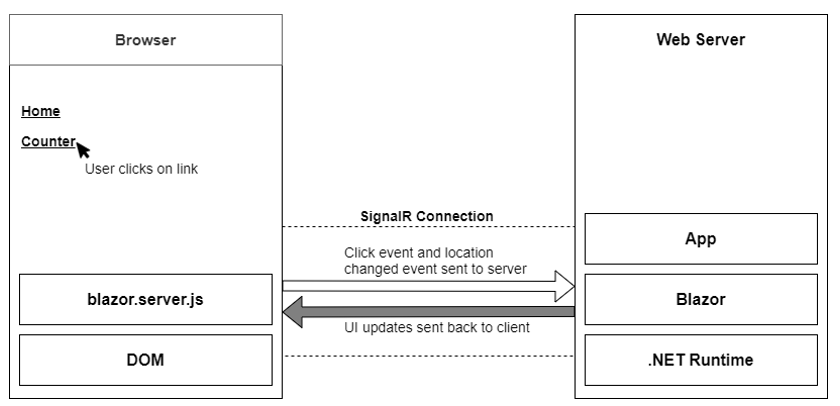
SignalR, Blazor'da olayları ve UI güncellemelerini istemci ve sunucu arasında ileri geri taşımak için kullanılırken, çerçevenin bir uygulama detayı olarak kabul edilir ve Blazor Server ile çalışan bir geliştiricinin yapılandırması veya etkileşim kurması gerekmeyecek bir şey olarak kabul edilir.

SignalR hakkında ayrıntılı bilgi için [https://dotnet.microsoft.com/apps/aspnet/signalr adresini](https://dotnet.microsoft.com/apps/aspnet/signalr) ziyaret edin.

**UI güncellemelerini hesaplama**

Bir kullanıcı uygulamayla etkileşim kurduğunda ne olur? Daha önce Blazor WebAssembly'de olayların, herhangi bir UI güncellemesinin hesaplanması ve DOM'a uygulanmasıyla birlikte tarayıcıda işlendiğini görmüştük. Ancak uygulama sunucuda çalıştığı için bu burada olamaz.

Blazor WebAssembly ile yaptığımız senaryonun aynısını izleyeceğiz, *ana* sayfa ve *sayaç* olmak üzere iki sayfalı bir Blazor Server uygulamamız var . Bu sayfaların hiçbirinde sırasıyla “Ev” veya “Sayaç” yazan bir başlık dışında hiçbir şey yoktur. Kullanıcı uygulamanın ana sayfasındadır ve sayaç sayfasına gitmek için bir bağlantıya tıklayacaktır. Ana sayfadan sayaç sayfasına giderken Blazor'un kullanıcı arayüzünü güncellemek için geçtiği süreci takip edeceğiz (şekil 1.8).



Kullanıcı menüdeki bağlantıya tıklar ve click olayı, Blazor'un istemci üzerindeki çalışma zamanı tarafından yakalanır. Çalışma zamanı daha sonra ne olduğunu anlamak için olayı işler. Bu durumda iki şey vardır, bir fare tıklaması olayı ve bir köprü olması nedeniyle bir gezinme olayı. Bu iki olay daha sonra paketlenir ve uygulama başladığında kurulan SignalR bağlantısı üzerinden sunucuya geri gönderilir.

Sunucuda, istemciden gönderilen mesaj paketinden çıkarılır ve işlenir. Blazor çerçevesi daha sonra gerekli herhangi bir uygulama kodunu çağırır. Bu durumda, sayaç sayfası bileşeninin bir örneğini başlatır ve ilgili yaşam döngüsü yöntemlerini yürütür.

Tamamlandığında, Blazor mevcut sayfanın sayaç sayfasına dönüştürülmesi için gereken minimum değişiklik sayısını hesaplayacak ve ardından bunları SignalR bağlantısı aracılığıyla istemciye geri gönderecektir. Sadece açık olmak gerekirse, Blazor istemciye tamamen yeni bir sayfa göndermeyecek. Yalnızca mevcut DOM'yi Sayaç sayfasıyla eşleşecek şekilde güncellemek için gereken minimum sayıda talimatı geri gönderir. Bizim durumumuzda, tek fark başlıktır. Blazor, başlıktaki metni “Ev”den “Sayaç”a değiştirmek için tek bir talimat gönderecektir. Başka hiçbir şey değişmeyecek.

İstemciye geri döndüğünde, değişiklikler paketten çıkarılır ve gerekli değişiklikler fiziksel DOM'ye uygulanır. Kullanıcının bakış açısından, uygulamada yeni bir sayfaya, sayaç sayfasına gitmiş gibi görünüyorlar. Ama hala aynı fiziksel sayfadalar, sadece farklı bir başlığı var.

Bunu zaten fark etmiş olabilirsiniz, ancak genel süreç Blazor WebAssembly'nin çalışma biçiminden farklı değil, sadece SignalR bağlantısının biraz üzerine uzatılmış durumda. Blazor Server, Angular, Vue veya Blazor WebAssembly kadar bir SPA'dır. Sadece mantığını çalıştırır ve UI güncellemelerini istemci yerine sunucuda hesaplar. Aslında, size biri Blazor Server'da ve diğeri Blazor WebAssembly'de yazılmış iki özdeş uygulama sunulsaydı, bir kullanıcı olarak aralarındaki farkı anlayamayacağınızı söyleyecek kadar ileri giderdim.

**Verim**

Bu modelin faydalarını ve ödünleşimlerini tartışmadan önce, performanstan hızlıca bahsetmek istiyorum. Bu barındırma modelinde devam eden tüm ağ sohbetleriyle, bunun özellikle iyi ölçeklenmeyebileceği aklınızdan geçmiş olabilir.

2019'da ASP.NET Core ekibi, Blazor Server uygulamalarının performans düzeylerini belirlemek için bazı testler yaptı. Azure'da bir uygulama kurarlar ve uygulamanın destekleyebileceği *aktif* kullanıcı sayısını kontrol ederek, onu farklı güçlendirilmiş sanal makinelerde test ederler. Sonuçlar burada.

* Standart D1 v2 Eşgörünümü (1vCPU ve 3.5GB Bellek). ***5000'den*** fazla eşzamanlı kullanıcı
* Standart D3 v2 Eşgörünümü (4vCPU ve 14 GB Bellek). 20.000'den ***fazla*** eşzamanlı kullanıcı

Gördüğünüz gibi, Blazor Server, performans söz konusu olduğunda hiç de gevşek değil. Desteklenebilecek istemci sayısını etkileyen ana faktör bellektir. Bu, sunucunun kendisine bağlı tüm istemcileri izlemesi gerektiğinden mantıklıdır - en çok istemci varsa, bellekte daha fazla bilgi depolanması gerekir.

Testten elde edilen diğer önemli bulgu, ağ gecikmesinin uygulamayı nasıl etkilediğiydi. Tüm etkileşimler işlenmek üzere sunucuya geri gönderildiğinden, gecikmenin kullanılabilirlik üzerinde büyük bir etkisi olabilir. Sunucu istemciden 250ms uzaktaysa, sunucuya gitmesi (250ms), ardından işlenmesi ve tekrar geri gitmesi (250ms) gerektiğinden, her etkileşimin işlenmesi en az 500 ms sürer.

Testler, gecikme süresi 200 ms'nin üzerine çıktığında, kullanıcı arayüzünün durgun ve daha az tepki vermeye başladığını buldu. Kaba bir kural olarak, kullanıcılarınızın her zaman sunucuyla aynı kıtada olmasını istersiniz. Küresel olarak kullanılabilen bir Blazor Sunucu uygulamasına sahip olmak istiyorsanız, tüm istemcileri bir sunucunun 200 ms'si içinde tutmayı hedefleyen uygulamanızın tüm dünyaya eşit bir şekilde dağıtılmasını sağlamanız gerekir.

**Avantajlar ve ödünleşimler**

Daha önce yaptığımız gibi, bir Blazor Server uygulaması seçmenin faydalarına ve avantajlarına bakalım.

* ***Küçük yük.***Uygulama istemcinin aksine sunucuda çalıştığından, ilk indirme önemli ölçüde daha küçüktür. CSS ve görüntüler gibi statik varlıklara bağlı olarak bir Blazor Server uygulaması 100-200 kb kadar küçük olabilir.
* ***Hızlı yükleme süresi.***Çok daha küçük bir yük ile uygulama çok daha hızlı yüklenir. Sunucu tarafı önceden oluşturma, kullanıcının hiçbir zaman bir yükleme mesajı görmemesine de yardımcı olur.
* ***Tam çalışma zamanına erişim.***Uygulama kodu, tam .NET çalışma zamanının üzerindeki sunucuda yürütülüyor. Bu, herhangi bir güvenlik kısıtlamasına çarpmadan gerekirse sunucu dosya sistemine erişim gibi şeyler yapabileceğiniz anlamına gelir.
* ***Kod güvenliği.***Tescilli kodunuz varsa ve insanların onu indirip sorgulamasını istemiyorsanız, Blazor Server iyi bir seçimdir. Uygulama kodunun tamamı sunucuda yürütülür ve istemciye yalnızca UI güncellemeleri gönderilir. Bu, kodunuzun hiçbir şekilde istemciye maruz kalmadığı anlamına gelir.

Blazor Server'ın bazı iyi faydaları var ama takaslar nasıl görünüyor?

* ***Ağır sunucu yükü.***Blazor WebAssembly, istemcinin gücünü kullanmamıza izin verdiğinde, Blazor Server tam tersini yapar. Artık işin neredeyse tamamı sunucu tarafından yapılıyor. Yani Blazor Server uygulamalarını desteklemek için altyapınıza daha büyük bir yatırım yapmanız gerekebilir. Uygulamanın boyutuna bağlı olarak, Blazor Sever tarafından kullanılan SignalR tabanlı oturumları doğru şekilde yönetmek için yük dengeleme de gerekebilir.
* ***Çevrimdışı çalışmıyor.***Blazor WebAssembly'nin adım adım çalışarak çevrimdışı olduğu yerde Blazor Server yapmaz. SignalR bağlantısı uygulamanın can damarıdır ve onsuz istemci hiçbir şekilde çalışamaz. Varsayılan olarak, bu, istemcinin bağlantıyı yeniden kurmaya çalıştığını söyleyen bir mesaj içeren bir bindirmeyle sonuçlanır. Bu başarısız olursa, kullanıcının uygulamayı yeniden başlatmak için tarayıcıyı yenilemesi gerekir.
* ***Gecikme.***Blazor Server uygulamaları, tasarımı nedeniyle gecikme sorunlarına karşı hassastır. Kullanıcının uygulama ile yaptığı her etkileşim, işlenmek üzere sunucuya geri gönderilmeli ve uygulanması gereken güncellemeleri beklemelidir. İstemci ve sunucu arasındaki bağlantıda yüksek bir gecikme varsa, kullanıcı arayüzünde fark edilir bir gecikme ortaya çıkar ve eylemler hızla yavaşlar. Gerçek sayılarda, 200 ms'nin üzerindeki bir gecikme bu sorunlara neden olmaya başlayacaktır.
* ***Kararlı bir bağlantı gerektirir.***Düşük gecikme ihtiyacının ardından ve çevrimdışı çalışamama ile bağlantı kurun. Blazor Server uygulamalarının sabit bir internet bağlantısına sahip olması gerekir. Bağlantı herhangi bir şekilde kesintiliyse, kullanıcı uygulamalarında sürekli olarak çok hızlı bir şekilde çok rahatsız edici hale gelen yeniden bağlanma bindirmesini görecektir. Bunun meydana gelebileceği bariz bir senaryo, bir kullanıcının kesintili bağlantıya sahip bir mobil cihazda olmasıdır.

Özetle, hızlı yüklenen bir uygulama arıyorsanız ve hızlı ve istikrarlı bir ağ bağlantısına sahip kullanıcılarınız varsa, Blazor Server harika bir seçimdir. Bu barındırma modelini seçerken ayrıca kod güvenliği de alıyorsunuz.

**Blazor Hibrit**

Blazor Hybrid, .NET MAUI çerçevesinden gelen teknoloji üzerine kuruludur ve geliştiricilerin Blazor kullanarak platformlar arası masaüstü uygulamaları yazmasına olanak tanır. Bileşenler, Blazor WebAssembly ve Blazor Server'da olduğu gibi C#, HTML ve CSS kullanılarak yazılır ve BlazorWebView adlı bir denetim kullanılarak oluşturulur.

**Mobil Blazor Bağları**

Mobile Blazor Bindings, deneysel bir barındırma modelidir ve bileşenlerin oluşturulmasına farklı bir yaklaşım getirir. Bu barındırma modelinin bileşenleri, yerel denetimler kullanılarak yazılmalıdır.

Liste 1.2, liste 1.1 ile aynı bileşeni içerir, ancak Mobile Blazor Bindings barındırma modeli için yeniden yazılmıştır.

<StackLayout>

<Label> Current count: @currentCount </Label>

<Button OnClick="@IncrementCount">Click me</Button>

</StackLayout>

@code {

private int currentCount = 0;

private void IncrementCount()

{

currentCount++;

}

}

Gördüğünüz gibi programlama modeli iki kod örneği arasında aynıdır. Kod bloğundaki mantık değişmedi, sonuçta sadece C#. Tek fark, yerel mobil kontroller için web teknolojilerinin değiştirildiği işaretlemededir. Bu, web tabanlı barındırma modelleri ile yerel barındırma modelleri arasında bileşen değiştiremeyeceğimiz anlamına gelir. Ancak, Blazor'un programlama modelinde uzmanlaştığımızda, bu bilgiyi başka türde kullanıcı arabirimi oluşturmak için kolayca kullanabiliriz.

Özet

* Blazor, geliştiricilerin JavaScript'e ihtiyaç duymadan zengin etkileşimli kullanıcı arabirimleri oluşturmak için C# ve .NET'in gücünü kullanmalarına olanak tanır
* Blazor, açık bir web standardı olan WebAssembly aracılığıyla tamamen tarayıcının içinde çalışabilen bir SPA çerçevesidir. Veya istemcinin tarayıcısını uygulamaya bağlamak için bir SignalR bağlantısı kullanan sunucuda.
* Blazor WebAssembly, Node veya Python gibi mevcut herhangi bir sunucu teknolojisiyle kullanılabilir. Ancak, kod bir .NET sınıf kitaplığı aracılığıyla kolayca paylaşılabildiğinden, bunu bir ASP.NET Core arka ucuyla kullanmanın gerçek faydaları vardır.
* Uygulamalar bileşenler kullanılarak yazılır. Bileşenler, tek başına veya birbirleriyle kombinasyon halinde çalışabilen bağımsız UI parçaları oluşturmamıza olanak tanır.