

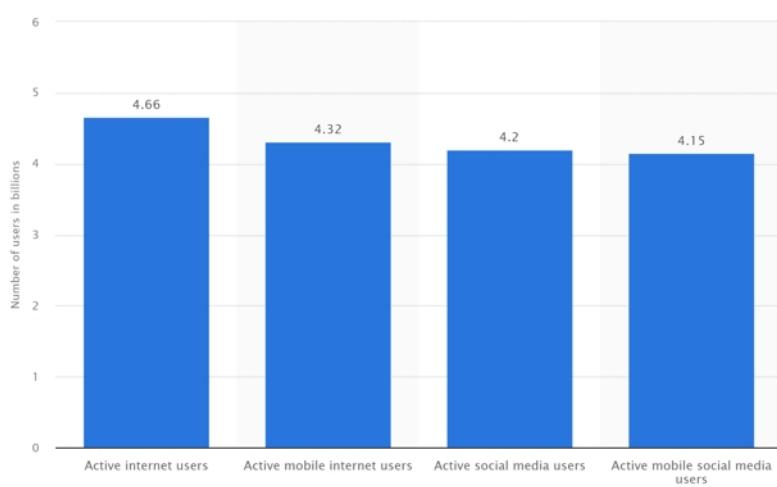
1. FLUTTER İLE MOBİL PROGRAMLAMAYA GİRİŞ

Giriş

Mobil platformlara kod geliştirmek bilgisayar ortamına ve İnternet üzerinden çalışacak web sitelerine yapılacak geliştirmelere benzese de kodun geliştirilen ortamda çalışmaması, mobil cihazların içerdiği farklı donanımsal özellikler, ekran boyutları, ekran çözünürlükleri ve batarya ömrünün iyi kullanımı gibi birçok faktör işin içeresine girmektedir. Derse başlamadan önce genel programlama yeteneklerinizin yeterli seviyede olması ve konulara çalışırken yukarıda sıralanan özelliklerin hatırlı tutulması önem arzettmektedir. Mobil platformlarda test yapılabilmesi için emülatörler kullanılmaktadır. Emülatörlerin sisteminizde rahat çalışması için bilgisayar performansının iyi olması ve sanallaştırma özelliğinin BIOS üzerinden açık olması gerekmektedir. Bu bölümde sizlere mobil programlama dünyası tanıtılacak ve Flutter ile mobil program geliştirmek için gerekli kurulumlar yaptırılacaktır.

1.1. Mobil Programlama Nedir ve Neden Gereklidir?

Teknolojinin ilerlemesiyle birlikte bilişim sektöründe kullanılan cihazlar ve kullanım amaçları da artmaya başladı. Cep telefonları bu artışın bir parçası olarak hayatımıza önceleri sadece ses ve SMS iletimi ile girmiş olsa da veri iletimi ve İnternet erişimi sayesinde farklı tür dosyaların aktarımı, işlenmesi ve depolanması için kullanır hale geldiler. Akıllı telefonların (smart phone) hayatımıza girmesi ile de artan algılayıcıları (sensor) ve her tür kişisel kullanım ihtiyacına cevap verecek işlemci hızı ve kapasiteye erişmeleri sayesinde artık hayatımızın vazgeçilmez bir parçası haline gelmiş durumdalar. Bu değişimi İnternete ve sosyal medyaya erişimde mobil platformların oranına bakarak görebiliriz. Aşağıdaki grafikte görüldüğü üzere Ocak 2021 verilerine göre aktif 4,66 milyar İnternet kullanıcısı ve 4,2 milyar sosyal medya kullanıcısı bulunmaktadır. Aynı doğrultuda mobil platformlardan İnternete erişim miktarı 4,32 milyar ve sosyal medya erişimi 4,15 milyar şeklinde gerçekleşmiştir. Bu bize açıkça İnternet erişiminin ve sosyal medya erişiminin özellikle son kullanıcılar seviyesinde düşünüldüğünde %95 üzerinde bir oranda mobil platformlardan gerçekleştiğini göstermektedir.



Mobil platformların geçmişine bakarsak günümüzde kullandığımız akıllı telefonlardan önce de uygulama yüklenebilen veya kendi hazır uygulamaları ile gelen mobil cihazlar mevcuttu. Bu eski donanımlarda özel hazırlanmış bir işletim sistemi bulunuyordu ve çoğunlukla C++’da gömülü olarak kod geliştirmek veya Java Midlet (Java ME) uygulaması geliştirmek gerekmektedir. Her iki yöntem de zor ve sadece tek bir cihaz için uygulama yazılmasına olanak tanıyan yöntemlerdi. Uygulama geliştiricisinin hem cihazın özelliklerinde bilgi sahibi olması hem de farklı modellerde yazılımını uyarlaması gerekmektedir. Günümüzde kullanılan akıllı telefonların önceki jenerasyonlara göre en büyük avantajı ise içerisinde barındırdıkları işletim sistemlerinin bilgisayarlarda kullanılan işletim sistemleri ile benzer çekirdeklerle sahip olmalarıdır. Bu sayede artık mobil cihazlara yazılım geliştirme ve geliştirilen yazılımın taşınabilir olma özelliği ciddi olarak artmıştır.

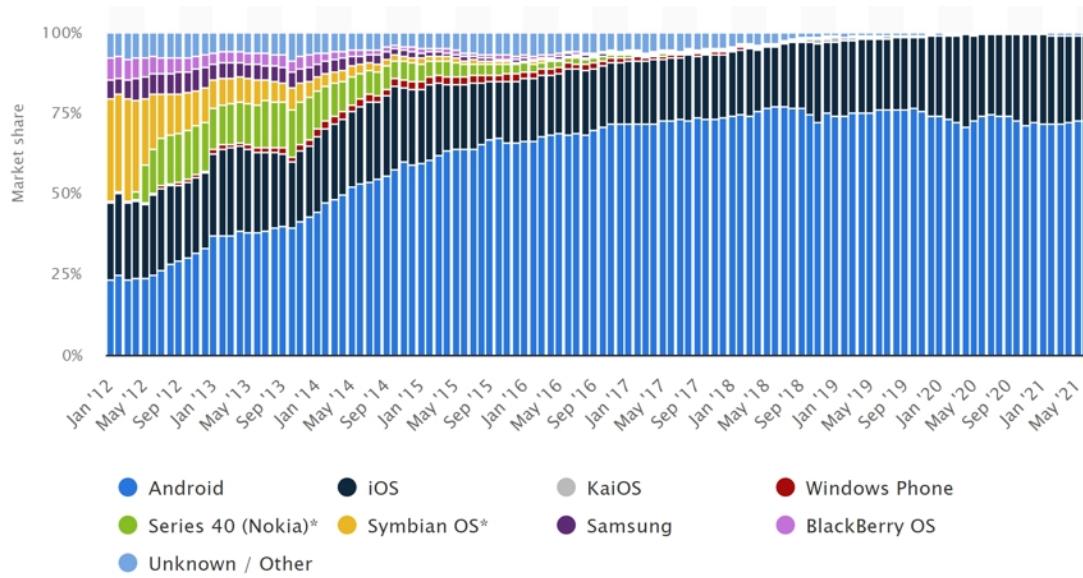
Kullandığımız akıllı telefonlar aslında bir masaüstü bilgisayardan farklı değildir. Yani içerisinde merkezi işlem birimi (CPU), hafiza (RAM), sabit hafiza (ROM), grafik işlemci ve genel amaçlı işletim sistemi bileşenlerini barındırmaktadır. Bilgisayarlarla mobil platformları karşılaştırıldığımızda aralarındaki asıl farkın çevre cihazları yani giriş-çıkış(I/O) cihazlarından kaynaklandığını rahatlıkla söyleyebiliriz. Bir bilgisayar, giriş cihazı olarak klavye, fare, tarayıcı ve mikrofon donanımları içermesine karşın bir akıllı telefon giriş cihazı olarak dokunmatik ekran, GPS sensoru, ivme sensoru, pusula, kamera, parmak izi okuyucu ve mikrofon gibi bileşenler içermektedir. Kullanıcı etkileşimi benzer şekilde farklılık gösterir. Bilgisayarlarda kullanıcı klavye ve fare kullanarak sisteme bilgi girişi yaparken, akıllı telefonlarda dokunmatik ekran ve genellikle yazılım tabanlı ekranda açılan sanal klavye ile etkileşim yapılmaktadır. Yine diğer algılayıcılarından gelen verilerin işlenmesi mobil platformlarda bilgisayarlardan farklı bir uygulama geliştirme sürecini beraberinde getirmektedir.

Mobil Programlama tanımı en yalın hali ile akıllı telefonlar (smart phones) veya daha geniş bir bakış açısı ile akıllı cihazlar (smart devices) için uygulama geliştirme sürecine verilen bir isimlendirmidir. Çıkış noktamız akıllı telefonlar olsa da geliştirilen uygulamalar akıllı telefonlarda kullanılan işletim sistemlerinin üzerinde çalıştığı televizyon, tablet hatta otomobil gibi farklı cihazlarda da kullanılabilimekte veya aynı platform üzerinden bu cihazlar için de uygulama geliştirilebilmektedir. Ders kapsamında anlatacağımız Flutter/Dart programlama platformu akıllı telefonların yanında giderek artan bir yelpazede farklı ortamlarda da (masaüstü, web, gömülü sistemler) çalışabilecek şekilde bir teknoloji kullanmaktadır. Amaç bir kere geliştirilen uygulamanın yeniden programlanması gerekmeden farklı platformlarda sorunsuz çalışabilmesidir. Bu açıdan Flutter sonraki bölümde açıklayacağımız alternatiflerinden bir adım öndedir.

1.2. Mobil Platformlar ve Aksesuarlar

Yüksek seviyeli dille yazılan uygulamalar donanımdan bağımsız olurlar. Yani yüksek seviyeli dilde yazılan bir uygulama Intel, AMD, ARM gibi farklı yongalar üzerinde aynı şekilde çalışabilmektedir. Ama ne yazık ki aynı şeyi işletim sistemleri için söylememiz mümkün değildir. Bir işletim sistemi için derlenen bir yazılım başka bir işletim sisteminde çalışmayaçaktır. Örneğin Windows platformunda yazdığımız bir uygulama Linux veya MacOS işletim sistemlerinde çalışmaz. Bunun birçok nedeni vardır. Buna girmeyeceğiz ama en basitinde arayüz yönetimi, dosya sistemi yönetimi farklılıklarını uygulamaları işletim sistemi bağımlı hale getirmektedir. Java ve .Net çatıları bu noktada bir esneklik getirmektedirler. Java için geliştirilen bir uygulama doğrudan bir işletim sisteminin algılayacağı makine kodlarına derlenmez. Bunun yerine Java Sanal Makinesi (Java Virtual Machine) adındaki “aslında kendisi de bir uygulama olan” bir platform için derlenir. Ve bu platformda Java kodları işlenerek işletim sisteminin algılayacağı makine kodları üretilir. Bu yaklaşım kodların taşınırlığını arttırsa da dosya sistemi gibi işletim sistemleri arasında uyum olmayan özelliklerden dolayı Linux altında çalışan bir Java uygulamasının Windows altında da çalışması için dosya işlemlerinde kodun yenilenmesi veya tüm işletim sistemleri düşünürlerek kodlanması gereklidir. Benzer durum mobil platformlarda da geçerlidir. Bu nedenle öncelikle piyasada aktif kullanımındaki mobil platformlar incelenerek geliştirilecek uygulamanın taşınabilir olması için gereksinimler belirlenmelidir.

Aşağıdaki grafikte mobil cihazlar için geliştirilmiş farklı işletim sistemlerinin 2012'de 2021 Haziran ayına kadar olan market dağılımları verilmektedir. Görüleceği üzere 2021 Haziran itibarı ile piyasadaki mobil cihazların %72,84 Android, %26,34 iOS ve sadece %0,82'lik kısmı diğer işletim sistemlerini kullanmaktadır. Anlaşılacağı üzere mobil cihazlar için geliştirilecek bir uygulamanın piyasadaki her bir cihazda çalışması isteniliyorsa hem Android hem de iOS işletim sistemleri için geliştirilmesi gerekmektedir.



Mobil platformlarda artık yer almayan işletim sistemleri için bir uyarlama yapmak veya onlara yönelik bir yazılım geliştirmek günümüz için gereksiz bir kaynak israfı olacaktır. Baktığımızda 2012 yılında oldukça popüler bir işletim sistemi olan Symbian OS açık kaynak kodlama yapısına sahip Android İşletim Sistemi'nin mobil cihaz üreticileri tarafından tercih edilmesi sonucu piyasadan tamamen silinmiş durumdadır. Neticeye baktığımızda mobil platformlarda karşımıza iki işletim sistemi çıkmaktadır. Bunlardan ilki Google firmasının desteklediği, Linux çekirdeği üzerine inşa edilen Android işletim sistemi ve ikincisi Apple firmasının MacOS'den uyarlayarak kendi iPhone ve iPad ürünleri için geliştirdiği iOS işletim sistemidir. iOS sadece Apple firmasının kendi ürünlerinde kullanıldığı için daha düşük bir pazar payına sahiptir. Diğer tarafta Android işletim sistemi farklı donanım üreticileri tarafından alınıp kendi ürünler için uyarlanarak kullanılabildiği için bu şekilde geniş bir market payına sahip olmuştur. Her ne kadar iOS'un market payı az görünse de tek bir firmanın ürün grubundaki kullanıldığı düşünülürse pazardaki başarısı yüksek denebilecek bir işletim sistemidir. Microsoft firması bu yarışta Windows8 ve Windows10 işletim sistemlerinin tasarımını mobil uyumlu yapsa da bu işletim sistemlerini baz alan Windows Phone platformu piyasada beklenen tepkiyi alamamıştır.

Mobil uygulama geliştirme süreci Android ve iOS üzere iki işletim sistemi hedef alınarak kurgulanabilir. Bu aşamada hangi programlama dilini kullanacağımız ve hangi uygulama çatılarını kullanacağımız bir uygulama geliştirmeden önce cevaplamamız gereken sorulardır. Sonraki başlıkta bu soruya detaylıca cevap arayacağız. Bu arada şunu da ekleyelim Huawei firması Google tekelinden kurtulmak için kendi işletim sistemi olan HarmonyOS'u piyasaya sürdü. Diğer taraftan Google cephesinde de Android sonrası için çalışmalar durmuş değil. Yeni işletim sistemi Google Fuchsia üzerindeki çalışmalarını 2016'dan beri sürdürmektedir. Bu işletim sistemlerinin belli bir Pazar payına ulaşması sonrası bunlar da mobil programlama dili seçiminde dikkate alınması gereken bir kriter olacaktır.

Mobil uygulama geliştirme süreci gün geçtikçe biraz daha karmaşıklaşmaya başlamıştır. Akıllı telefonlar ilk çıktığı zamanlarda sadece tek bir cihaz türü için yazılım geliştiriliyorken şimdi televizyon, tablet, telefon, otomobil ve giyilebilir teknolojiler (aklıllı saat, bileklik, gözlük vb.) yazılım geliştirme ihtiyacı doğmuştur. Mobil platformda çalışacak uygulamanın bu donanımlarda çalışması veya haberleşmesi gerekliyse bunun uygulama geliştirme süreçlerinde düşünülmeli, buna uygun kodlama platformu ve kütüphanelerin seçilmesi ve uygun bir tasarımin yapılması gereklidir. Bu ders kapsamında sadece akıllı telefonlarda geliştirilecek uygulamalar için temel bilgileri içeren bir müfredat anlatılacaktır. Diğer aksesuarlar için gerekli proje şablonları ve kütüphaneler kod geliştirme ortamlarının arayüzünden ve ilgili mobil platformların İnternet sayfalarından incelenebilir.

1.3. Mobil Programlama Dilleri ve Türleri

Flutter ile mobil programlamaya geçmeden önce piyasadaki mobil programlama dillerine ve bunların türlerine bir ışık tutmakta fayda vardır. Eğer mobil bir uygulama geliştirilmesi gerekiyorsa bir proje yöneticisi elindeki insan kaynağı ve geliştirilecek iş modeline göre uygun kodlama platformunu seçebilmelidir. Bu süreçteki yanlış kararlar genelde ilgili uygulamanın farklı bir kodlama dili veya kodlama

çatısı (framework) ile yeniden yazılmasını gerektirebilir. Bu açıdan günümüz popüler mobil uygulama geliştirme platformlarını konuya başlamadan ele almamız ve alternatifleri bilerek konuyu öğrenmemiz faydalı olacaktır. Öncelikle mobil uygulama geliştirme yaklaşımlarını türlerine göre inceleyelim. Karşımıza üç temel yaklaşım çıkmaktadır; Doğal (Native), Çok Ortam Destekli (Cross-Platform) ve Mobil Web (Mobile Web). Her yaklaşımın kendine göre avantaj ve dezavantajları bulunmaktadır. Ayrıca her türde farklı bir kodlama dili ve farklı bir uygulama geliştirme yaklaşımı öğrenilmesi gerekmektedir. Şimdi mobil uygulama geliştirme türlerini ve dillerini detaylıca inceleyelim.

1.3.1. Mobil Programlama Türleri

Doğal (Native)

Adından da anlaşılacağı gibi bu türdeki uygulama geliştirme modeli işletim sisteminin kendine ait kolama dili kullanılarak doğrudan işletim sistemi üzerinde çalışacak uygulamaların geliştirilmesini kapsamaktadır. Android işletim sistemi için **Java** ve **Kotlin** programlama dili iOS işletim sistemi için de **Objective-C** ve **Swift** programlama dilleri bu türde uygulama geliştirmek için kullanılır. **Doğal (Native)** uygulamaların avantajı mobil cihazdaki tüm özelliklere doğrudan erişebilmesi, ara katman kullanmadığı için tam performanslı çalışması ve yine bu sayede bataryayı daha az tüketmeleri olarak sıralanabilir. Dezavantajı ise sadece kendi işletim sistemi üzerinde çalışabilmesidir. Eğer bir uygulamanın hem Android hem de iOS'da çalışması isteniyorsa her ikisinin kendi kodlama dili ile iki uygulama geliştirilmesi gerekmektedir. Genelde her iki mobil platformda birden uzmanlaşmış bir yazılımcı bulunması da mümkün olmadığı için ürün geliştirme maliyetinin ikiye katlanması gerektirmektedir. Bu nedenle geliştirilecek uygulama eğer akıllı telefonlarda donanımsal özelliklere çok bağımlı değilse ve performansta kayıplar uygulamanın çalışmasını aksatmayacak düzeydeyse Doğal (Native) uygulama geliştirme yöntemi küçük ve orta ölçekli yazılım firmaları tarafından daha az tercih edilmeye başlanmıştır.

Not: Native ifadesi işletim sisteminin kendi dilini kullanmak manasında kullanıldığı gibi doğrudan C++ dili ile işletim sistemi çekirdeği üzerine kod geliştirmek için de kullanılmaktadır. Her ne kadar bu başlıkta anlatılan Doğal (Native) uygulama geliştirme yöntemi ile işletim sistemi üzerinde doğrudan çalışan mobil uygulamalar geliştirildiğini söylesek de bu uygulamalar "Sandbox" olarak ifade edilen bir yöntemle birbirinden izole şekilde çalışmaktadır. Yani bir uygulama diğer uygulamanın işlemcide koşan işlemlerine, RAM'deki verilerine ve kalıcı hafızada diğer uygulamalara ait dosya bölümüne erişemezler. Diğer manada kullanılan Native uygulama geliştirme daha alt seviyede işletim sisteminin doğrudan çekirdeği (kernel) üzerinde çalışacak (genel olarak C++ dili ile geliştirilen) uygulamaların kodlanması için kullanılmaktadır. Android için çekirdek(kernel) seviyesinde geliştirme Native Development Toolkit (NDK) ile yapılmaktadır. Bu yöntem C++ ile yazılmış hazır kütüphaneleri kodda çağrırmak ve performans gereken kısımların C++ ile kodlanması amacıyla ile de kullanılmaktadır. Coğunlukla geliştirilen uygulamaların işletim sistemi çekirdeğine doğrudan erişim ihtiyacı olmamaktadır. Mobil platformlardaki güvenlik gereksiniminden dolayı mobil platformların marketlerinden indirilen tüm uygulamalar Sandbox yöntemi ile izole şekilde çalışırlar.

Çok Ortam Destekli (Cross-Platform)

Cross-Platform bilgisayar terminolojisinde tek bir dil ile geliştirilen bir uygulamanın birden fazla platformda (işletim sistemi türünde) değiştirilmeden çalışabilmesi veya aynı anda birden fazla platforma göre derlenebilmesini ifade etmektedir. Bu geliştirme modelinde uygulama, işletim sisteminin kendi kodlama dilinden bağımsız bir dil ile oluşturulmaktadır. Uygulamalar, işletim sistemi üzerinde çalışabilmesi için yardımcı kütüphane dosyaları ve özel bir kod dönüştürücü yapısını bünyelerine dahil eder veya harici olarak kullanırlar. Doğal (Native) geliştirme yöntemine göre daha az işgücü gerektirdiği için avantajlidir. Uygulamanın işletim sistemi üzerinde çalışabilmesi için ara katman içerdiginden performans kaybı bir dezavantajı olarak görülebilir. Bu performans kaybı akıllı telefonlardaki işlemci kapasitelerinin artması sayesinde çoğu uygulama türü için artık önemsenmeyecek seviyelerdedir ve ara katmanların getirdiği ek yük giderek azaltılmaya çalışılmaktadır. Diğer bir dezavantaj ise kullanılan geliştirme modeline göre akıllı telefonların bazı donanımsal özelliklerine erişmek için ilgili işletim sisteminin doğal (native) dili ile kod geliştirmeye ihtiyaç duyulabilmesidir. Mobil uygulama geliştirme piyasasına bakıldığından Cross-Platform dillere olan ilginin giderek arttığı görülmektedir.

Mobil Web (Mobile Web)

Diğer bir mobil uygulama geliştirme modeli olarak da web tabanlı uygulamaları görmekteyiz. Web sayfalarının mobil platformlarda bir uygulama gibi görülmemesini sağlamak suretiyle web içeriğinin bir uygulama gibi son kullanıcıya sunulması sağlanır. Mobil platformlarda İnternet sitelerinin daha fazla ziyaret edilmesi sonucu mobil platformlardan gelen kullanıcılarla iyi bir sunum sağlanması artık internet sayfalarında aranan temel özellik haline gelmiştir. Farklı ekran çözünürlüklerinde ve farklı ekran boyutlarında kullanıcıların veriye erişimi ve etkileşimi düzenleyen bir web sayfası geliştirme modeli oluşmuştur. Bu model günümüzde “Responsive Web Design” olarak nitelendirilmektedir ve bir İnternet sitesinin mobil platformlarda görüntülenmeye uygun olması manasında da kullanılmaktadır. Mobil Web uygulama türünde, kullanıcının uygulamayı kullanabilmesi için çevrimiçi olması gerekmektedir. Bu gereksinimin ortadan kaldırılması adına Mobil Web türündeki uygulamalarda İnternet sayfası içeriği cihazda yerel olarak kaydedilir ve uygulama içerisinde kullanıcı gezintisi yerel kopya sayesinde çevrimdışı iken de sağlanır. Bu yaklaşımда responsive özelliklerde sunulurken kullanıcı deneyimi sanka doğal(native) bir uygulama kullanıyor gibi sağlanır. Yaklaşım isimlendirme olarak “Progressive Web App” adı verilmiştir. Ayrıca İnternet sitesinin ana sayfasının kısa yolu akıllı telefondaki masaüstüne kaydedilerek kullanıcıya sanka doğal(native) bir uygulama kullanıyor hissi verilmeye çalışılır.

Mobil Web türündeki avantaj, geliştirilmesi için mobil platforma özgü harcanması gereken kaynak çok azdır. Sistemin zaten kullandığı bir web platformu ve bu platformu kodlayan ekibi var ise yeni bir personel almaksızın mobil platformlarda çalışacak mobil web türü uygulamalar geliştirilebilir. Zira günümüzde artık web sayfaları tamamen mobil desteği verecek şekilde inşa edilmektedir. Mobil web türü uygulama geliştirme süreci diğer turlere göre daha hızlı ve kolay olmaktadır. Çünkü zaten piyasada web teknolojileri ile ilgilenmiş yazılımcı sayısı fazla ve ücret skalası mobil platform kodlama yapanlara göre aşağıdadır. Mobil web türünün getirdiği kritik zorluksa telefonun donanımsal yapısından tamamen bağımsız çalışmasıdır. Bir internet tarayıcısı aracılığı ile çalışan uygulamamız telefonun donanımsal bileşenlerin nerdeyse tamamına doğrudan erişemez. Bunun için ek bir doğal(native) kodlanmış bölümün uygulamaya dahil edilmesini gerektirir. Bu gibi durumlar da mobil web türünün getirdiği hızlı üretilme avantajını ortadan kaldırılmaktadır. Bakıldığından eğer bir uygulama aynı zamanda bir İnternet sayfası üzerinden son kullanıcıya sunulan bir hizmeti aynı şekilde mobil platformlarda vermek niyetiyle yapılacaksa tercih edilmeli, telefon donanımı ile etkileşen bir uygulama geliştirilecek ise tercih edilmemelidir.

1.3.2. Mobil Programlama Dilleri

Her gelişen teknoloji dalı için çeşitlilik nasıl artıyorsa mobil platformlar için uygulama geliştirilecek diller de artmaktadır. Öncelikle doğal (native) uygulama geliştirme dillerini sonra da cross-platform uygulama geliştirme dillerini inceleyelim.

Android: Java - Kotlin

Android işletim sistemi için ana uygulama geliştirme dili Google tarafından Java olarak tercih edilmiştir. Android işletim sistemi Java dilinde yazılan kodları çalıştırma için bir sanal makine “Android Runtime (ART)” kullanmaktadır. Bu yaklaşım masaüstü sistemlerdeki Java Sanal Makinesine benzer şekilde çalışır. Android işletim sisteminin fonksiyonları ART sayesinde uygulama geliştiricilerinden izole edilir ve bu sistem güvenliğini arttırır. Google, Android için geliştirme ortamını “Android Studio”, yazılım geliştirme kitini “Software Development Kit (SDK)”, belgelendirmeleri ve birçok yardımcı materyali <https://developer.android.com/> sayfası üzerinden sunmaktadır.

Kotlin dili yine Google tarafından Android SDK’sı kullanarak Android işletim sistemlerine doğal (native) uygulama geliştirmek için üretilen bir dildir. Kotlin dili Java’da görülen hantallıkların giderilmesi ve daha az kodlama yaparak daha çok işlevi yerine getirmeyi amaçlayan bir iyileştirmektedir. Kotlin ile yazılan uygulamalar yine ART üzerine çalışacağı için Java bytecode’ya dönüştürülürler. Kotlin dili Android platformu için doğal(native) kodlama yapmayı düşünecek yazılımcılar için günümüzde öğrenilmesi gereken programlama dilidir.

iOS: Objective-C – Swift

iOS işletim sistemi için ise Apple firmasının ilk tercih ettiği dil Objective-C dildidir. Bu dil C dilinin sentaksını kullanır ve bunun üzerine nesneye yönelimli bir yaklaşım getirmiştir. C++ dilinden farklı bir dildir. Apple’ın Swift dilini kullanmaya başlaması ile artık mobil geliştirme için tercih edilmemektedir.

Swift iOS için doğal(native) uygulama geliştirmede kullanılan programlama dilidir. MacOS işletim sistemi üzerinde çalışan Xcode geliştirme ortamı ile kodlanabilmektedir. Swift dilinde Objective-C ile yazılmış kütüphaneleri kullanıma destek de sunulmaktadır. Apple firması iOS üzerinde geliştirme yapacak yazılımcılara tüm gerekli içeriği <https://developer.apple.com/> internet adresi üzerinden sunmaktadır. iOS için doğal(native) uygulama geliştirmek için MacOS işletim sistemi ve Xcode editörüne sahip olmak gerekmektedir. Apple firması bu tutumuyla uygulama geliştiricileri yine kendi ürünlerini kullanıma ve kendi ekosistemleri içerisinde kalmaya zorlamaktadır.

Xamarin & C#

Xamarin, Microsoft Firmasının .Net platformunun mobil uygulama geliştirmede kullanımını için açık-kaynak olarak geliştirilmiş bir cross-platform çerçevedir. Xamarin ile aynı anda Android, iOS ve Windows işletim sistemlerinde çalışacak uygulama geliştirilebilir. Xamarin bir soyutlama katmanı kullanarak ortak uygulama kodlarını alt tarafta yer alan platform kodları ile haberleştirir. Microsoft mobil platformlarda kaybettiği yarısı Xamarin ile yazılım sektöründe devam ettirmeye çalışmaktadır. Ne yazık ki Xamarin de alternatifi cross-platform geliştirme ortamları ile rekabette geri kalmıştır. Yine de .Net teknolojilerine hâkim bir yazılımcının mobil platformlar için uygulama geliştirmesi gerektiği noktada uygun bir çözüm olarak karşımıza çıkmaktadır. PC veya Mac üzerinde Xamarin ile geliştirme yapılmaktadır.

Apache Cordova & HTML-CSS-JavaScript

Cross-Platform uygulama geliştirmede web teknolojilerinin kullanıldığı bu yaklaşımda uygulamanın özü frontend web teknolojileri olan html, css ve JavaScript dilleri ile bir web sayfası kodları gibi geliştirilir. Telefon donanımına erişim ise web servisi çağrıları gibi Cordova çatısına yapılan API çağrıları ile gerçekleştirilmektedir. Uygulamanın kendisi de bir internet tarayıcısı gibi çalışan WebView bileşeni üzerinde olmaktadır. Android, iOS, Windows ve başka ortamlarda çalışabilen uygulamalar geliştirme imkânı sunar.

React Native & JavaScript

React Native, Facebook firmasının desteklediği ve en yaygın kullanıma sahip cross-platform mobil uygulama geliştirme platformudur. Web geliştiricilerin aşina olduğu JavaScript dilini doğal(native) uygulama içerisinde arayüz çizimi için kullanan bir yapısı vardır. React Native JavaScript dilinde geliştirilen arayüz çizimini çalışma zamanında üzerinde çalışacağı işletim sisteminin kendi arayüz bileşenlerine çevirmek suretiyle doğal(native) bir uygulama hissiyatı verir. Çalışma esnasında gerçekleşen dönüştürme işlemi kabul edilebilir bir performans kaybı getirmektedir. React Native ciddi bir geliştirici topluluğuna ve çok sayıda hazır bileşenlere sahiptir. Dil olarak JavaScript kullanıyor olsa da web geliştirmede kullanılan bileşenlerden farklı isimde bileşenler kullanılmaktadır. Arayüz çizimi ve durum yönetimi de web uygulaması geliştirmeden farklıdır.

Flutter & Dart

Google firması 2011 yılında Dart dilini ilk defa tanıtmıştır. İlk stabil sürümünü 2017 yılında piyasaya sürmüştür ve sonrasında Flutter Framework’ü cross-platform mobil uygulama geliştirme için tasarlamıştır. Flutter diğer cross-platform uygulama geliştirme yöntemlerinden esinlenerek hazırlanmıştır. Bu açıdan çalışma mekanizmasının pek çok yanı React Native’e benzemektedir. React Native’den ayrılan en önemli özelliği bileşenlerinin üzerinde çalıştığı işletim sistemi bileşenlerine dönüştürülmemesi bunun yerine Skia grafik motorunu kullanarak arayüzü doğrudan çizmektedir. Bu yönü ile Flutter, React Native’e kıyasla arayüz render işleminde ciddi bir performans artışı kazanmaktadır. Flutter aslında bir arayüz çizim çatısıdır (Framework). Kodlama dili olarak kullanılan Dart dili nesneye yönelik C tabanlı dillerin (C++, C#, Java, JavaScript) iyi yanlarının bir araya getirilmiş bir kombinasyonudur denebilir. Bu nedenle daha önce bu dillerden birini kullanmış bir yazılımcının Dart diline hâkim olması çok kısa bir zamanda gerçekleşmektedir. Flutter’ın ilginç özelliklerinden biri ise arayüzün nasıl olacağına derleme aşamasında değil çalışma zamanında karar vermesidir. Bu sayede farklı platformlarda aynı görselliği sağlayabilmektedir. Flutter ile Android, iOS, Web, Windows, Linux, MacOS ve hatta farklı gömülü sistemler için bile uygulama geliştirilebilmektedir. Flutter’ın tasarımındaki esneklik onun ileride farklı platformları da destekleyebilmesini sağlamaktadır. Flutter, Google firmasının desteğini aldığı ve geniş bir topluluğa sahip olduğu için iki yıl içerisinde çoğu cross-platform mobil uygulama geliştirme araçlarından daha popüler hale gelmiştir. Üçüncü parti bileşenler açısından eski platformlara göre yeterli alternatif olmamasına karşın Google’ın üçüncü parti

bileşenleri sunduğu paket kütüphanesi ve yürüttüğü “Flutter Favorites” programı ile uygulama geliştiricilerin bu yönde aradıklarını bulma ve doğru bileşenleri edinmesine yardımcı olmaktadır.

Diger Cross-Platform Uygulama Geliştirme Ortamları

Yukarıda sıraladığımız uygulama geliştirme ortamları dışında özellikle cross-platform gurubunda sayabileceğimiz çok farklı alternatifler bulunmaktadır. Bunlardan Ionic yukarıdakilerden sonra gelen en popüler alternatif olarak değerlendirilebilir. Diğer alternatiflerin olduğunu bilmekle beraber eğer bu alana yeni giriş yapılacaksa en popüler olanları veya topluluğu aktif ve giderek büyüyenlerden birini tercih etmek yerinde olacaktır. Diğer ortamları keşfetmek için İnternet üzerinde basit bir araştırma yapmak yeterli olacaktır. Her yıl aktif olan geliştirme ortamları değiŞebileceği için doğrudan bir kaynak paylaşımamıştır.

1.4. Neden Flutter?

Sayıdığımız bunca mobil uygulama geliştirme türleri ve dilleri arasında biz neden Flutter öğrenmeliyiz sorusuna birkaç madde ile cevap verelim;

Flutter cross-platform bir uygulama geliştirme platformudur ve Android, iOS, Web, Windows, Linux, MacOS ve giderek genişleyen bir çalışabildiği platform desteği bulunmaktadır.

Flutter Skia grafik motorunu kullandığı için arayüz derleme işleminde performans kaybı nerdedeyse yoktur. Doğal (Native) bir uygulamanın arayüz çizimine eş bir performansta çalışır.

Arkasında Google gibi sektörü yönlendiren bir firma bulunmaktadır ve Flutter kullanan-geliştiren topluluğu giderek büyümektedir.

Arayüz tasarımları için kullandığı temel bileşenlerin (Widgets) öğrenilmesi ve kullanımı kolaydır.

Telefondaki bileşenlere Flutter’da geliştirilmiş kütüphanelerle erişmek mümkündür. JavaScript temelli cross-platform geliştirme yöntemlerindeki gibi bir API çağrısı yapılmaz doğrudan doğal (native) kod ile bütünsüzleşmiş çalışmaktadır.

Endüstrinin ihtiyaçlarına hızlıca uyum sağlayabilecek formdadır. Sayısız Widget (Flutter’daki arayüz bileşenlerinin genel adı) ve doğal(native) kod yazma imkânı sunar.

Flutter kendi içerisinde bir uygulama yaşam döngüsü yönetimi sunmaktadır. Web temelli çözümlerde bu özellik ek bileşen olarak bulunmaktadır. Web programlama kendi doğası gereği stateless (durumsuz) çalışmaktadır.

Flutter kodlama yapısı Android’ın doğal(native) kodlama yapısına çok yakındır. Google burada ekiplerarası deneyim aktarımı ile bir avantaj oluşturmuştur.

Bu maddeleri daha detaylı incelemek isteyenler bağlantıdaki içeriği veya İnternet’te bulunan Flutter özelliklerinin diğer alternatifleriyle karşılaştırmalarını inceleyebilirler.

1.5. Flutter Kurulumu ve İlk Proje Oluşturma

Google firması Flutter öğrenmek ve Flutter ile geliştirme yapmak isteyenler için gerekli tüm içeriği <https://flutter.dev/> İnternet adresinden sunmaktadır. Bu ders içeriğinde bu İnternet adresine sürekli referanslar verilecektir ve buradaki içeriği daha rahat takip etmeniz için yönlendirmelerde bulunulacaktır.

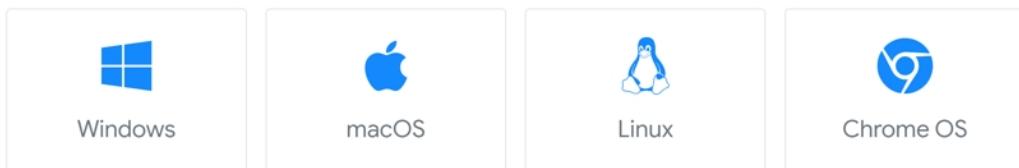
Flutter’ın İnternet sayfasını ziyaret ettiinizde sağ üst köşede “Get Started” butonu görülmektedir. Bu butona tıkladığımızda bizi doğruba kurulum ekranı ile karşılaşmaktadır.



Install

[Docs](#) > [Get started](#) > [Install](#)

Select the operating system on which you are installing Flutter:



Flutter cross-platform bir uygulama geliştirme çatısı olmasının yanında istenilen işletim sistemi üzerinde geliştirme yapmaya da olanak tanımaktadır. Kurulum ekranından seçilen işletim sistemine göre Flutter için gereksinimler ve adım adım kurulum prosedürü verilmektedir. Hangi işletim sistemine sahip olursanız olun Flutter'da cross-platform uygulama geliştirebilirsiniz fakat iOS ortamında uygulamanızı test etmek istiyorsanız MacOS işletim sistemine sahip olmanız gerekmektedir. Kurulum aşamasından sonraki ders içeri Windows 10 bir işletim sistemi üzerinden gerçekleşecektir. Ama anlatılan içerik işletim sisteminden bağımsızdır. İşletim sistemleri arasındaki tek fark kurulum gereksinimleri ve kurulum adımlarıdır.

Gereksinimler

Kurulum işlemindeki ilk adım kurulum yapacağımız işletim sistemine göre sistem gereksinimlerine bakmak olacaktır. Eğer kurulum yapmak istediğimiz sistem bu gereksinimleri karşılamıyorsa muhtemelen Flutter veya Flutter'ın çalışabilmesi için gerekli bileşenleri kuramayacaksınız. Bu nedenle kurulum işlemine başlamadan önce kurulum yapmak istediğiniz sistemin gereksinimleri tam olarak karşıladığından emin olunuz. Gereksinimler işletim sistemine göre farklılık gösterdiği için gereksinimleri işletim sistemine göre inceleyelim.

Windows

İşletim Sistemi: Windows 7 SP1 veya üstü (64-bit), x86-64 tabanlı.

Disk Alanı: 1.64 GB (gerekli IDE ve diğer araçların kurulumu için önerim 20 GB boş alana sahip olmanızdır.)

Araçlar: Flutter çalıştırırken bazı ek araçları kullanmak gerekecektir. En azından sisteminizde;

- **Windows PowerShell 5.0** veya üst sürümü olmalıdır. (Windows 10'da PowerShell kurulu gelmektedir.)
- Windows komut satırı için Git uygulaması **Git for Windows**. Eğer bilgisayarınızda Git zaten kuruluysa komut satırına git komutunu yazdığınızda aşağıdaki gibi bir ekran görmemeniz gerekmektedir. Kurulumda Git'i kurduğunuz yeri Ortam Değişkenlerinde (Environment Variables) Path bilgisi olarak eklemeniz ve kurulum yaptıktan sonra bilgisayarınızı yeniden başlatmanız gereklidir.

```
PowerShell 7 (x64)
PS C:\Users\crane> git --version
git version 2.31.1.windows.1
PS C:\Users\crane>
```

MacOS

İşletim Sistemi: MacOS

Disk Alanı: 2.8 GB (gerekli IDE ve diğer araçların kurulumu için önerim 20 GB boş alana sahip olmanızdır.)

Araçlar: Flutter kurulum ve paket yönetimi için **git** kullanmaktadır. MacOS'de çalışmak için kurulacak Xcode IDE'si git'i bünyesinde barındırmaktadır. Ama istenilirse git ayrı olarak da kurulabilir.

Linux

İşletim Sistemi: 64-bit Linux

Disk Alanı: 600 MB (gerekli IDE ve diğer araçların kurulumu için önerim 20 GB boş alana sahip olmanızdır.)

Araçlar: Flutter sisteminizdeki paket yöneticileri aracılığı ile indirebileceğiniz komut satırı araçlarına ihtiyaç duyar.

- **bash**

- **curl**

- **file**

- **git 2.x**

- **mkdir**

- **rm**

- **unzip**

- **which**

- **xz-utils**

- **zip**

Ortak Kütüphaneler: Flutter test komutu sistemde olan **libGLU.so.1** paketine ihtiyaç duymaktadır. Ubuntu/Debian işletim sistemlerinde **libglu1-mesa**, Fedora işletim sisteminde **mesa-libGLU** ismindeki mesa paketi olarak dağıtılmaktadır.

Flutter SDK Kurulumu

İşletim sisteminize göre temel gereksinimleri karşıladıktan sonra Flutter SDK (Software Development Kit) kurulumuna geçebilirsiniz. Flutter sürekli güncellenen ve farklı kullanıcı tiplerine göre farklı kanallarla(channel) sunulan bir yapıya sahiptir. Bu dağıtımlara SDK Releases sayfası üzerinde erişebilirsiniz. SDK Release sayfasındaki kanallar;

Stable: Flutter SDK'sının stabil yani üretim projelerinde kullanılabilir halini ifade etmektedir. Bu kanalda verilen SDK özellikleri bütün test aşamalarından başarıyla geçmiş demektir ve uygulama geliştirmede gönül rahatlığıyla kullanılabilir. Google Play Store'a ve Apple Store'a yüklenecek uygulamaların muhakkak stable channel ile üretilmiş uygulamalar olması gereklidir.

Beta: Beta aşamasındaki sürümleri simgeler. Geliştirme aşaması tamamlanmış ve topluluğun deneyimlemesine sunulmuş sürümler bu kanalda verilmektedir. Beta kanalında Stable kanalındaki sürümler de yer alır.

Dev: Development Channel yani üretim kanalıdır. Henüz tam olarak test edilmemiş ve dile yeni eklenen özellikleri derlenerek sunulduğu kanaldır. Stable ve Beta kanalında olmayan ve genellikle ayda en az bir kere yenilenen sürümler bu kanalda sunulmaktadır.

Master: GitHub üzerindeki aktif proje kanalıdır. Bu kanal sadece SDK geliştiricilerine hitap etmektedir. Master kanalındaki kodlar her zaman tutarlı çalışmayıabilen. Flutter'a eklenen en yeni özellikler aynı anda bu kanal aracılığı ile deneyimlenebilmektedir.

Biz kurulum için “Stable Channel” üzerindeki en son sürümü kullanacağız. Bu ders notu hazırlanırken son stabil sürüm 2.2.3 sürümüdür. Bunun için indirme bağlantısı kurulum adımlının ikinci aşamasında işletim sistemi seçiminize göre aşağıdaki gibi görünecektir. Kurulum işleminin bundan sonraki aşamaları Windows işletim sistemine göre anlatılacaktır. Diğer işletim sistemleri içi flutter.dev adresinden kurulum adımlarını takip edebilirsiniz.

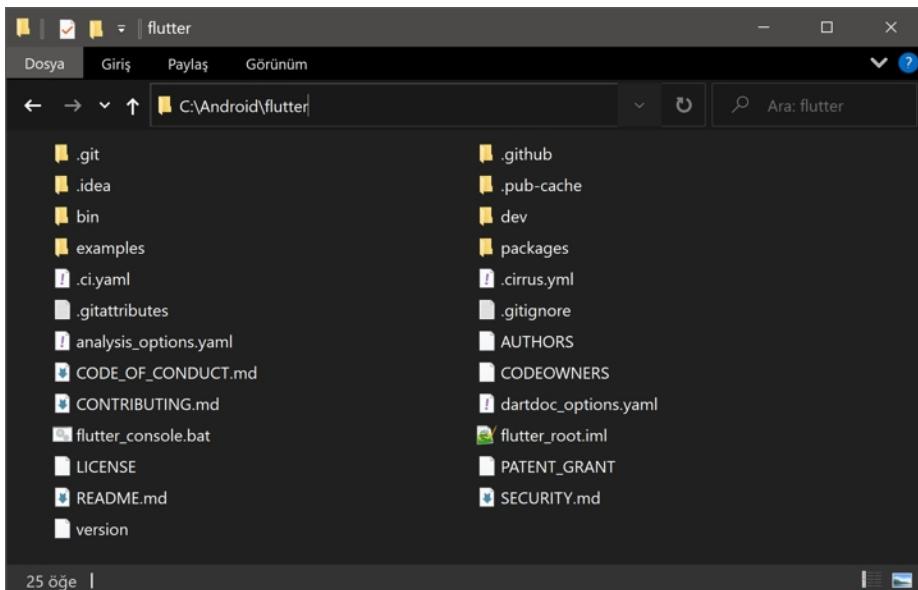
<https://flutter.dev/docs/get-started/install/windows>

Get the Flutter SDK

1. Download the following installation bundle to get the latest stable release of the Flutter SDK:

[flutter_windows_2.2.3-stable.zip](#)

Burada gösterilen zip dosyasını bilgisayarınıza indirerek istediğiniz bir klasör altında ayıklamak suretiyle Flutter SDK’yı bilgisayarınıza yükleyebilirsiniz. SDK’yı yükleyeceğiniz klasörün Türkçe karakter, boşluk, tire(-) ve özel herhangi bir noktalama işaretini içermemesine özen gösterin. Aksi halde proje derleme aşamasında bazı dosyaların bulunmasında sıkıntılar yaşayabilirsiniz. Buradaki anlatımda Windows işletim sisteminde “C:\Android” altında “flutter” olarak isimlendirilen bir klasörde zip dosyası ayıklanmıştır. Ayıklanan SDK kurulumu aşağıdaki gibi bir klasör yapısı sağlayacaktır.



Kurulum dizini olarak “Masaüstü” veya “Program Files” gibi özel kullanıcı izinleri gerektiren klasörleri seçmeyiniz.

Yukarıda anlatılan SDK zip dosyası ayıklaması yerine git üzerinden de kurulum işlemini yapabilirsiniz. Bunun için komut satırında veya PowerShell üzerinden kurulum yapmak istediğiniz klasöre giderek aşağıdaki komutu çalıştırınız. (Not: gereksinimlerdeki git kurulumunu yapmış olmanız gereklidir.)

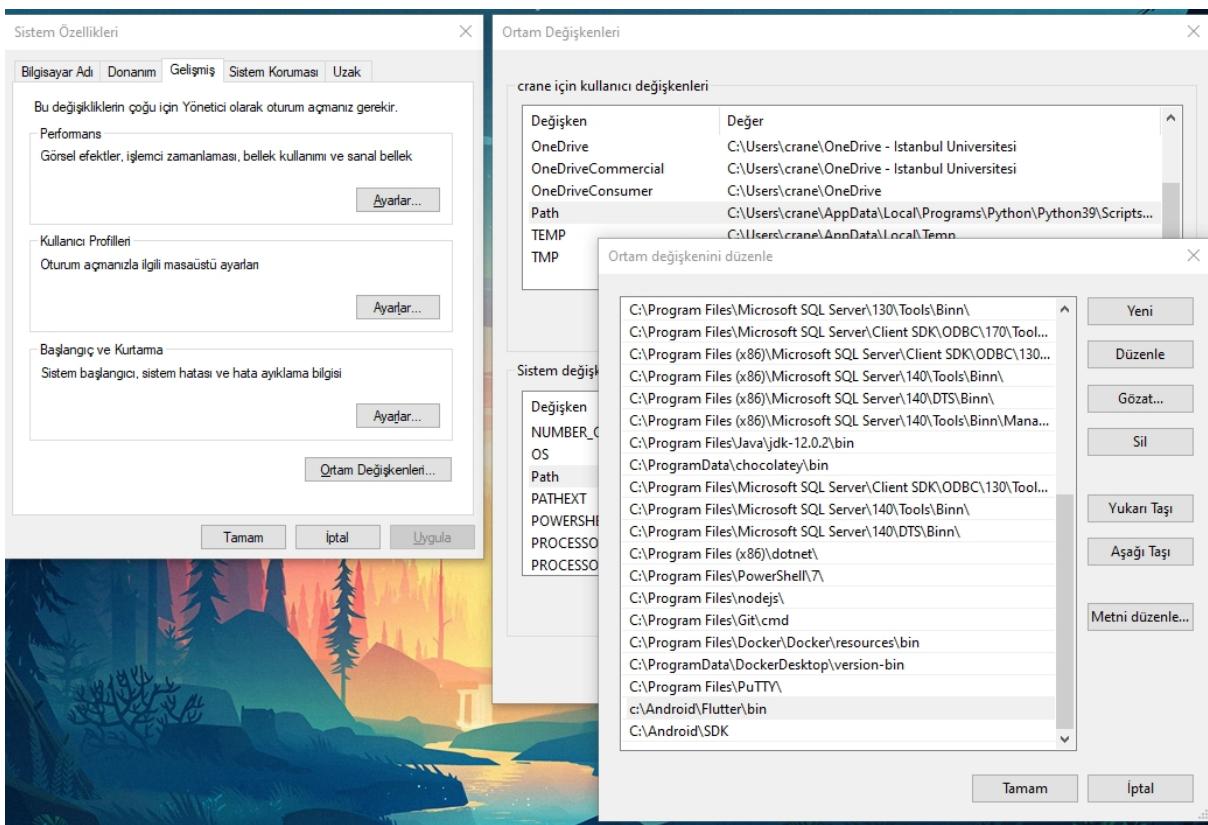
```
C:\Android>git clone https://github.com/flutter/flutter.git -b stable
```

Flutter SDK Yolunun İşletim Sistemine Gösterilmesi

Flutter SDK’sı bir klasör içerişine yüklendiği için bu klasör içerişindeki çalıştırılabilir dosyaların ve kütüphanelerin işletim sistemi tarafından görünmesi sağlanmalıdır. Bunu gerçekleştirmek için;

Windows işletim sisteminde “**Ortam Değişkenleri**” (Environment Variables) içerisinde kullanıcı değişkenleri (User Variables) başlığında bulunan **Path** bilgisine Flutter SDK içerisindeki “bin” klasörünün

yolunun eklenmesi gerekmektedir. Ortam Değişkenlerini düzenlemek için Windows Başlat menüsünden aratarak ulaşabilirsiniz. Düzenleme için yönetici haklarına sahip olmanız gereklidir. Yukarıdaki kurulum için Path girdisine “c:\Android\flutter\bin” eklenmesi gerekmektedir. Kurulum yaptığınız yere göre bunu düzenlemeniz gereklidir. Aşağıda yapılan işlemin örnek görseli verilmiştir.



Linux ve MacOS işletim sistemlerinde yönetici hakkı ile aşağıdaki komutu çalıştırmanız yeterli olacaktır. Burada kurulumun “/flutter” altında yapıldığı varsayılmaktadır.

```
export PATH="$PATH:'pwd'/flutter/bin"
```

Path bilgisinin güncel halinin işletim sistemi tarafından kullanılabilmesi için kullanıcı oturumunun kapatılıp açılması veya bilgisayarın yeniden başlatılması gerekmektedir. Kurulumun bu aşamasından sonra terminal penceresinde (Windows'ta konsol-PowerShell olarak geçmektedir. Konsol-Terminal terimleri metin içerisinde aynı manada kullanılmaktadır.)

```
flutter doctor
```

komutunu girerek Flutter SDK yol tanımlamasının doğru olup olmadığını kontrol ediniz. Aşağıdaki gibi bir çıktı almanız gerekmektedir.

```
Doctor summary (to see all details, run flutter doctor -v):
[!] Flutter (Channel dev, 2.2.0-10.1.pre, on Microsoft Windows [Version 10.0.19043.1165], locale tr-TR)
[!] Android toolchain - develop for Android devices (Android SDK version 30.0.2)
[!] Chrome - develop for the web
[!] Visual Studio - develop for Windows (Visual Studio Enterprise 2019 16.10.4)
[!] Android Studio (version 4.1.0)
[!] IntelliJ IDEA Community Edition (version 2021.1)
[!] Connected device (3 available)
```

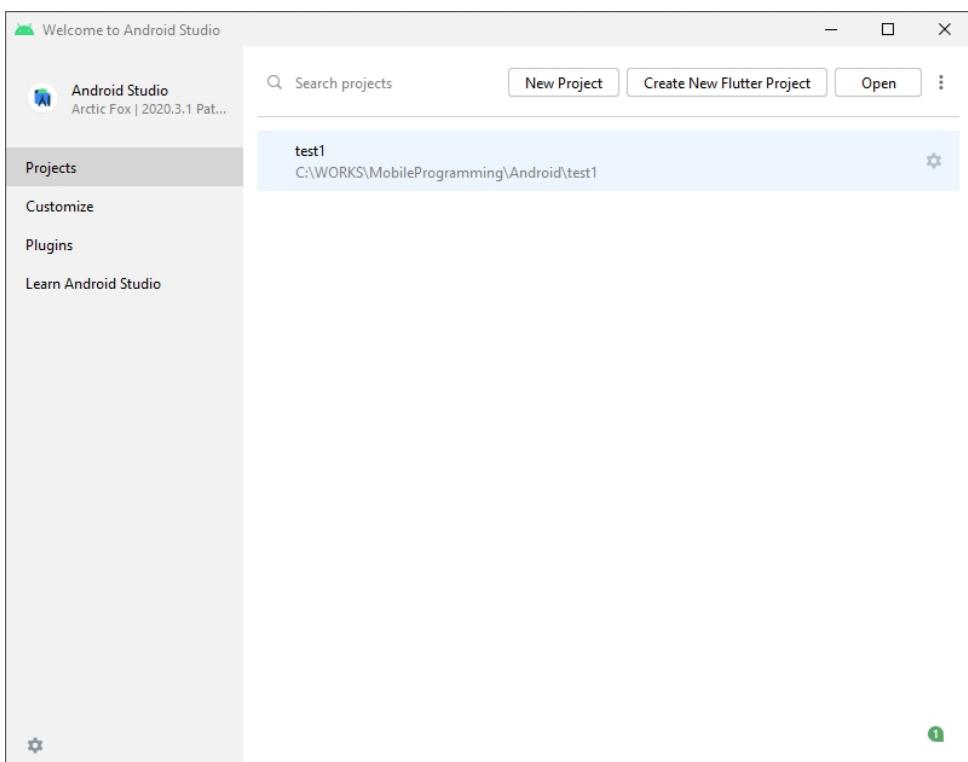
Alacağınız çıktıda buradakinden farklı olarak birçok maddenin “X” ile işaretli olduğunu göreceksiniz. Bu bir sıkıntı değildir. “**flutter doctor**” komutu bilgisayarınızdaki kurulumları test ederek Flutter kodlamamanız için gerekli yazılımların ve SDK’ların varlığını kontrol etmektedir. Devam eden maddelerde bu kurulumları tamamlayacağız.

Not: Google Flutter kurulumunda varsayılan ayar olarak Flutter’ın iyileştirilmesi için ananım kullanım bilgilerini toplamaktadır. `flutter config --no-analytics` komutunu kullanarak bu bilgi gönderimini kapatabilirsiniz.

Android Studio ve Android SDK Kurulumu

Flutter kodlamak için birden fazla IDE (Integrated Development Environment- Tümleşik Geliştirme Ortamı) kullanmamız mümkün değildir; Android Studio, IntelliJ, Visual Studio Code. İçerigin devamında Microsoft firmasının geliştirdiği ve kullanım açısından yalın ve esnek olduğu için çoğunlukla tercih edilen Visual Studio Code (VS Code) IDE'si kullanılacaktır fakat VS Code kurulumundan önce Android geliştirme yapmak için Google firmasının sunduğu Android Studio IDE'sinin kurulumunu yapmamız gerekiyor. Android Studio, Android için uygulama geliştirmek ve test etmek için gerekli Android SDK ve Emülatörlerini bize kolayca sağlayacaktır. Android SDK'sını Android Studio olmadan da bilgisayarınıza kurmanız mümkün ama bu hem daha zahmetli hem de Flutter kurulumunu doğrulamada bize zaman kaybettirecektir. Bu nedenle Flutter kodlamaya geçmeden önce Android Studio ve Android SDK'sını kurmamız gerekmektedir.

Android Studio IDE'sini <https://developer.android.com/studio> adresinden indirebilirsiniz. Android Studio kurulumunda size neleri kurmak istedığınızı soracaktr. Varsayılan kurulum genel olarak ihtiyacımızı karşılamaktadır. Android Studio kurulumunda dikkat etmemiz gereken önemli nokta Android SDK için seçilecek kurulum dizinidir. Eğer bu kurulum dizinini değiştirmezseniz kullanıcı hesabınızın altında kurulum yapılacaktır. Bu önceden belirttiğimiz gibi Türkçe işletim sistemlerinde bazen derleyicilerin çalışmaması gibi problemlere yol açmaktadır (Özellikle kullanıcı adınız Türkçeye özgü karakterler içeriyorsa). Bu nedenle kurulum aşamasında Android SDK yolunu "C:\Android\SDK" şeklinde değiştirmenizi tavsiye ederim. Kurulumda, güncel Android SDK, Android SDK Command-line Tools. Android Studio ve Android SDK'nın çalışması için bilgisayarınızda Java Development Kit (JDK)'nın kurulu olması gerekmektedir. Bilgisayarınızda Java 8 veya üzeri bir sürümün kurulu olduğunu kontrol etmelisiniz. JDK'nın kurulu olduğu dizinin de Ortam Değişkenlerinden (Environment Variables) **JAVA_HOME** adında bir kayıta yazılması gerekmektedir.



Android Studio kurulumu bittiğinde flutter doctor komutu ile "Android Toolchain" başlığının onaylanıp onaylanmadığını kontrol ediniz. Not: Bu denemeyi yapmadan önce Android Studio kurulumundan sonra oturumu yeniden başlatınız. Eğer "flutter doctor" komutunda Android Toolchain için onay göremezseniz;

```
flutter config --android-studio-dir <directory>
```

komutunu "<directory>" kısmına Android Studio IDE'sini kurduğunuz yolu girerek çalıştırabilirsiniz. Not: Burada Android SDK'sı değil Android Studio IDE'si yolunu istemektedir.

Bazı ek özelliklerin çalışması ve Android bir cihaz ile denemeler yapmak için;

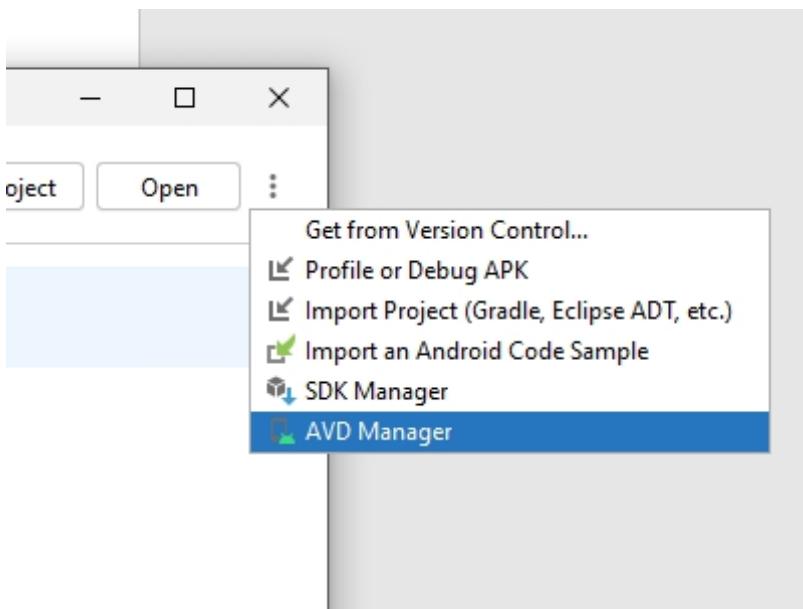
Android cihazınızda yazdığınız kodları test etmek istiyorsanız cihazın geliştirici özelliklerini (Developer Options) açmanız burada USB hata ayıklama (USB Debugging) aktif hale getirmeniz gerekmektedir. Detaylı anlatım Android'in kendi dokümantasyonunda mevcuttur.

Sadece Windows İşletim Sisteminde “Google USB Driver” kurulmalıdır.

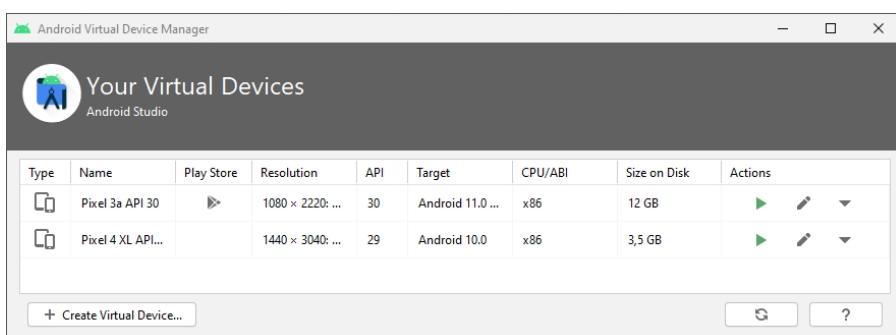
Android SDK’sını kurduğunuz yolu Ortam Değişkenlerinde “**ANDROID_SDK_ROOT**” anahtarı ile kaydediniz. Normalde Flutter Android Studio kurulumundaki tanımlı Android SDK yolunu kullanmaktadır.

Emülatörler farklı işletim sistemlerinde çalışacak kodların denenmesi için kullanılmaktadır. Flutter ile Android ve iOS işletim sistemlerine geliştirme yapacağımız için bu işletim sistemlerini bir emülatör aracılığı ile Windows, Linux veya MacOS işletim sistemleri altında çalıştırmalıyız. Android Studio kendi içerisinde bir Emülatör yönetim uygulaması (AVD Manager) ve farklı özelliklerde emülatör imaj dosyaları ile sunulmaktadır. Android Studio kurulumu sonrasında AVD Manager uygulamasını giriş ekranında sağ üst

köşedeki üç nokta menüsü altından veya IDE açıldığı zaman üst kısımdaki araç çubوغunda simgesine tıklayarak açabilirsiniz.



AVD Manager içerisinde bir adet Android işletim sistemi imajı oluşturmamız yeterlidir. Her tür işleminizde kullanmanız için yeni sanal makine oluşturma adımlarındaki sihirbazın önerilerini takip edebilirsiniz. Aşağıda AVD Manager'a ait bir görsel ve oluşturulmuş iki farklı Android imajı bulunmaktadır. “**+Create Virtual Device**” butonuna tıklayarak yeni imaj oluşturma sihirbazını açabilirsiniz.

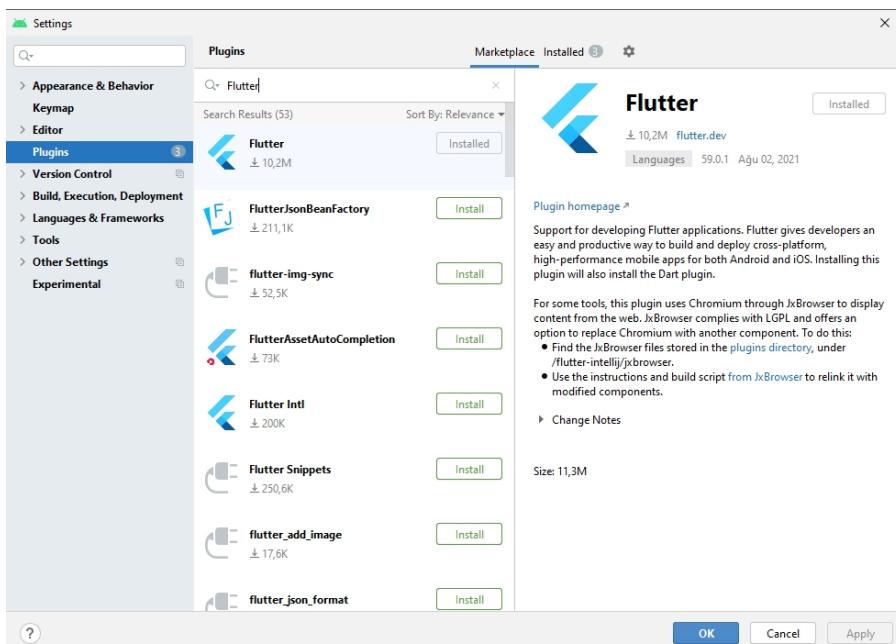


Eğer imajınızın Google Play Store üzerinden sunulan servisleri de simüle etmesini istiyorsanız Play Store ikonu bulunan imajlardan birini tercih ediniz. AVD kurulumu bilgisayarlarınızdaki sanallaştırmanın açılmasını (BIOS üzerinden) ve intel işlemcili makinelerde HAXM hızlandırıcısının kurulumunu isteyebilir. Google imaj dosyasının x86 işlemcisi olan sürümlerini tercih etmenizi önerir (Hız açısından). İstenilirse sanal cihaz imajı oluşturma ekranında ARM işlemcisini simüle eden imaj dosyası da İnternetten indirilerek kurulabilir.

Kuracağınız imaj için en güncel işletim sistemi sürümünü ve deneme yapmak istediğiniz ekran özelliklerini seçerek imajınızı hızlıca oluşturabilirsiniz. Bir imaj dosyası türünü bir kere indirdikten sonra aynı tür birden çok sanal cihaz oluşturulabilir. Bunların her biri bilgisayarınızda ayrı ayrı tutulacaktır. Sanal cihazınız için tercihen 2GB RAM ayırmalar önerilir. Bu oran kendi bilgisayarınızda RAM'den alınacağı için sistem kaynaklarına göre emülatörün kullandığı RAM miktarı azaltılabilir. 512MB ve aşağısında RAM değerlerinde emülatör veya uygulamanız çalışmayabilir. Emülatörü oluşturduktan sonra AVD Manager ekranındaki "Actions" bölümünden "play ▶" butonu ile çalıştırabilirsiniz. Emülatör açıldığında normal bir telefon görünümü ile açılacaktır.



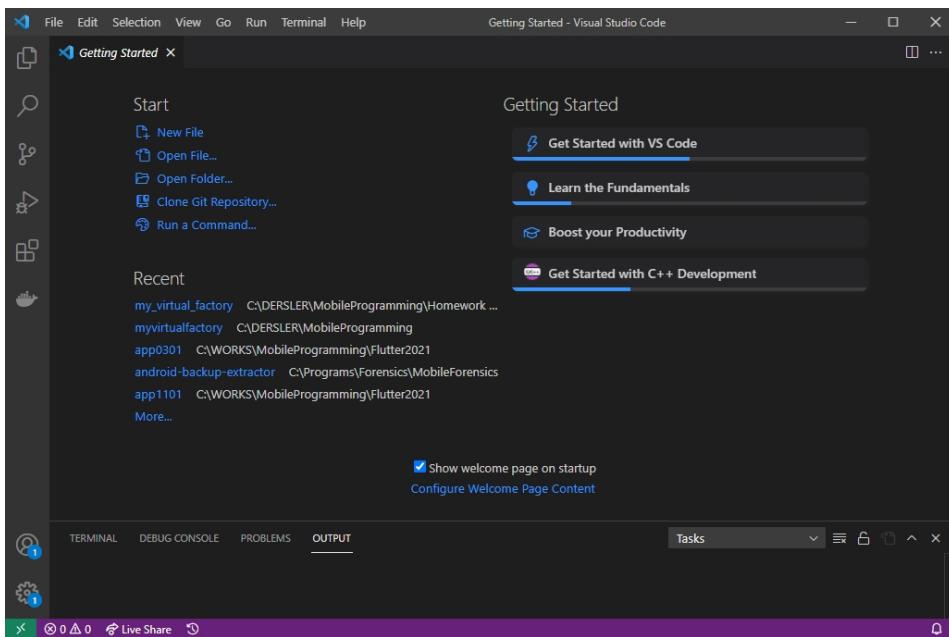
Bu aşamada Android Studio kurulumu ve gerekli ayarlamaları tamamlamış bulunuyoruz. VS Code ile kodlama yapacağımız için Android Studio içerisinde ek bir işlem yapmamız gerekmektedir. Eğer Android Studio içerisinde de Flutter projesi oluşturmak ve çalıştmak istiyorsanız Android Studio için Flutter ve Dart eklentilerini kurmanız gereklidir. Bunun için Android Studio giriş ekranından "Plugins" veya Android Studio proje perspektifindeki üst menüden "File -> Settings -> Plugins" seçenekleri takip edilerek açılan eklenti (Plugin) yönetim sayfasından "Marketplace" sekmesine geçilir. Burada "Flutter" ve "Dart" eklentilerinin Android Studio Plugin Marketplace üzerinden aratıp kurmanız gereklidir.



Visual Studio Code ve Flutter Eklentilerinin Kurulumu

Visual Studio Code (VS Code), Microsoft firmasının hafif kod geliştirme IDE'si olarak tüm işletim sistemlerinde çalışabilecek şekilde sunduğu ve tamamen Plugin temelli bir geliştirme ortamıdır. Birçok farklı

dilde uygulama geliştirilebilen VS Code içerisinde Flutter ile mobil uygulama geliştirmek de mümkündür.



Bunun için VS Code kurulumunu bilgisayarınıza yaptıktan sonra sol paneldeki Extentions (Ctrl + Shift + X) butonuna tıklayarak eklenti yönetim ekranını açabilirsiniz. Burada eklenti havuzundan (Extention Marketplace); “Dart, Flutter, Vscode-icons” eklentilerini kurmamız gerekmektedir. Sonrasında VSCode ile Flutter projeleri yazmaya hazırız. Eklenti kurulumlarından sonra VSCode IDE’sini yeniden başlatınız. Android Studio’da yeni Flutter projesi oluşturmak için proje şablonu ikonu ve bir sihirbaz olmasına karşın VSCode içerisinde bu özellik bulunmamaktadır. VSCode içerisinde yeni Flutter projelerini Flutter’ın kendi komut desteği ile Terminal penceresi üzerinden yapacağız. VSCode, kendi içerisinde Terminal penceresi sunarak bu süreçleri tek ekranda yapmamıza olanak vermektedir.

Flutter Doctor ile Son Kontrol

Flutter kurulumlarını gerçekleştirdiğiniz başta çalıştığımız “flutter doctor” komutu ile kurulumların başarılı şekilde tamamlanıp tamamlanmadığını kontrol ediniz.

```
PowerShell 7 (x64)
PS C:\Users\crane> flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[!] Flutter (Channel dev, 2.5.0-6.0.pre, on Microsoft Windows [Version 10.0.19043.1165], locale tr-TR)
[!] Android toolchain - develop for Android devices (Android SDK version 30.0.2)
[!] Chrome - develop for the web
[!] Visual Studio - develop for Windows (Visual Studio Enterprise 2019 16.10.4)
[!] Android Studio (version 2020.3)
[!] IntelliJ IDEA Community Edition (version 2021.1)
[!] Connected device (3 available)

• No issues found!
PS C:\Users\crane>
```

Eğer yukarıdaki maddelerde onaylanmamış bir tane göründüğünüz bunla ilgili kurulumlar bilgisayarınızda eksik veya Flutter SDK’sı düzgün ayarlanmamış demektir. Tek tek maddeleri ele alacak olursak;

Flutter (Channel dev, 2.5.0-6.0.pre, on Microsoft Windows [Version 10.0.19043.1165], locale tr-TR)

Kurulum yapılan bilgisayar ve kurulumun hangi Flutter dağılımı ile yapıldığını göstermektedir. Bu kurulumda Flutter Development Channel üzerinden 2.5.0 sürümünün son ön yüklemesinin Türkçe bir Windows 10 işletim sistemine yapıldığı (Flutter SDK’sının bu şekilde düzenlendiği) görülmektedir.

Android toolchain - develop for Android devices (Android SDK version 30.0.2)

Android Studio ve Android SDK’sının bilgisayarda kurulu olduğu ve Flutter’ın hangi Android SDK’sını baz aldığı göstermektedir. Bir Android Studio kurulumunda birden fazla Android SDK sürümü kurulabilir.

Aksi belirtilmemişçe Flutter en yüksek SDK ile çalışmak için kendini ayarlar. Burada da 30.0.2 Android SDK sürümünün kurulu bilgisayarda yüklü ve Flutter tarafından erişebilir olduğu görülmektedir.

Chrome - develop for the web

Flutter ile sadece Mobil platformlara değil Web ve Masaüstü için de uygulama geliştirebileceğimizi söylemişistik. Bilgisayarınızda “Google Chrome” uygulaması kurulu olmasını bu açıdan kontrol etmektedir. Eğer Web üzerinden uygulamanızın çalışmasını test etmeyecekseniz bu opsyonu doğrulamak zorunda değilsiniz.

Visual Studio- develop for Windows (Visual Studio Enterprise 2019 16.10.4)

Kurulum yapılan makine bir Windows işletim sisteme sahip ise geliştireceğiniz uygulamaların masaüstü sürümünün Windows işletim sisteminde çalıştırılabilmesi gerekmektedir. Flutter, Windows işletim sisteminde çalışmak için Visual Studio C++ kütüphanelerini kullanmaktadır. Bu kütüphanelerin bilgisayarınızda düzgün şekilde yüklenmesinin en kolay yolu Visual Studio IDE’sini bilgisayarınıza C++ Masaüstü geliştirme desteği ile kurmaktır. Visual Studio ile Visual Studio Code birbirinden farklı iki ayrı geliştirme ortamıdır. “Visual Studio” Microsoft firmasının uygulama geliştirme için amiral gemisi konumundadır. Community, Professional ve Enterprise olmak üzere üç farklı sürümle piyasaya sürülmektedir. Kurulum yapılan bilgisayarda Visual Studio Enterprise 2019 sürümü Flutter tarafından bulunmuştur. Eğer Windows işletim sisteminde çalışacak uygulamalar yapmak istiyorsanız (Dersimiz kapsamında emülatör çalıştırımaktan daha basit olduğu için çoğu örnekte tercih edilecektir.) Visual Studio İnternet sayfasından Community sürümünü ücretsiz olarak indirip kurabilirsiniz.

Android Studio (version 2020.3)

Bilgisayarınızda Android Studio’nun kurulu ve ayarlarının uygun olduğunu gösterir.

IntelliJ IDEA Community Edition (version 2021.1)

Bilgisayarınızda InterlliJ IDEA’nın (Android Studio ve VSCode haricinde kullanabileceğiniz diğer IDE) kurulu ve ayarlarının uygun olduğunu gösterir.

Connected device (3 available)

Bilgisayarınızda Flutter uygulamanızı çalıştırabileceğiniz bağlı ortam sayısını göstermektedir. Kurduğunuz Flutter Channel'a göre burada (0 available) ibaresini görebilirsiniz. Bu yanlış bir işlem yaptınız manasına gelmemektedir. Eğer Stable Channel üzerinden kurulum yaparsanız varsayılan olarak sadece mobil platformlara uygulama geliştirirsiniz. Yani Flutter derleme işleminde sadece Android ve iOS işletim sistemleri için derlenmiş uygulama üretecektir. Bu durumda Android / iOS cihazı veya Emülatörü sisteme bağlı ve çalışır durumda değilse uygun bir cihaz bulunamadı (0 available) görülmektedir. Daha önceki başlıkta anlatıldığı şekilde Android Emülatörü çalıştırıp emülatör açıkken “flutter doctor” komutunu çalıştırığınızda (1 available) ibaresini görmeniz gereklidir. Örnek kurulumumuzda dikkat ederseniz Developer Channel’dan bir Flutter yüklemesi mevcuttur. Bu nedenle web (chrome), web (firefox) ve Windows olmak üzere 3 farklı çalıştırılabilir cihaz “flutter doctor” tarafından bulunmuştur.

Eğer kurulumunuzun web ve Windows desteklerini içermesini istiyorsanız aktif kanalı değiştirerek ve konfigürasyonunu düzelterek bu ortamları desteklemeyi sağlayabilirsiniz. Aşağıdaki linklerde bu kurulum adımları anlatılmaktadır;

Masaüstü Desteği Kurulumu: <https://flutter.dev/desktop>

Web Desteği Kurulumu: <https://flutter.dev/docs/get-started/web>

Basitçe masaüstü desteginin Windows altında açılması için aşağıdaki komutları terminalden girmeniz yeterlidir. (Not: Visual Studio Community Edition’ı C++ Masaüstü geliştirme desteği ile kurdugunuza varsayıyoruz.)

```
flutter config --enable-windows-desktop
```

VSCode ile İlk Projemizi Oluşturalım

Flutter doctor ile kontrolleri yaptığınızda gerekli yazılımların bilgisayarınızda olduğunu teyit ettikten sonra artık ilk Flutter projemizi oluşturabiliriz. Dediğimiz gibi VS Code ile çalışacağımız için Flutter ile ilgili birtakım işlemleri terminal üzerinden gerçekleştireceğiz. VS Code terminal ekranına alıştığınızda bunun proje oluşturma sihirbazlarını kullanmaktan daha pratik olduğunu göreceksiniz. Adım adım ilk projemizi oluşturalım ve VS Code ekranına aşina olalım.

Terminalden Proje Oluşturma

İşletim sistemlerinde Terminal (Windows jargonunda Konsol) işletim sistemine komut göndermek için kullanıldığımız arayüzdür. En eski işletim sistemlerinden günümüze gelen işletim sisteminin vazgeçilmez parçasıdır. İşletim sisteminin özü kernel(çekirdek) kullanıcıların etkileştiği kısmı da Shell(kabuk) olarak ifade edilmektedir. Terminal de kabuk kısmının ana elemanıdır. İşletim sistemi üzerinde doğrudan yazılım geliştirme sürecine bu nedenle Kabuk Programlama (ShellScripting) adı takılmıştır. Biz de Flutter komutlarını kabuk üzerinden (Terminal vasıtasiyla) çağıracağız. Hatta bunu yapabilmek için kurulum adımda Flutter kurduğumuz klasörü işletim sistemi **Path** tanımlamasına kaydettik.

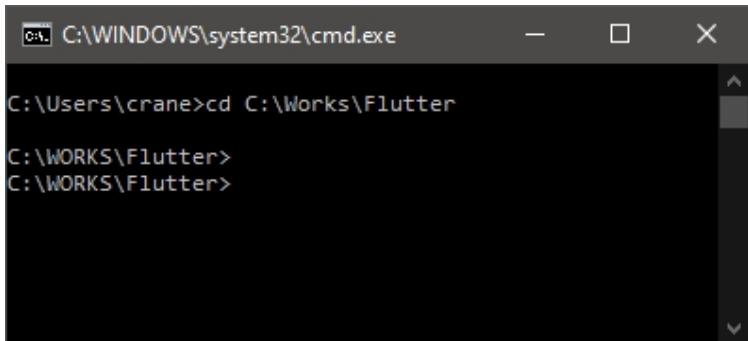
Öncelikle Terminal ekranını açmak için farklı seçeneklerden bahsedelim. Windows işletim sisteminde üç farklı şekilde terminal(konsol) kullanılabilir.

Komut İstemi (Command Prompt): Win+R ile açılan çalıştır ekranına “cmd” yazarak komut istemi açılabilir.

PowerShell: Win+R ile açılan çalıştır ekranına “powershell” yazarak veya Başlat’ta PowerShell ifadesini aratarak PowerShell uygulaması açılabilir. Power Shell artık GitHub üzerinden açık kaynak bir uygulama olarak geliştirilmektedir. Tüm işletim sistemlerinde kullanılabilir. Klasik terminal ekranlarına göre daha konforlu bir çalışma ortamı sunmaktadır. Ders içerisindeki ekran görüntüleri de genelde “PowerShell 7” üzerinden alınmıştır.

VSCode IDE’si İçerisinden: VSCode ile uygulama geliştirirken terminal erişimini daha çok IDE içerisindeki Terminal penceresinde gerçekleştireceğiz. Bunun için VS Code içerisinde alt bölmedeki Terminal sekmesine geçmeniz gerekmektedir. Eğer Terminal sekmesi görünmüyorsa üst menüden View -> Terminal veya **Ctrl+T** kısayol tuş kombinasyonu ile açabilirsiniz. VS Code kendi terminal penceresinde PowerShell kullanmaktadır.

Flutter projesini oluşturmak istediğimiz dizini terminal ekranında aktif dizin haline getirmeliyiz. Örneğin Projelerimizi C:\Works\Flutter dizini altında oluşturmak istiyorsak terminal ekranında bu dizini aktif dizin yapmamız gereklidir.



Flutter projesi oluşturmak için basitçe **create** komutunu kullanabiliriz.

```
flutter create myapp  
cd myapp
```

Terminal üzerinden yeni proje oluştururken ek özellikleri de tanımlanabilir. Bunun için Flutter sayfasından detaylı komut kümesine bakılabilir. Bizim için varsayılan ayarlarla bir proje oluşturmak şimdilik yeterlidir.

Daha detaylı bir proje oluşturma örneği olarak aşağıdaki örnek incelenebilir. Burada Android ve iOS tarafından kullanılacak native dillerin bilgileri verilmiştir.

```
flutter create --project-name myapp --org dev.flutter --android-language java --ios-language objc myapp
cd myapp
```

Proje oluşturmak için diğer alternatiflere <https://flutter.dev/docs/get-started/test-drive> sayfasından bakabilirsiniz.

Bölüm Özeti

Bu bölüm mobil platformlar, mobil programlama dilleri, tür açısından mobil programlama dünyası ve bu konu ile ilgili basit bir tarihe sunulmuştur. Mobil programlama özünde Web ve Masaüstü programlamaya benzese de bazı farklılıklar içermektedir ve günümüzde çok hızla gelişen bir alan haline gelmiştir. Hızlı değişimle beraber yazılım geliştirme süreci dallanmış ve maliyeti artmıştır. Bu durum cross-platform yazılım geliştirme ihtiyacını doğurmuştur. Flutter, Google firmasının desteklediği Dart programlama dilini temel alan açık kaynak, cross-platform bir uygulama geliştirme çerçevesi (framework) olarak karşımıza çıkmaktadır. Google Flutter için kurulum ve diğer içerikleri flutter.dev İnternet adresi üzerinden sağlanmaktadır. Flutter ile uygulama geliştirmek için öncelikle bilgisayarımızın buna hazır hale getirilmesi gerekmektedir. Bunun için Flutter SDK, JDK, Android Studio, Android SDK, Visual Studio Code ve IDE'lere ait Flutter ve Dart eklentilerinin kurulumları yapılmalıdır. Ayrıca masaüstünde veya web için çalışan Flutter uygulamaları geliştirmek için ek kurulumlar gerekmektedir.

Kaynakça

<https://flutter.dev/>

<https://developer.ibm.com/articles/choosing-the-best-programming-language-for-mobile-app-development/>

<https://www.statista.com/statistics/272698/global-market-share-held-by-mobile-operating-systems-since-2009/>

2. DART PROGRAMLAMA DİLİ TEMELLERİ

Giriş

Dart programlama dili C kökenli dillere çok benzemektedir. Hatta birçok özelliği Java, JavaScript, C#, C++ dillerinden alınmadır diyebiliriz. Bu bölümde Dart diline hızlı bir bakış atılacak ve diğer dillerden temel farklılıklarına değinilecektir. Algoritma geliştirme ve C kökenli bir program dili uygulama geliştirme yeteneğinizin var olduğu varsayımlı ile ders içeriği hazırlanmıştır. Ders içerisinde dil ile ilgili genel bilgi düzeyindeki bazı detaylar hızlı geçilecektir.

2.1. Basit Bir Dart Programı

```
// Fonksiyon Tanımlama
void printInteger(int aNumber) {
    print('The number is $aNumber.');// Konsola çıktı yaz
}
// Uygulamanın başlangıç noktası
void main() {
    var number = 42; // Değişken tanımlama ve ilk değer verme
    printInteger(number); // Fonksiyon çağrısı
}
```

Yukarıda basit bir Dart uygulaması görülmektedir. Tüm C kökenli dillerde olduğu gibi uygulama main adında bir fonksiyon ile başlamaktadır. Değişken tanımlama ve fonksiyon tanımlama şekli Java, JavaScript dili kullanımına benzemektedir. Yine bu dillerde bulunan blok yapısı ve (;) ile ifade sonlandırma Dart dilinde kullanılmaktadır.

2.2. Önemli Noktalar

Dart dilini yapısal olarak incelemeye başlamadan önce bazı önemli bilgileri sıralamakta fayda var.

Dart dilinde görebileceğiniz her şey bir objedir (object). Fonksiyonlar, değişkenler hatta null değeri bile bir obje olarak dilin içerisinde tanımlanır. Objeler de bir sınıfından (class) türetilir. Tüm objeler Object adındaki sınıfından türetilirler. Bu özelliği Dart'ı diğer dillere üstün kılan taraflarındandır. Özellikle fonksiyonların dahi obje olarak ele alınması dile esneklik kazandırır. Genel amaçlı dillerde bu kullanım pek görülmemektedir. Aynı işlevi yapmak için Delegates(C#) veya Function Pointers(C++) kullanılmaktadır.

Dart güçlü tip tanımlı (strongly typed) bir dildir. Buna karşın değişken tanımlamalarında tip belirtmemesizin tanımlamaya izin verir. Ama aslında burada değişkene ilk atanan değere göre türü bellidir.

Dart diline 2.12 sürümü itibariyle **Null Safety** özelliği getirilmiştir. Bu özellik sayesinde bir değişken siz null olabilir demediğiniz sürece null olamaz. Bir değişkenin null değer alabilmesini istiyorsanız tanımlama aşamasında veri tipinden sonra (?) soru işaretçi simbolü ekleyiniz. Örneğin int türü bir değişken null değeri alamazken int? türü bir değişken null değeri içerebilir.

Bir değişkenin herhangi bir türden veri saklamasını istiyorsanız veri tipiniz **Object** veya **Object?** (null-safety açılmışsa) olmalıdır.

Dart dili C# ve Java'da olduğu gibi jenerik(Generik) tanımlamalara izin vermektedir. Ör: List<int> , List<Object>

Dart dili global, yerel ve static değişken tanımlamalarına izin vermektedir.

Diğer nesneye yönelik dillerde kullanılan **public**, **protected**, **private** erişim belirteçleri kullanılmaz. Bunun yerine alt çizgi (_) karakterini bir değişkenin yerel (private) olup olmadığını göstermek için kullanır. Eğer alt çizgi (_) ile başlıyorsa private bir tanımlamadır.

Dart, C++ dilindeki gibi global seviyede fonksiyon tanımlamaya izin verir. Ana fonksiyonumuz olan **main** gibi. Sınıf içerisinde üye fonksiyon tanımlanabilir. Diğer dillerden farklı olarak fonksiyon içerisinde fonksiyon tanımlanmasına da izin vermektedir. Unutmayalım ki fonksiyonlar Dart dilinde birer obje olarak ele alınır.

2.3. Anahtar Kelimeler

Anahtar kelimeler (Keywords) bir programlama dilinde sembollerle verilemeyen ve dilin özünü oluşturan sentaksı ifade etmektedir. Dart dili, C kökenli dillerle aşağı yukarı aynı anahtar kelimeleri bünyesinde barındırmaktadır. Bu anahtar kelimelerin tam bir listesine;

<https://dart.dev/guides/language/language-tour#keywords>

adresinden bakılabilir. Anahtar kelimeleri tanımlayıcı (identifier – değişken, fonksiyon, sınıf vb. yapı adlandırması) olarak kullanmayınız. Eğer illa kullanım ihtiyacı varsa alt çizgi kullanımı ile bunu anahtar kelimelerden farklılaştırmamız gereklidir.

2.4. Değişkenler

Dart dilinde değişken (variable) tanımlaması JavaScript diline benzer şekilde aşağıdaki gibi yapılabilir.

```
var name = 'Can';
```

Burada **name** adında bir değişken tanımlaması yapılmıştır ve bu değişkene ilk değer olarak “Can” kelimesi aktarılmıştır. **name** değişkeni otomatik olarak **String** türü olarak ele alınacaktır. Bu açıdan Dart dili tür bağımlılığı olarak JavaScript’ten farklılık göstermektedir. JavaScript dilinde **name** değişkenine daha sonrasında tamsayı, ondalıklı sayı gibi farklı türde veri atadığımızda JavaScript tür dönüşümü yapar fakat Dart dilinde değişken ilk atamada oluşturulduğu tür ile devam etmek zorundadır. JavaScript’teki gibi bir yapı kullanmak istenilirse değişken tanımlaması **Object** türünde veya **dynamic** anahtar kelimesi ile (eğer gerekliyse) yapılabilir.

```
Object name = 'Can';
```

var ile yapılan tanımlamanın eşleniği aşağıdaki gibi String türünü kullanmaktadır. Eğer kodun okunaklılığını açısından türün yazılması gerekliyse ilk değişken tanımlamamız aşağıdaki gibi de yapılabilir. Yerel değişken tanımlamalarında Dart tasarım kılavuzunda belirtildiği üzere **var** ile tanımlama yapılmasını önermektedir.

```
String name = 'Can';
```

2.4.1. Varsayılan Değer

Bir değişken bellekte oluşturulacağı zaman içerisinde programcı tarafından ilk değeri verilerek oluşturulabilir veya verilmeden varsayılan değer (Default Value) ile bellekte yer bulur. Bu durum C/C++ programlama dilleri için ciddi bir sıkıntıdır. Çünkü performans gereği bu dillerde yerel değişkenlerde varsayılan değer atanmaz ve ilk değeri verilmemiş bir değişken bellekte kalan son veri izine göre bir değer alır. Bu durum dikkatsiz bir programcının uygulama içerisinde veri tutarsızlıklarını yapmasına (ilk değer atanmadan değişkendeki veriyi okuduğu için) neden olur. Bu nedenle tam nesne yönelimli programlama dillerinde ilk değeri atanmamış değişkenleri kullanamazsınız veya hepsinin bir varsayılan değeri (genelde bu veri tipi için sıfırın karşılığıdır. Integer:0, Float:0.0, Object:null) bulunur. Dart dilinde eğer null-safety özelliği aktif edilmemişse tüm değişkenlere (sayı türleri de dahil) varsayılan değer olarak **null** atanır. Çünkü daha önce de belirttiğimiz gibi Dart dilinde tüm değişkenler (temel türler dahil) **Object** türünden oluşturulmaktadır ve bir nesnedir.

Null-safety kullanımında null değeri alamayacak bir türün null alabilecek şekilde tanımlanmasını istiyorsak (? soru işaretü) sembolünü veri tipinden sonra kullanarak tanımlamam yapabiliriz. Örneğin;

```
int? satirSayisi;  
assert(satirSayisi == null);
```

Null-safety açılan bir sisteme bir değişken null değerine sahipse kullanılmadan önce içerisinde bir değer atanması gereklidir. Diğer bir deyişle bir değişken atama operatörünün daima önce sağ tarafında (Rvalue) yer almış olmalıdır.

```
int satirSayisi = 0;
```

Bir yerel değişken oluşturulurken ilk değeri verilmek zorunda değildir. Daha sonra verilebilir.

```
int satirSayisi;  
satirSayisi = 0;  
print(satirSayisi);
```

2.4.2. Late Değişkenler

Dart 2.12 sürümüyle değişken tanımlamasına late düzenleyicisi getirilmiştir. late anahtar kelimesi iki amaç için değişken tanımlamasında kullanılabilir;

Null olamayacak bir değişkenin tanımlamasından sonra ilk değerinin verileceğini belirtmek için.

Değeri sonradan gelecek (lazily loading) değişken oluşturmak için.

Dart içerisindeki akış kontrol analiz mekanizması genel olarak null içermeyecek bir değişkene yapılan atamanın kullanımından önce gerçekleşip gerçekleşmediğini tespit edip uygulama geliştiriciye

sunabilmektedir. Fakat bu analizin başarısız olduğu durumları bertaraf etmek için **late** düzenleyicisi kullanılır. Örneğin global değişken tanımlarında bu durum olmaktadır ve late kullanımı ile Dart'a bunun ilk değerinin kullanımından önce verileceğini söyleyebilirsiniz. Tabii olarak bunu demeniz ilk değer vermeyeceğiniz manasına gelmez. Eğer **late** kullanıp da ilk değeri vermezseniz bu sefer uygulamanız çalışma zamanı hatası üretecektir.

```
late String isim;  
void main() {  
  isim = 'Ahmet';  
  print(isim);  
}
```

late kullanımı eğer ilk değer atamasının külfeti fazlaysa bunun sadece gerekli olduğunda yapılması, gereksizse yapılmaması için bir performans artışı sağlamaktadır. late kullanarak gereksiz ilk yükleme işlemlerini bertaraf edebiliriz. Diğer bir kullanım alanı da yerel bir değişkene ilk değer atama işleminde ait olduğu obje referansını (this) kullanmaya ihtiyacımız olduğundadır.

```
// Bu programda _sicakligiOku() fonksiyonuna tek çağrı budur  
late String sicaklik = _sicakligiOku(); // Lazily initialized.
```

Yukarıdaki örnekte **late** kullanımı eğer **sicaklik** değişkeni kod içerisinde hiç erişilmeyecekse, **sicakligiOku()** fonksiyonunun hiç çağrılmamasını sağlar.

2.4.3. final ve const

Değişkenler uygulama geliştirirken bellek üzerindeki verilere erişmemiz için olmazsa olmazımızdır. Bu nedenle bir algoritmanın çalışma kapasitesi değişken kullanımındaki başarısına da bağlıdır. Bazen de bellekte sabit olarak bulunacak ve uygulamanın yaşamı boyunca değişmeyecek alanlara ihtiyaç duyulur. Bazen de bir defa veriyi giriş çıkış cihazlarından çekip sonra uygulama kapanana kadar aynen saklayacak alanlara ihtiyaç duyulur. Çoğu programlama dilinde **const** ve **final** anahtar kelimeleri bu ihtiyacı karşılamak için sabit (constant) veya veri(data) oluşturmada kullanılmaktadır.

Dart programlama dili diğer dillerden biraz farklı olarak, sadece aralarında ufak bir anlam farkıyla, bu iki anahtar kelimeyi de kullanmaktadır. Sabit tanımlamak için değişken tanımlamada kullandığımız var anahtar kelimesini final veya const ile değiştirmemiz yeterlidir.

Eğer sabitimiz değiştirilemez ve derleme zamanı(compile-time) tutacağı değer belli ise **const** ile oluşturulur. Sabitimiz bir defaya mahsus değer atamasına izin verecek ve ilk atanın değeri sabit olarak tutacak ise **final** kullanılmalıdır. Sınıf içerisinde ilk değeri dışarıdan gelmesi gereken ve sınıfın oluşturan objenin yaşam süresi boyunca değiştirilmeyecek tanımlar için final kullanılmaktadır. final'ın bu şekilde kullanımını Dart dilinde sınıf tanımlarken dışa bağımlı belli kuralları koymada uygulama geliştiricinin işine yaramaktadır. Sınıf içerisindeki yerel sabitleri bu nedenle **final**, global alanda bulunan sabitleri **const** tanımlamak gerekmektedir. Bir sınıf içerisinde **const** tanımlanması gerekliliği **static const** kullanmak gerekmektedir.

final tanımlamalarında veri türü ek bilgi olarak yazılabilir.

```
final name = 'Can'; // Veri tipi olmaksızın  
final String nickname = 'Crane'; // Veri tipi ile
```

final değeri ilk sefer atandıktan sonra kod içerisinde değiştirilemez.

```
name = 'Ahmet'; // Error: a final variable can only be set once.
```

const bir tanımlamaya başka bir const ifadenin işleme sokulmuş hali atanabilir (C++ makroları gibi).

```
const bar = 1000000; // Basınç değeri (dynes/cm2)  
const double atm = 1.01325 * bar; // Standart hava basıncı
```

2.5. Temel Veri Türleri

Her programlama dilinin özünde temel veri türleri (Build-In Types) mevcuttur. Kullanım ihtiyacına göre uygulama geliştiriciler kendi veri türlerini (class, struct enum kullanımı ile) tanımlayabilmektedir. Dilin belleği kullanabilme yeteneği içерdiği temel veri türü zenginliğine bağlıdır. Temel veri türlerinin çok olması karmaşıklığı, az olması ise verimi etkileyen bir faktördür denebilir. Dart dili aşağıdaki temel veri türlerini desteklemektedir;

Sayı Türleri (int, double)

String

Boolean (bool)

List (diğer bir ifadeyle diziler “arrays”)

Set

Map

Runes

Symbol

null değeri(Null)

Dart dili zengin bir temel veri tipi desteği sunmasının yanında kod içerisinde doğrudan veri (literal) olarak metin kullanımına ‘**bu bir metindir**’ ve boolean ‘**true**’ kullanımına izin vermektedir.

Tüm veri tipleri Object türünden miras alan nesne tabanlı bir yaklaşımla oluşturulduğu için değişken tanımlamasında her zaman ilk değer yüklemesi için yapıçı (**constructor**) fonksiyonunu çağrıma şansınız vardır. Misal Map türünü oluşturmak için kendi özel Map() fonksiyonu kullanılır.

Bu temel veri türlerinin dışında da Dart dilinin tanımlamalarda kullandığı özel amaçlı türler bulunmaktadır. Object, Future, Stream, Iterable, void, dynamic, Never bunlara örnek verilebilir.

2.5.1. Sayı Türleri

Dart dilinde sayısal değerler **int** ve **double** (kayan noktalı) olarak iki temel veri türü ile ifade edilmektedir. Her iki veri türü de **num** adındaki veri türünden türemektedir. Bu sayede veri türleri üzerinde **num** için tanımlanmış **abs()**, **ceil()**, **floor()** gibi temel matematiksel fonksiyonlar doğrudan kullanılabilmektedir. Daha geniş bir matematik fonksiyon koleksiyonu **dart:math** kütüphanesi altında yer almaktadır.

int, ondalık hanesi olmayan sayıları (tam sayıları) tanımlamada kullanılır. Kod içerisinde farklı şekilde tam sayı literalleri kullanılabilir.

```
var x = 1;          // Onluk sistemde tamsayı literalı
var hex = 0xDEADBEEF; //16'lık sistemde tamsayı literalı
var exponent = 8e5; //Bilimsel gösterimde tamsayı literalı (10 üzeri)
```

Ondalık ayracı (. Nokta) bulunan bir literal otomatik olarak double türü şeklinde ele alınır. Kodlamada ondalık ayracı daima İngilizce dilindeki kullanımı olan nokta işaretidir. Türkçede ondalık ayracı virgülür. (,) Ekranda yerleştirme ayarlarına göre ondalık sayılar virgülle veya noktayla ayrılmış şekilde gösterilseler de kodlamada daima ondalık ayracı olarak nokta kullanılır.

```
var y = 1.1;          //Klasik ondalıklı yazım.
var exponents = 1.42e5; //Bilimsel gösterimde ondalık sayı. 142000
```

num türünde de değişken tanımlaması yapılabilir. Bu şekilde tanımlanan bir değişken hem tamsayı hem de ondalık sayıları saklayabilir.

```
num x = 1; // x hem int hem double türü değer alabilir.  
x += 2.5;
```

Tamsayı türündeki bir literal ihtiyaç duyulduğunda otomatik olarak double türüne çevrilebilir. String'den sayı türlerine dönüşüm için `.parse()`, sayı türlerinden de String'e dönüşüm için `.toString()` metodları kullanılır. Dart dilinde tüm veri türleri obje olduğu için bu fonksiyonlar literallerde dahi çağrılabılır. Yani

```
String s = 1.toString();
```

Geçerli ve 1 sayısal değerini "1" String şecline dönüştürerek bellekte saklar.

2.5.2. String

Dart dilinde String (Karakter dizisi veya Katar olarak Türkçeleştirilmektedir.) bir obje türüdür. Tüm dünya dillerini desteklemesi için UTF-16 kodlama tablosunu kullanmaktadır. Yani her bir karakter bellekte 2 Byte uzunluğunda yer kaplamaktadır ve aynı anda metnin içinde Yunan, Latin, Arap, Çin alfabetesinden karakterler yer alabilir. Dart dilinde JavaScript kodlarında olduğu gibi literalleri hem tek tırnak ('') kullanarak hem de çift tırnak ("") kullanarak yazabiliriz. Eğer metin içerisinde tırnak işaretleri kullanımı gerekiyorsa kaçış sekansı denilen ters bölü (\) sembolünden sonra tırnak karakteri yazılabilir.

```
var s1 = Tek tırnaklı string sabiti;  
var s2 = "Çift tırnak ile oluşturulmuş string sabiti";  
var s3 = 'Tek tırnaklı yapıda tek tırnak için (\') kaçış sekansı';  
var s4 = "Çift tırnaklı tanımlamada ' doğrudan yazılır. Kaçış sekansı gerekmekz.";
```

Dart dilinin String tarafından en büyük artısı **String Interpolation** kullanımıdır. Bu yöntem sayesinde bir String içerişine değişkenler veya işlem sonuçları çok rahat bir şekilde \${expression} formatında eklenebilmektedir. Birçok dilde bunu yapmak için String'i + veya & sembollerile parçalara bölgerek yazmak gerekmektedir. Dart, String Interpolation'ı desteklemesinin yanında + simgesi ile String parçalarının birleştirilmeyi de destekler.

```
int i = 5;  
var s = 'burada yazılan karakterin içerisinde rakamla ${i+i} yazıyla on yazılır.';  
var s2 = s + 'ardına bu metni ekledim';
```

Çok satırda tek bir metin ifadeyi yazmak için 3 kere tekrarlayan tek veya çift tırnak kullanılır.

```
var s1 = ''''  
Çok satırlı bir metni  
Bu şekilde yazabilirsiniz.  
'''';  
var s2 = """Bu da benzer şekilde  
çok satıra ayrılmış bir metindir.""";
```

Bir String sabitinin önüne (r) ön eki eklenerek ham (raw) olarak oluşturulması sağlanabilir. Yani içerdeği ters bölü(\) sembolü kaçış sekansı için kullanılmaz ve doğrudan çıktı olarak görülür.

2.5.3. Boolean

Boolean türü mantıksal işlemler sonucu oluşan **true** veya **false** değeri alabilen bir türdür. Dart dilinde mantıksal değer alması gereken **if** kalımı veya **assert** fonksiyonu parametresi Boolean türünde olmak zorundadır. Dart dilindeki tür güvenliği C/C++ dillerinde olduğu gibi if kalımı içerisinde kontrol koşulu olarak tamsayı kabul etmez.

```
// Stringin boş olup olmadığı kontrolü
var fullName = '';
assert(fullName.isEmpty);
// Matematiksel kıyaslama
var hitPoints = 0;
assert(hitPoints <= 0);
// Bir objenin null olup olmadığı kontrolü
var unicorn;
assert(unicorn == null);
// Matematiksel bir işlemin belirsiz sonuç üretmesi kontrolü
var iMeantToDoThis = 0 / 0;
assert(iMeantToDoThis.isNaN);
```

assert metodunun kullanımında dikkat edilirse bool değer alabilmek için sonucu bool çıkacak bir mantıksal işlem (== gibi) veya bir üye metot çağrıları (**isEmpty**, **isNaN**) yapılmaktadır.

2.5.4. List

Bir çok programlama dilinde en fazla kullanılan veri yapısı diziler veya sıralı listelerdir. Çoğu nesneye yönelik dilde listeleri oluşturmak için ek koleksiyon sınıfları yazılmıştır. Dart dilinde listeler dilin özünde bir veri türü olarak ele alınmaktadır. Diğer taraftan Dart dilinde oluşturulan diziler (array) de bir liste olarak işlem görürler.

Dart dilinde liste tanımlaması JavaScript dilindeki dizi tanımlamasına benzemektedir.

```
var list = [1, 2, 3];
```

Burada oluşturduğumuz list adındaki değişken List<int> veri türündedir ve sonrasında içerisinde sadece tamsayı değerler saklayabiliriz. Eğer içerisinde Object türü genel veriler saklamak isteniyorsa var kullanımı yerine List<Object> şeklinde tanımlama yapılmalıdır.

Dart dilindeki enteresan bir kullanım liste sabitleri gibi virgülle ayrılan elemanlardan sonuncudan sonra da virgül koyulabilmesidir. Dart dili sentaks olarak her iki kullanıma da izin vermektedir. Elemanların bitiminden sonra virgül kullanımı listenin görsel olarak formatlanması kolaylaştırıldığı için de daha çok tercih edilir.

```
var list = [
  'Araba',
  'Gemi',
  'Uçak',
];
```

C kökenli tüm dillerde olduğu gibi Dart dilinde diziler(listeler) 0 (sıfır) ile başlayan bir indekslemeye sahiptir.

Eğer bir listenin derleme-zamanı bir sabit olmasını ve içerdeği hiçbir elemanın değişimmemesini istiyorsak dizi simbolü önüne **const** anahtar kelimesini eklememiz gereklidir.

```
var constantList = const [1, 2, 3];
// constantList[1] = 1; // Bu işlem hata oluşturacaktır.
```

Bir listedeki elemanları başka bir listeye kopyalanması gerektiğinde çoğu programlama dilinde bunun için bir döngü yazılması gerekmektedir. Dart dil içerisindeki **spread operator** (...) ve **null-aware spread operator** (...)? sayesinde bu işlemi çok kolay şekilde yapabilmektedir.

```
var list = [1, 2, 3];
var list2 = [0, ...list];
assert(list2.length == 4);
```

Yukarıdaki örnekte eğer list boş bir küme olsaydı, uygulama hata verip sonlanacaktır. Bu nedenle list2'nin oluşturulmasında eğer list boş gelecekse (...)? operatörünün kullanımı tercih edilmelidir.

```
var list;  
var list2 = [0, ...?list];  
assert(list2.length == 1);
```

2.5.5. Set

Dart dilinde Set sıralanmamış tekil elemanlardan oluşan bir kümeyi ifade eder. List ile arasındaki en önemli fark içeriği elemanların tekil olmasıdır. List'e benzer özellikleri vardır tanımlanmasında köşeli parantezler [] yerine küme parantezleri {} kullanılır.

```
var haftaici = ['pazartesi', 'salı', 'çarşamba', 'perşembe', 'cuma'];
```

Yukarıdaki ifadede haftaici kümesi Set<String> türünde oluşturulmuştur ve kümeye String harici bir türde değer atamaya çalışmak veri uyuşmazlığı hatası oluşturacaktır.

Eğer boş bir Set oluşturmak istenilirse sadece küme parantezleri {} kullanılarak tanımlama yapılabilir.

```
var isimler = <String>{};
```

Boş Set oluştururken değişken tanımlaması var şeklinde yapıldığında küme parantezleri önüne <String> tür belirleyicisinin yazılması zorunludur. Aksi halde Dart derleyicisi bu tanımlamayı Map<dynamic,dynamic> olarak algılayacaktır.

Set, boyut bilgisi için: **.length**, eleman eklemek için: **.add()**, başka bir kümeden toplu eleman eklemek için: **.addAll()** gibi üye özellikler ve fonksiyonlar barındırmaktadır.

```
var hafta = <String>{};  
hafta.add('cumartesi');  
hafta.addAll(haftaici);  
assert(hafta.length == 6); //hafta kümesinin 6 eleman içerip içermediği kontrolü
```

Set için List'de tanımlı olan üç nokta (...) ve (...?) operatörleri aynı şekilde kullanılmaktadır. **const** ifadesi sabit tanımlamak için yine küme parantezleri önüne koyulduğunda uygulama içerisinde değiştirilemeyecek bir küme üretilir.

2.5.6. Map

Map genel olarak verileri anahtar(**key**)-değer(**value**) çiftleri şeklinde saklamak için kullanılan veri yapılarıdır. Buradaki **key** ve **value** değerleri herhangi bir türden (**object**) olabilir. Map içerisinde **key** tekil olmak zorundadır ama **value** değeri birden çok kez tekrar edebilir.

```
var tuslar = {  
    // Key:      Value  
    'enter': 'Başlat',  
    'esc': 'Bitir',  
    'sol': 'Sola Hareket Et'  
};  
var kodlar = {  
    2: 'Başla',  
    10: 'Bitir',  
    18: 'Sağ',  
};
```

Yukarıdaki örneklerde tuslar adındaki Map <String , String> formundadır. Key hanesi String bir ifade (ör: 'enter') Value hanesi yine String bir ifade olarak şekillenmiştir (ör: 'Başlat'). kodlar adındaki Map'de Key hanesi tamsayı türde, Value hanesi de String türde veri saklamaktadır.

Map yapısı da boş oluşturulup içerisinde değerler sonradan yazılabilir.

```
var tuslar = Map<String, String>();
tuslar ['enter'] = 'Başlat';
tuslar ['esc'] = 'Bitir';
tuslar ['sol'] = 'Sola Hareket Et';
var kodlar = Map<int, String>();
kodlar [2] = 'Başla';
kodlar [10] = 'Bitir';
kodlar [18] = 'Sağ';
```

Boş bir Map oluşturmak için parantezler () kullanıldığını görmekteyiz. Aslında bu bir fonksiyon çağrısidir ve Map için yapıçı (constructor) fonksiyonu çağrıyoruz manasına gelmektedir. Dart dilinde her veri türünün Object'ten türetildiğini söylemişik. Aslında buradaki Map türü değişken oluşturma satırlarımız

```
var tuslar = new Map<String, String>();
```

yazımıyla eşdeğerdir. C# veya Java kullananlar bu yazım şeklini bekleyebilirler. Dart dilinde **new** anahtar kelimesinin yazımı isteğe bağlıdır. Basitlik açısından yazmamayı tercih etmekteyiz. C++ dilinden gelenler burada dikkatli olmalıdır. C++ dilinde new anahtar kelimesinin yazılması ile yazılmaması farklı manalarda kullanılmaktadır. Dart dilinde ise new yazmasak da C# ve Java'daki gibi new anahtar kelimesinin olduğu şekilde işlem görmektedir.

Map'deki value bölgesinde saklanan değeri okumak için dizilerde olduğu gibi index operatörleri [] içerisinde verinin index değeri yerine key alanında saklanan bilgi girilir. Bu açıdan baktığınızda Map kullanımını index değerleri 0,1,2,3 yerine sizin vereceğiniz key değerleri ile oluşan özel bir List gibi düşünebilirsiniz.

```
print ( kodlar [10] ); //Ekrana 'Bitir' ifadesini yazar;
```

const ile sabit Map tanımlanabilir. (...) ve (...) işlemleri Map üzerinde de uygulanabilmektedir.

2.5.7. Runes

Dart dilinde runes bir metin içerisinde Unicode karakterlerin eklenmesini sağlar. Bunun için characters adındaki paket kullanılır. Kullanılan Unicode kodlama tablosu “Unicode Code Charts” sayfasından incelenebilir.

Unicode dünyada kullanılan tüm yazı sistemlerindeki karakter, sayı, semboller tekil bir sayısal değerle ifade etmektedir. Dart dili UTF-16 kodlama sistemini kullandığı için Unicode değerlerini metnin içerisinde kullanmak için özel bir yazım şekli olmalıdır. Unicode karakterlerinin yazdırılması için genel kullanılan yöntem \uXXXXX formatında kaçış sekansını takip eden u karakteri ve ardından gelen 4 adet 16'lık tabandaki rakamdan oluşur. Eğer dörtten farklı miktarda rakam kullanılacaksa rakamlar küme parantezine alınarak \u{XXXXX} formatında yazılırlar.

Örneğin yazı içerisinde kalp karakteri ♥ eklemek için \u2665 kodu karakterin görülmesi istenilen noktada metne eklenir.

2.5.8. Symbol

Symbol, Dart dilinde tanımlanan bir operatörü veya identifier'ı oluşturan bir obje türüdür. Bir tanımlamanın # karakteri ile başlatılması ile oluşturulur. Normal kodlama içerisinde kullanım ihtiyacımız olmayacağı.

2.6. Fonksiyonlar

Fonksiyon konusu Dart dilini diğer dillerden farklılaştıran önemli ayrıntılardan biridir. Fonksiyon tanımlama şekli JavaScript dilindeki kullanıma benzer. Bunun yanında Dart dilinde fonksiyonlar (functions) da her seyde olduğu gibi birer objedir. Bir obje olmaları sayesinde fonksiyonlar değişkenleri kullanıldığı yerlerde kullanılabilmektedir. Yani bir fonksiyon başka bir fonksiyonun parametresi, geri dönüş değeri olabilir. Fonksiyonlar iç içe tanımlanabilir, atama operatörü ile başka değişkenlere atanabilirler.

Bir fonksiyon basit şekilde aşağıdaki gibi tanımlanabilmektedir.

```
bool func1(int index) {  
    return myList[index] != null;  
}
```

Bu tanımlama şekli tüm programlama dillerindeki genel yapıya benzemektedir. Fonksiyonun geri dönüş türü adı ve parametre listesi verilerek tanımlaması yapılır. Parametre listesinde her bir parametre için veri türü ve parametre adı belirtilir. Dart dilinde fonksiyon tanımlamasında pek çok alan çıkarılabilir. Dart dilinde fonksiyonun geri dönüş veri türünü ve parametrelerin türünü belirtmek zorunlu değildir. Dart fonksiyon içerisindeki kullanım şeklinde parametre ve geri dönüş türünü kendisi keşfetebilir. Yukarıdaki fonksiyon aşağıdaki gibi de tanımlanabilmektedir.

```
func1(index) {  
    return myList[index] != null;  
}
```

Tek satırlık bir içeriği olan fonksiyon yazılıyorsa fonksiyon gövdesi oluşturmak yerine => şeklinde bir ok operatörü kullanılabilir. Bu operatör kodlama yaparken gereksiz satır atlamaları olmasını engeller ve kod daha derli toplu durur.

```
bool func1(int index) => return myList[index] != null;
```

Kısa fonksiyon yazarken => operatöründen sonra fonksiyon gövdesine tek bir ifade yazılabilir ama if bloğu şeklinde bir statement yazılmaz. Eğer koşullu bir yazım gerekiyorsa ?: yapısı kullanılabilir.

2.6.1. Fonksiyon Parametreleri

Dart programlama dilinde fonksiyon parametreleri istenilen sayıda opsionlu veya zorunlu olarak tanımlanabilir ve sıralaması değişimle isimlendirilmiş parametreler kullanılabilir.

Özellikle Flutter çerçevesini(framework) kullanırken isimlendirilmiş parametrelerin kullanımı kod yazmayı kolaylaştırıyor ve kodlamaya belli bir düzen getiriyor.

İsimlendirilmiş Parametreler:

Bir fonksiyonu çağrıduğumuzda klasik fonksiyon tanımlamasında yaşadığımız en büyük sıkıntılardan biri fonksiyon tanımlamasında verilen parametrenin sırasını hatırlamaktır. Genel programlama mantığında fonksiyon çağrılarında parametreler tanımlandığı sırada fonksiyon çağrısına beslenmelidir. Daha dramatik bir durum da fonksiyonun ilerde güncellenmesi gerekiyorsa parametrelerden birinin yerini değiştirmek veya araya yeni parametre eklemektir. Bu durum da fonksiyonun çağrıldığı tüm kod bloğu gözden geçirilmeli ve bütün çağrılar sıralamayı kaydırmadan yeni yapıya göre kodlanmalıdır. İşte isimlendirilmiş parametre (named parameter) kullanımı bizi bu zorluktan kurtarır. Fonksiyon tanımlanırken verilen parametre adı fonksiyon çağrısında da **parametre_adi: değer** formatında parametreye verilecek değerin önüne yazılır. Benzer yaklaşım şu an gelişmiş nesneye yönelik programlama dillerinin hepsinde bulunmaktadır veya eklenmesi planlanmıştır.

```
gorunum(kalin: true, gizli: false);
```

Buradaki fonksiyon çağrısı örneği için aşağıdaki gibi bir fonksiyon tanımı söz konusudur.

```
void gorunum ({bool? kalın, bool? gizli }) {...}
```

İsimlendirilmiş parametre kullanımı için klasik fonksiyon tanımlamasında isimlendirilecek parametreleri küme parantezleri {} içerisine yazmanız gerekmektedir. İsimlendirilmiş parametreler ad verilerek değer gönderildiği için istenilen sırada fonksiyon çağrısında beslenebilirler. İsimlendirilmiş parametreler aynı zamanda opsionlu parametrelerdir. Yani fonksiyon çağrısında beslenmese de fonksiyon çağrı yapılabılır. Bu açıdan isimlendirilmiş bir parametrenin verilmemesi durumunda **null** değeri almaması için varsayılan

değer (**default value**) tanımlaması yapılmasında fayda vardır. İsimlendirilmiş bir parametrenin beslenmesini zorunlu kılmak için **required** anahtar kelimesi önüne eklenir.

```
const Scrollbar({Key? key, required Widget child})
```

Flutter çerçevesindeki bir yapıçı fonksiyonda görüleceği üzere **child** adındaki parametrenin beslenmesi zorunludur. Burada hem **key** hem de **child** parametreleri isimlendirilmiş parametredir.

Opsiyonlu Parametreler:

Eğer isimlendirilmiş parametre tanımlamayı sadece opsiyonlu parametre oluşturmak isteniyorsa ilgili parametre kümesi köşeli parantezler [] arasında yazılır. Diğer programlama dillerinde opsiyonlu bir parametre oluşturmak için varsayılan değer verilmesi gerekmektedir. Dart dili bu kullanımla varsayılan değer verilmeden de opsiyonlu parametre tanımlamaya izin verir. Opsiyonlu bir parametre fonksiyon çağrısında eğer beslenmezse **null** değeri alacağını unutmamak gereklidir. Fonksiyon içerisindeki akış buna göre kurgulanmalıdır.

```
String say(String from, String msg, [String? device]) {
  var result = '$from says $msg';
  if (device != null) {
    result = '$result with a $device';
  }
  return result;
}
```

Yukarıdaki fonksiyonda **device** parametresi null alabilen bir String türü olarak ve opsiyonlu tanımlanmıştır. Bu fonksiyona iki farklı şekilde (2 parametreli ve 3 parametreli) çağrı yapılabılır.

```
//iki parametreli çağrı ve çıktı kontrolü
assert(say('Bob', 'Howdy') == 'Bob says Howdy');
//üç parametreli çağrı ve çıktı kontrolü
assert(say('Bob', 'Howdy', 'smoke signal') ==
  'Bob says Howdy with a smoke signal');
```

Varsayılan Parametre Değeri (Default Value):

İsimlendirilmiş parametreler ve opsiyonlu parametreler için parametre adından sonra eşittir (=) operatörü kullanarak varsayılan değer tanımlaması yapabiliriz. Varsayılan olarak atayacağımız değerin derleme zamanı sabiti (compile-time constant) olması gerekmektedir. Eğer varsayılan değer tanımlanmazsa ve fonksiyon çağrısında bu parametre beslenmezse varsayılan olarak **null** değeri atanmaktadır.

```
void gorunum ({bool kalin = false, bool gizli = false}) {...}
// kalin true; gizli ise false olacaktır.
gorunum (kalin: true);
```

Yukarıda isimlendirilmiş parametrede varsayılan değer kullanımı görülmektedir.

```
String say(String from, String msg,
  [String device = 'carrier pigeon']) {
  var result = '$from says $msg with a $device';
  return result;
}
assert(say('Bob', 'Howdy') ==
  'Bob says Howdy with a carrier pigeon');
```

Bu örnekte de opsiyonlu **device** parametresi için varsayılan değer tanımlaması yapılmıştır. Önceki fonksiyon akışı **device** değeri **null** gelemeyeceği için yeniden düzenlenmiştir.

2.6.2. main() Fonksiyonu

Her Dart uygulaması C++ dilinde olduğu gibi en üst seviyede bir main() fonksiyonu olmasını gerektirir. main() fonksiyonunun geri dönüş türü void türüdür. Yani geriye bir değer döndürmez. Parametre olarak List<String> veya boş olarak kullanılır.

```
void main() {
  print('Hello, World!');
}
```

Yukarıdaki örnek genellikle Flutter uygulamalarında kullanacağımız parametresi olmayan main fonksiyonu örneğidir. Aşağıdaki örnekte ise konsoldan beslenen argüman listesini alan ve test eden bir main fonksiyonu vardır.

```
// Run the app like this: dart args.dart 1 test
void main(List<String> arguments) {
  print(arguments);
  assert(arguments.length == 2);
  assert(int.parse(arguments[0]) == 1);
  assert(arguments[1] == 'test');
}
```

2.6.3. Anonim Fonksiyonlar

Fonksiyon tanımlamasında geri dönüş türü parametre türlerini yazmaya bilmemizi söylemiştık. Dart eğer fonksiyon sadece yazıldığı yerde kullanılacaksa yani başka bir yerde adıyla bir çağrı yapılmayacaksa isimsiz fonksiyon tanımlamamıza izin vermektedir. JavaScript kodlaması yapanlar için aşağına bir uygulama olmasına karşın diğer dillerde pek alışık olmadığımız aşağıdaki örnek Dart dilinde geçerli bir fonksiyon tanımlamasıdır.

```
(msg) => '! ${msg} !';
```

Bu örnek tek ifade içeren bir fonksiyon tanımlamasıdır. Tek ifade içeriği için fonksiyon gövdesinde küme parantezleri {} yerine ok => operatörü kullanılmıştır. Fonksiyon String msg adında bir parametre alır, aldığı String'in başına ve sonuna ünlem karakteri ekler ve yine bir String veri döndürür. Fonksiyonun tüm bu tanımlaması (msg) kısmında verilmektedir. Evet görsel olarak daha önceki fonksiyon tanımlamalarına hiç benzememektedir. Eğer Dart dilinin geri dönüş değeri istemediği, parametre türü istemediği ve anonim (isimsiz) fonksiyon yazmaya izin verdiği düşünürsek bu satır anlamlı bir satırdır. Uzun şekilde yukarıdaki ifade şuna denktir;

```
String fonksiyonAdi(String msg)
{
  '! ${msg} !';
}
```

2.6.4. Fonksiyonların Diğer Özellikleri

Fonksiyonlar Dart dili içerisinde birer objedirler. Bu sayede diğer objeler gibi değişkenlere aktarılabilirler.

```
var yükselt = (msg) => '!!! ${msg.toUpperCase()} !!!';
assert(yükselet('hello') == '!!! HELLO !!!');
```

Yukarıdaki örnekte yükselt adındaki değişken aslında bir fonksiyon ifade etmektedir. yükselt tanımlayıcısı bu manada C++ dilindeki fonksiyon göstericisi veya C# dilindeki delege özelliğindedir. Bu diller fonksiyonları işaret eden değişkenler için özel bir sentaks kullanırlar. Fakat Dart dilinde özel bir simbol eklenmez. Klasik değişken tanımlaması gibi yapılır. Buradaki asıl soru bir fonksiyon saklayan değişkendeki fonksiyonun kodları ne zaman çalışır? Fonksiyon gövdesinin işlenmesi için fonksiyonun çağrılmaması gereklidir. Fonksiyon gösteren değişken adının yanına parantezler ve varsa parametreleri tanımlanarak fonksiyon çağrıması yapılabilir. Bu basit kod sentaksi karmaşık işlevlerde iç içe geçmiş fonksiyon yapılarını yazmayı ve yönetmeyi kolaylaştırmaktadır.

Fonksiyonlarda varsayılan geri dönüş değeri (Return Value) **null** değeridir. Bunun nedenini basitçe bir fonksiyonun aynı zamanda bir obje olmasından çıkarabiliriz. Her şeyi nesne tabanlı olarak işleyen Dart dili, veri olmayan noktada **null** değeri kullanmayı uygun görmüştür.

```
foo() {}  
assert(foo() == null);
```

Bu örnekte foo adındaki fonksiyon sadece boş bir gövdeye sahiptir ve assert işlevinde yapılan kontrol fonksiyon null olacağı için geçerli sayılacaktır.

2.6.5. Fonksiyonlar ve Yaşam Alanı

Dart dili iç içe fonksiyon tanımlamasına izin vermektedir. Bu durum birçok nesneye yönelik programlama dilinin yaklaşımından farklıdır. Örneğin C# ve Java dillerinde bir fonksiyon ancak bir sınıfın içerisinde tanımlanabilir. C++ dilinde ise durum biraz daha farklıdır ve global yani hiçbir sınıfa(class) ait olmayan fonksiyon tanımlamamıza imkan vermektedir. Yine C++ dilinde küme parantezlerini {} kod içerisinde istedigimiz yerde kullanarak bir yaşam alanı (Scope) oluşturabilmekteyiz. Dart dilinde C++ dilindeki bu zenginlik alınmıştır. Global fonksiyon oluşturabiliriz ve bunun üstüne fonksiyonları iç içe de tanımlayabiliriz. Bu durum bize hem bir karmaşıklık hem de bir üstünlük sağlayacaktır. Karmaşıklık kodun ileriye yönelik okunabilirliğinin azalması olarak söyleyebiliriz. Avantajımız ise dış fonksiyonda tanımlı bir değişkenin daha iç fonksiyonlarda erişilebilmesi ve kullanılabilmesidir. Bunun tersi ise mümkün değildir. Yani içerisinde tanımlanmış bir değişken veya fonksiyon bulunduğu küme parantezleri dışında {} (scope) kullanılamazlar.

```
var ustSeviye = 1;  
void main() {  
    var mainIcinde = 2;  
    void birinciFonksiyon() {  
        var fonksiyonIcinde = 3;  
        void icFonksiyon() {  
            var icFonksiyonIcinde = 4;  
            ustSeviye = 5;  
            mainIcinde = 6;  
            fonksiyonIcinde = 7;  
            icFonksiyonIcinde = 8;  
        }  
    }  
}
```

Yukarıdaki örnekte ustSeviye adındaki değişken global olarak tanımlanmıştır ve uygulamanın her yerinden erişilebilir. mainIcinde değişkeni main fonksiyonu içerisinde tanımlanmıştır ve main fonksiyonunun başlama küme parantezi {} ile bitimini belirten son küme parantezi arasında kullanılabilir. Benzer şekilde fonksiyonIcinde ve icFonksiyonIcinde değişkenleri kendi tanımlandıkları küme parantezleri arasında kullanılabilirmektedir. Hal böyleyken biz tüm değişkenlere icFonksiyon'un gövdesinden erişebiliriz. Bu bizi bazı durumlarda veri aktarımı için fonksiyonlara sürekli parametre besleme gereksiniminden kurtarır. Yine de bu tarz kod yazarken dikkatli olunması gereklidir. Gördüğünüz gibi iç içe tanımlanmış yaşam alanları (scope) kodun ilerleyen dönemlerde okunmasını zorlaştıracaktır.

Şimdi fonksiyonların obje olması ve iç içe tanımlanabilmesinin farklı bir uygulamasına bakalım. Aşağıdaki örnekte bir fonksiyonun çağrılmama anında farklılaşmasını sağlayan tanımlayıcı bir fonksiyon kullanımı gösterilmektedir. (Not: Sınıflar seviyesinde benzer kullanımı diğer dillerde de yapabiliriz. Dart burada salt fonksiyon seviyesinde bu işlemi sağlayarak bize kolaylık sunmaktadır.)

```

Function duzenleyici(int baslama) {
  return (int i) => baslama + i;
}
void main() {
  var topla2 = duzenleyici (2);
  var topla4 = duzenleyici (4);
  print(topla2(3)); // 5
  print(topla4(3)); // 7
}

```

Örneğimizde düzenleyici fonksiyonu geri dönüş değeri olarak bir fonksiyon üretmektedir. Üretilen bu fonksiyonun gövdesi için ise aldığı parametreyi başlama noktası olarak bir toplama işlemine vermektedir. İçerde üretilen fonksiyon çağrıları yapıldığı noktada düzenleyici fonksiyonun verdiği başlama değerini referans alarak farklı şekilde çalışmaktadır. Bu kullanımın örnek projelerde sonradan davranış sağlamak için uygulandığını göreceğiz.

2.7. Operatörler

Dart dilindeki operatörler C kökenli dillerdeki operatörlerle aşağı yukarı aynıdır ve aynı amaç için kullanılırlar. Bunların tam bir listesine Dart.dev sayfasından incelenebilir. Burada diğer dillere göre farklı özellikteki bazı operatörlerini inceleyeceğiz.

2.7.1 Tür Kontrol Operatörleri

Dart dilinde her türlü tanımlamanın bir obje olduğunu hatırlımızda bulundurursak, en önemli işlevlerden birinin bir objenin türünü test etme gereksinimi olduğunu rahatlıkla söyleyebiliriz. Özellikle kod içerisinde dinamik işlevlerimiz varsa gelen türün ne olduğunu bilmemiz ve buna göre işlem yapmamız gerekebilir. Bu işlem için üç temel operatör; as, is, is! Operatörleridir.

Operatör Anlamı

- | | |
|-----|---|
| as | Tür dönüşümü (Aynı zamanda bir kütüphanenin özel isim verilmesinde de kullanılmaktadır) |
| is | Tür kontrolü; Eğer obje belirtilen türde ise true sonucu üretir |
| is! | is operatörünün tersi sonuç üretir. Eğer obje belirtilen türde ise false sonucu üretir. |

as operatörünü özellikle miras alma yöntemi kullanıldığında üst sınıf türünde bir değişkenin alt sınıf özelliklerine erişim için kullanırız. Aşağıdaki örnekte olduğu gibi employee adındaki değişken Person sınıfından miras alınmış bir örneği iken Person sınıfında tanımlı bir özelliği olan firstName alanına erişim için as operatörü sadece bu işlemde geçerli olmak üzere tür dönüşümü için kullanılır. employee değişkeninin asıl türü değişmez.

```
(employee as Person).firstName = 'Bob';
```

Aynı kodu is operatörü kullanarak aşağıdaki şekilde de oluşturabiliriz. Aşağıdaki kullanımda employee değişkeninin türü kontrol edildikten sonra işlem yapılmaktadır.

```

if (employee is Person) {
  // Tür kontrolü
  employee.firstName = 'Bob';
}

```

2.7.2. Null Kontrollü Atama Operatörü ??= ve Koşul Operatörü ??

Tüm dillerde atama operatörü (=) sağ taraftaki değeri (Right Value – Rvalue), sol tarafta (Left Value - Lvalue) belirtilen bellek bölgесine (Kod içerisinde bellek bölgесini değişkenlerle ifade ettiğimizi akılda çıkarmayınız. Yani diğer bir deyişle Lvalue olarak genellikle bir değişken veya değişkene eş değer gelecek bir ifade yer almmalıdır.) aktarmayı sağlamaktadır. Çoğu zaman bir değişkenin null değeri olup olmadığını kontrol etmek ve bu durumda içeresine değer aktarmak isteyebilirsiniz. Dart dilinde bunun için (??=) operatörü tanımlanmıştır.

```
x = value;  
y ??= value;
```

Yukarıdaki örnekte x değişkenine value değeri her zaman atanırken y değişkeni sadece null ise içerisinde value değeri aktarılır. Aksi halde y değişkenine atama işlemi yapılmaz.

?? operatörünü de bir değişkene aktarmak istediğimiz değer null olursa ikinci bir seçenek vermek için kullanabiliriz.

```
y = value ?? 10;
```

Yukarıdaki örnekte eğer value null değil ise y değişkenine değeri aktarılır. Eğer value değişkeni null ise bu durumda y'ye 10 değeri aktarılır. Bu operatör kod içerisinde uzun if-else kalıpları yazmak yerine kodun daha kısa ve kolay takip edilmesini sağlayacaktır.

2.7.3. Kademeli İşlemler .. , ?..

Bir obje üzerinde birden fazla üye değişkene değer atama veya üye fonksiyonlarını çağrıma ihtiyacı olmaktadır. Bu durumda obje adını tekrarlı şekilde yazmak yerine kademeli (cascading) işlem operatörleri kullanılabilir.

```
var paint = Paint()  
  ..color = Colors.black  
  ..strokeCap = StrokeCap.round  
  ..strokeWidth = 5.0;
```

Yukarıdaki örnekte paint değişkenine önce Paint() yapıcı fonksiyonu çağrılarak Paint sınıfında bir obje aktarılır ve sonrasında bu obje üzerinde üye fonksiyonlar çağrılmaktadır. Burada dikkat edilmesi gereken kademeli işlem operatörü (..) çağrımadan önceki satırda değişkenin muhakkak bir obje referansı ile doldurulmuş olması gereklidir. Genelde bunu yapıcı (constructor) fonksiyonlarını çağırarak gerçekleştiririz.

Eğer kademeli işlemlerin yapılacağı değerin null olma ihtimali varsa, kademeli işlemleri null bir obje referansı üzerinde işlemeyerek atlamak için ilk kademede ((..)) operatörü kullanabiliriz. Bu sayede null bir obje referansı üzerinden işlem yapılmayacak ve çalışma zamanı hatası üretilmeyecektir. Fakat unutulmamalıdır ki bu durumda ilgili kodlar da çalıştırılmayacaktır. Aşağıda buna bir örnek verilmektedir.

```
querySelector('#confirm') // Sayfa içerisinde bir obje referansı alalım.  
  ..text = 'Confirm' // Bu referans üzerinden üye özelliklere eriş (null değilse)  
  ..classes.add('important')  
  ..onClick.listen((e) => window.alert('Confirmed!'));
```

2.8. Akış Kontrolü

Dart dilinde C türevi dillerdeki akış kontrol yapıları aynı şekilde kullanılmaktadır. **if-else**, **switch-case** ve **assert** akış kontrolü için kullanılır. **for**, **while**, **do-while** döngü yapıları kullanılmaktadır. **break**, **continue** anahtar kelimeleri de döngü kaçış noktaları olarak aynı şekilde kullanılmaktadır.

Dart dili akış kontrolü için Boolean değeri koşul hanesinde beklemektedir. Yani bir if yapısında koşul olarak değer true veya false dışında bir şekilde yorumlanamaz. 0 veya 1 gibi bir tamsayı değeri bir koşu ifadesi olamamaktadır. (C++'da 0 tamsayısı da false olarak kabul edilmektedir. Bu dilden gelen uygulama geliştiriciler için bu farka dikkat edilmelidir.)

Üzerindeki verilerin iteratif olarak taranabileceği veri türlerinde (List, Set gibi) her bir elemanı tek tek ele alan **for-in** iteratif döngü yapısı kullanılabilmektedir.

```
for (var aday in adaylar) {  
  aday.gorusme();  
}
```

adımlar değişkeni bir List olmak üzere içeriği veri türündeki her bir eleman sırasıyla adı değişkenine aktararak tüm elemanlar üzerinde çalışacak bir döngüye sahip oluruz. Bu kullanım klasik for döngüsü yazdığımızda her elemana indeks değeri vererek erişme külfetinden bizi kurtarır. for-in döngüsü ile yapılan işlemi iteratif veri türleri üzerinde çalıştırabileceğiniz forEach() adında bir metod da tanımlıdır. Bu metoda vereceğiniz bir fonksiyon veri yapısı üzerindeki her bir eleman için çağrılacaktır.

2.8. Hata Yönetimi - Exception

Dart dili de diğer dillerdeki **try-catch-finally** hata yakalama ve yönetim kod yapısını kullanmaktadır. Dart dilinde obje türünde **Exception** ve **Error** adındaki iki türden oluşturacağınız kendi hata türlerinizi de fırlatabilirsiniz. Hata fırlatmak için **throw** anahtar kelimesi kullanılır. Hata yakalamak istenildiğinde eğer hatanın türü belirtilecekse **on catch** anahtar kelimeleri kullanılır. Tüm hata türlerini yakalamak için sadece **catch** ifadesi kullanılır. Try-Catch yapılarında genel kural olarak **on catch** için yazılacak bloklarda sıralama önemlidir ve dar kapsamdan geniş kapsamlıya göre yazılmalıdır.

Yakalanan bir hatanın üst seviye bir kodda tekrar işlenmesini istiyorsak **catch** bloğu içerisinde **rethrow** anahtar kelimesi ile aynı hatayı tekrar fırlatırız. Aşağıdaki kodda beş bloktan oluşan bir try-catch-finally yapısı görülmektedir. Sırayla blokları inceleyeceğiz olursak birinci blok try bloğu işlevsel kodun bulunduğu kısımdır. Eğer hataOlusturacakIslem fonksiyonu içerisinde bir hata oluşursa bu önce ikinci blokta ozelHataTuru ile belirtilen türde bir hata mı bakılır. ozelHataTuru değilse üçüncü bloğa bakılır. Eğer Exception türünde de değilse dördüncü bloğa akış geçer. 2., 3., ve 4. Bloklardan herhangi birine giren akış diğer catch bloklarına girmeyecektir. Son olarak finally bloğu her halükarda gerçekleşecektir.

Burada özel olarak ikinci blokta hata yakalanıp işlendikten sonra rethrow ile tekrar fırlatılmaktadır. Eğer hata ile ilgili bilgi isteniyorsa catch(e) ile hata ile ilgili temel bilgi edinilir. Hata dökümü (Stack Trace) almak için catch(e, s) şeklinde catch yapısı iki parametreli oluşturulur.

```
try {
  hataOlusturacakIslem();
} on ozelHataTuru {
  // Bizim tanımladığımız özel bir hata türü
  ozelHataYakalandi();
  rethrow; // Bulunan fonksiyon sonlanarak hata üst fonksiyona devreder.
} on Exception catch (e) {
  // Genel hata türü.
  print('Bilinmeyen hata: $e');
} catch (e , s) {
  // Herhangi bir tür tanımlaması olmaksızın herşeyi yakala
  print('Tahmin edilemeyen birşeyler oldu: $e');
  print('Stack trace:\n $s');
} finally {
  artiklariTemizle(); // Her halükarda bu fonksiyonu çağır.
}
```

2.9. Açıklamalar

Dart dilinde tek satır açıklama (comment) için iki bölmeye simgesi (//) çok satırlı açıklama yazmak için (/* ile */) arasında açıklamalar yazılır. C kökenli diğer dillerde aynı şekilde kod içerisinde açıklama yapılır. Üç adet bölmeye simgesi (///) kullanarak belgelendirme için açıklama girişi yapılır.

2.10. Metadata

Metadata kod içerisinde ek bilgi vermek gereken durumlarda kullanılır. Metadata dipnotu (annotation) tanımlamak için (@) karakteri kullanılır. @Deprecated, @deprecated, @override Dart dilinde kullanılan temel dipnotlardır. Bunların dışında kendi dipnot yapınızı da bir sınıf olarak tanımlayabilirsiniz.

2.11. Tip Tanımlama (Typedef)

TypeDef anahtar kelimesi bir veri türünün kod içerisinde yeniden isimlendirilmesi için kullanılır. Aşağıdaki örnekte IntList adında bir tanımlama yapılmıştır. Bunu bir takma isim verme gibi düşünebiliriz.

```
typedef IntList = List<int>;  
IntList il = [1, 2, 3];
```

Bölüm Özeti

Dart, Flutter çerçevesinin kullandığı programlama dilidir. C kökenli diller ailesinden olan Dart programlama dili C++, Java, C#, JavaScript dillerindeki iyi yönleri toplayıp basitleştirilmiş bir dil sentaksı sunmaktadır. Pek çok özelliği diğer dillerle aynı olmasının yanında kendine has özellikleri de vardır. Bunların başında dil içerisindeki her şeyin obje türü olması fonksiyonlarının, temel türlerin dahi obje olarak tanımlanması gelmektedir. Dart dili bu sayede oldukça işlevsel çalışabilmektedir. Null-Safety özelliği sayesinde içerisinde değer atanmayan yani null bir objenin değer atanmadan kullanımı engellenir. Bu çalışma zamanında tutarsız çalışan uygulamaların olmasını kısmen engeller. Dart dilinde dizi tanımlamak için List veri türü kullanılmaktadır. List, Set gibi diğer dillerde koleksiyon sınıfları olarak bulunan veri yapıları Dart dilinde veri türü olarak tanımlanmıştır. Fonksiyonların obje olarak tanımlanması iç içe fonksiyon tanımlama ve fonksiyonların dinamik bağlanması gibi ek kullanım imkanları sunar.

Kaynakça

<https://dart.dev/guides/language/language-tour>

3. DART PROGRAMLAMA DİLİ GELİŞMİŞ ÖZELLİKLER

Giriş

Bu bölümde Dart programlama dilindeki nesneye yönelik programlama mantığı ve asenkron programlama ile ilgili özellikler anlatılacaktır. Başlamadan önce genel olarak sınıf (class), kalıtım (inheritance), asenkron programlama, multi-thread programlama gibi kavramları anımsayalım. Bu kavumlara diğer dillerden aşina değilseniz konu ile ilgili bildiğiniz bir dilde bu kavamları incelemeniz Dart ile diğer dillerdeki yapısal benzerlik ve farklılıklarını anlamak adına yardımcı olacaktır.

3.1. Sınıf (Class)

Dart programlama dili nesne tabanlı bir programlama dilidir. Kalıtım modeli olarak sınıf (class) ve mixin tabanlı bir çözüm kullanmaktadır. Kod içerisindeki her bir obje (object), kendi sınıfından üretilen bir örnektir. Null haricindeki tüm sınıflar Object sınıfından türer. Dart dili Java, C# gibi dillerde olduğu gibi tek bir ana sınıfın miras almayı destekleri C++'da ise aynı anda birden fazla sınıfın miras alma yapmaktadır. Dart tek sınıfın miras almayı desteklemesinin yanında C++ programlama dilindeki bu esnekliği de mixin tabanlı miras almayı sağlamaktadır. Dart programlama dilinde bir sınıf ancak tek bir sınıfın miras alırken bunun yanında birden fazla mixin tabanlı miras alma yapabilir. Şimdi kısaca dilin temel nesneye yönelik programlama özelliklerinden bahsedeceğiz. Konu hakkında detaylı bilgiyi Dart.dev sitesindeki Dilin Özellikleri (Dart.dev->Guides->Language->Specification) sayfasından inceleyebilirsiniz.

```
class A {  
  var i = 0;  
  int? j;  
  f(x) => 3;  
}
```

Basit bir sınıf tanımlamasını yukarıdaki gibi yapabiliriz. Diğer nesneye yönelik programlama dillerinde olduğu gibi üye değişkenleri ve üye fonksiyonları bulunmaktadır. Bazı temel farklılıklara gelirsek;

Dart'ta public, protected ve private etiketleri kullanılmamaktadır. Alt çizgi (_) ile başlayan tanımlamalar private diğer tanımlamalar public olarak ele alınmaktadır. protected belirtecininse bir karşılığı Dart'ta mevcut değildir.

Üye değişkenlerin tanımlanmasında ilk değer ataması yapılabilir. Java gibi diller buna izin vermemektedir. Değer atanmayan değişken null değeri ile oluşturulur.

3.1.1. Sınıf Üyelerine Erişim

Sınıfin üyelerine erişim için sınıfın bir örneği üzerinden nokta(.) operatörü ile üye değişkenlere erişilir ve üye fonksiyonlar çağrılır.

```
var a = A();
// Değere erişim.
assert(a.i == 0);
// Fonksiyon çağrısı
int b = a.f(1);
```

Eğer sınıfın örneğinin null olma ihtimali varsa üyelerine erişimde hata oluşturulmaksızın kodun ilerlemesi için soru işaretçi-nokta (?) operatörünü kullanabiliriz.

```
// Eğer a null değil ise j değeri c ye aktarılır.
var c = a?.j;
```

3.1.2. Yapıçı (Constructor) Fonksiyonlar

Dart'ta yapıçı fonksiyonlar normal tanımlanabileceği gibi isimlendirilmiş yapıçı fonksiyonlar da tanımlanabilir. Bu yapıçı fonksiyonun işlevini kodu okuyana aktarmada iyi bir yöntemdir.

```
var p1 = Point(2, 2);
var p2 = Point.fromJson({'x': 1, 'y': 2});
```

Bu iki örnekte Point adındaki sınıfından p1 ve p2 örnekleri türetilmiştir. p1'in oluşturulmasında klasik iki parametreli bir yapıçı fonksiyon kullanılırken, p2'nin oluşturulmasında fromJson şeklinde adlandırılmış bir yapıçı fonksiyon kullanılmıştır. Burada ek bilgi okumaksızın fromJson yapıçı fonksiyonunun bir Point objesi oluşturmak için girdiyi JSON formatında aldığıni anlayabilmekteyiz.

Bazı sınıflarda const yapıçı fonksiyon tanımlama yapılmamaktadır. const bir yapıçı fonksiyon kullanımı kod içerisinde aynı parametre listesi ile çağrılan birden fazla örnek için tek bir sabit oluşturur.

```
var a = const BasePoint(1, 1);
var b = const BasePoint (1, 1);
// a ve b aynı yerde tutulan bilgiye iki referanstır.
```

Eğer sınıfın bir yapıçı fonksiyonu bulunmuyorsa varsayılan olarak parametre almayan yapıçı fonksiyonu oluşturulur.

Dart miras alma yoluyla yapıçı fonksiyonları alt sınıflara aktarmaz. Eğer bir türemiş sınıf kendi yapıçı fonksiyonunu içermezse sadece parametresi olmayan varsayılan yapıçı fonksiyonu olacaktır.

Dart üye değişkenlere veri yüklemeyi kolaylaştırmak için this anahtar kelimesini yapıçı fonksiyon parametre listesinde kullanır. Böylece sadece üye değişkenlere ilk değer vermek için yazılan yapıçı fonksiyonları gövdesiz hale getirebilmektedir.

```
class Point {  
    double x = 0;  
    double y = 0;  
    Point(double x, double y) {  
        this.x = x;  
        this.y = y;  
    }  
}
```

Yukarıdaki yapıçı fonksiyon yerine;

```
class Point {  
    double x = 0;  
    double y = 0;  
    Point(this.x, this.y);  
}
```

şeklinde yazılabılır.

Yapıcı fonksiyonun gövdesinin işlenmesine başlamadan önce eğer yapılmasını istediğiniz işlemler varsa örneğin üst sınıfın yapıçı fonksiyonunun çağrılmaması , bazı üye değişkenlere ilk değer atanması gibi, bu durumda iki nokta üst üste(:) operatöründen sonra bu işlemler sıralanabilir.

Üst sınıf tanımlamasına erişmek için **super** anahtar kelimesi kullanılmaktadır.

```
class Employee extends Person {  
    Employee() : super.fromJson(fetchDefaultData()) , x = 'unassigned';  
    // ...  
}
```

Benzer kullanımı yapıçı fonksiyonlar arasında yönlendirme için de kullanabiliriz. Örneğin iki parametre alan yapıçı fonksiyon içerisinde parametre almayan yapıçıdaaki işlemlerin yine aynen bulunması gerekiyorsa bu sefer **super** yerine **this** anahtar kelimesi ile kendi içindeki diğer yapıçı fonksiyonları çağırabilir.

```
class Point {  
    double x, y;  
    Point(this.x, this.y);  
    // isimsiz yapıçıya işi yönlendirir.  
    Point.alongXAxis(double x) : this(x, 0);  
}
```

Dart'ta bir diğer yapıçı fonksiyon özelliği **factory** anahtar kelimesi ile tek bir örnek üzerinde çalışan bir yapıçı oluşturmaya izin vermesidir. **factory** anahtar kelimesiyle tanımlanan yapıçı fonksiyon her çağrılığında sınıfın yeni bir örneğini oluşturmaz. Bu kullanım özellikle tampon bellek olarak saklanacak bilgilerin yoksa oluştur varsa kullan formundaki kullanımını için kolaylık sağlar.

Aşağıdaki **Logger** sınıfının amacı **log** adlı üye fonksiyon çağrılığında gönderilen mesajı belirtilen kütüğe yazmaktadır. (Buradaki örnekte implemantasyonu yapılmamış sadece konsola mesajı geri basmaktadır.) Uygulama içindeki tüm kütükleri (log kayıt yerlerini) ifade etmek için **_cache** adında **static final** bir **Map** sınıf içerisinde tanımlanmıştır. Bu tanım static olduğu için bellekte bir kere olacak, final olduğu için oluşumu ilk yapıçı çağrısında gerçekleşecek ve altçizgi ile tanımlandığı için private olacak yanı **Logger** sınıfı dışında erişime kapalı olacaktır. **Logger** sınıfından yeni örnek türetecek tek yapıçı fonksiyon **Logger._internal**'dır. Bu da altçizgili tanımlandığı için sadece sınıf içerisinde erişilebilir. Bunu çağrıran isimsiz factory yapıçı fonksiyona (**factory Logger(String name)**) dikkat ederseniz **_cache** Map'ı üzerinde eğer ilgili **Logger** kaydı yoksa bir yenisini oluşturur ve referansını döner ama varsa sadece referansını döner. **Logger.fromJson** yapıçı fonksiyonu da verilen JSON formundaki name-Logger eşleşmesine uyan kaydı bulmak için diğer factory yapıçı fonksiyonu çağrıır. factory yapıçı fonksiyonu diğer dillerdeki static yapıçı fonksiyonu tanımlamaya benzer özel bir yaklaşımındır. factory yapıclarla ilgili bilinmesi gereken diğer bir durum da **this** anahtar kelmesinin bu yapıclar içerisinde kullanılamayacağıdır.

```

class Logger {
  final String name;
  bool mute = false;
  static final Map<String, Logger> _cache =
    <String, Logger>{};
  factory Logger(String name) {
    return _cache.putIfAbsent(
      name, () => Logger._internal(name));
  }
  factory Logger.fromJson(Map<String, Object> json) {
    return Logger(json['name'].toString());
  }
  Logger._internal(this.name);
  void log(String msg) {
    if (!mute) print(msg);
  }
}

```

Yukarıdaki kod parçası gibi çağrılır.

```

var logger = Logger('UI'); //factory yapıcı çağrılmıyor. UI yok oluşturulur
                           //cache'e eklenir.
logger.log('Buton tıklandı');
var logMap = {'name': 'UI'}; //isimlendirilmiş factory yapıcı çağrılmır.
                           //UI zaten _cache de tanımlıydı. putIfAbsent
                           //fonksiyonu Map'deki kaydı getirir.
var loggerJson = Logger.fromJson(logMap);

```

3.1.3. Operatörler

Sınıflar için operatör tanımlaması yapılabilir. Buna genellikle operatör aşırı yükleme (operator overloading) denmektedir. Bir sınıf için operatör tanımlanması o sınıfın temel veri türleri gibi bu operatörlerle kullanılmasına imkân sağlar.

Tanımlanabilecek operatörlerin bir listesi;

```

< + | []
> / ^ []=
<= ~/ & ~
>= * << ==
- %>>

```

Buradaki örnekte **Vector** adında oluşturulan sınıf içerisinde (+) ve (-) operatörleri tanımlanmıştır. Bu tanımlama sayesinde main içerisinde oluşturulan **v** ve **w** adındaki iki örnek üzerinde toplama ve çıkartma işlemleri gerçekleştirilir.

```

class Vector {
  final int x, y;
  Vector(this.x, this.y);
  Vector operator +(Vector v) => Vector(x + v.x, y + v.y);
  Vector operator -(Vector v) => Vector(x - v.x, y - v.y);
  // Operator == and hashCode da olması gereken operatör tanımlamalarıdır.
  // ...
}
void main() {
  final v = Vector(2, 3);
  final w = Vector(2, 2);
  assert(v + w == Vector(4, 5));
  assert(v - w == Vector(0, 1));
}

```

3.1.4. Getter ve Setter

get ve set anahtar kelimeleri C# ve Java gibi dillerde bir objenin özelliklerini (property) tanımlamak için kullanılırlar. Dart'ta da bu yaklaşımı görmekteyiz. Gerektiğinde bir özelliğin değerinin değiştirilmesi veya elde edilmesi için hesaplama gereken durumlarda get ve set anahtar kelimeleri ile getter ve setter tanımlaması yapılabilir.

Aşağıdaki örnekte right ve bottom özellikleri left-witdh ve top-height değer çiftlerine bağlıdır. Bu nedenle right ve bottom üye bir değişken olarak değil bir özellik(property) olarak tanımlanmak durumundadır.

```
class Rectangle {  
    double left, top, width, height;  
    Rectangle(this.left, this.top, this.width, this.height);  
    // right ve bottom hesaplanan iki değerdir.  
    double get right => left + width;  
    set right(double value) => left = value - width;  
    double get bottom => top + height;  
    set bottom(double value) => top = value - height;  
}  
void main() {  
    var rect = Rectangle(3, 4, 20, 15);  
    assert(rect.left == 3);  
    rect.right = 12;  
    assert(rect.left == -8);  
}
```

3.1.5. Abstract Sınıf ve Abstract Metod

Dart, soyut (abstract) sınıf ve metod tanımlamaya izin vermektedir. Soyut sınıf tanımlamak için sınıf önüne abstract anahtar kelimesi yazılır. Soyut bir sınıfın örneği oluşturulamaz ancak miras alınarak kullanılması gereklidir. Genelde soyut sınıflar içerisinde soyut fonksiyonlar barındırır. Soyut fonksiyon tanımı gövdesi yazılmamış sadece isimlendirilmiş fonksiyon manasındadır. Soyut fonksiyon tanımlamak için kümeye parantezleri dahil fonksiyon gövdesi çıkarılarak sadece noktalı virgül (,) simbolü ile fonksiyon tanımlaması bitirilir.

```
abstract class Olusturucu {  
    // İçerisinde Üye değişken ve fonksiyonlar tanımlanır.  
    void birseyYap(); // soyut fonksiyon tanımlaması  
}  
class EtkinOlusturucu extends Olusturucu {  
    void birseyYap() {  
        // Burada soyut sınıfından miras alınan soyut fonksiyonun gövdesi oluşturulur.  
    }  
}
```

3.1.6. Implements ve Extends ile Miras

Nesneye yönelikli dillerde arayüz (interface) tanımlamak için özel bir kalıp vardır. Dart ise arayüz tanımlaması için yine sınıfları kullanır. Eğer bir sınıfın sadece davranışsal bir görüntüsü ile çalışılacaksa, yani üye değişkenleri alınmayacak sadece fonksiyonlarının prototipi alt sınıfa aktarılacaksa **implements** anahtar kelimesi ile üst sınıfı bir arayüz gibi kullanırız. Dart birden fazla sınıftan **implements** ile arayüz formunda miras almayı destekler.

```

// Person sınıfı _name değişkeni ve greet fonksiyonu tanımlıdır.
class Person {
    // _name sadece bu kütüphane içerisinde görülen bir değişkendir ()
    final String _name;
    // Person sınıfının yapıcı fonksiyonu
    Person(this._name);
    // greet fonksiyonunun Person sınıfında gerçekleşmiş hali
    String greet(String who) => 'Hello, $who. I am ${_name}.';
}
// Person sınıfından implements ile bir arayüz miras alımı
class Impostor implements Person {
    String get _name => '';
    String greet(String who) => 'Hi $who. Do you know who I am?';
}
String greetBob(Person person) => person.greet('Bob');
void main() {
    print(greetBob(Person('Kathy')));
    print(greetBob(Impostor()));
}

```

Yukarıdaki kod parçasında Person adındaki sınıfın implements ile arayüz formunda miras alınmıştır. Arayüz olarak alınan miras olduğu için Person sınıfındaki _name ve greet() Impostor sınıfına aktarılmamıştır. Imposter'a sadece tek String parametre alan ve String dönüşüm değeri olan greet adındaki bir fonksiyonu yeniden gerçekleştirmesi gerekiği bilgisi aktarılmıştır. Imposter içerisinde _name tanımlı olmadığı için yeniden tanımlanabilemektedir. greet metodu ise Person arayüzünden aktarılan yapısına uygun şekilde yeniden tanımlanmalıdır.

Bu yapıdaki güzel taraf şudur. greetBob adındaki global fonksiyon Person türünde bir parametre almaktadır ve greet metodunu çağrımaktadır. main içerisindeki greetBob çağrılarında birinde Person diğerinde ise Imposter türünde yaptığımız parametre aktarımlarında farklı greet metodlarının çağrıldığı görülecektir.

Bir sınıfın birden fazla arayüzden miras alması mümkündür.

```
class Point implements Comparable, Location {...}
```

Bir sınıfın miras alırken tüm özellik ve davranışlarını devralmasını istiyorsak **extends** anahtar kelimesi ile miras alımı yapılmalıdır. extends ile miras alımlarında sadece tek bir sınıfın miras alımı yapılır. Extends ile miras alımında amaç miras alınan sınıfın özelliklerini genişletmek suretiyle aynı yapıda daha gelişmiş bir sınıf oluşturmaktr. Extends ile miras alındığında ebeveyndeki tüm her şey miras alınır. **@override** dípnotu (annotation) ile miras alınan fonksiyonlar bilinçli şekilde ezilebilir.

```

// First adındaki sınıf
class First {
  static int num = 1;
  void firstFunc(){
    print('hello');
  }
}

// First sınıfından miras alan
class Second extends First{
// Her hangi bir override gerekmeyez.
}
void main(){

  // First sınıf örneği
  var one = First();

  // firstFunc() çağrıı
  one.firstFunc();

  // static değişkenin konsola yazdırılması
  print(First.num);

  // Second sınıf örneği
  var second = Second();

  // miras alınmış firstFunc()
  // için çağrı
  second.firstFunc();
}

```

3.1.7. noSuchMethod()

Miras alma süreçlerinden kaynaklı dinamik oluşturulmuş bir objenin üye fonksiyonlarını çağrıırken olmayan bir üye çağrıı durumunda uygulamanın hata vermesi yerine bunu düzenleyecek noSuchMethod() fonksiyonu kullanılmaktadır.

```

class A {
  // noSuchMethod kodlanmadığında olmayan fonksiyona erişim
  // NoSuchMethodError hatasının oluşmasını sağlar.
  @override
  void noSuchMethod(Invocation invocation) {
    print('You tried to use a non-existent member: '
      '${invocation.memberName}');
  }
}

```

3.1.8 Mixin ile Sınıflara Özellik Ekleme

Dart ile birden fazla sınıfın(arayüzden) implements ile davranış alabileceğimizi ama sadece bir tane ana miras yolunu extends ile belirttiğimizi söylemiştık. Eğer birden fazla koldan sınıfın özellikler alması gerekiyorsa çözümümüz ne olmalıdır? C# ve Java dillerinde nesne yönelimli yaklaşımın bellek yönetiminin zorlaştıracağı için bir çözüm sunulmamış C++’da ise tam aksine çok sayıda ana hattan miras alımına izin verilmiştir. Dart ise bu ikisinin arasında bir çözümü mixin adı verilen yan parçacıkların with anahtar kelimesi ile miras alımına eklenmesiyle sunmaktadır.

```

class Musician extends Performer with Musical {
    // ...
}

class Maestro extends Person
    with Musical, Aggressive, Demented {
    Maestro(String maestroName) {
        name = maestroName;
        canConduct = true;
    }
}

mixin Musical {
    bool canPlayPiano = false;
    bool canCompose = false;
    bool canConduct = false;
    void entertainMe() {
        if (canPlayPiano) {
            print('Playing piano');
        } else if (canConduct) {
            print('Waving hands');
        } else {
            print('Humming to self');
        }
    }
}

```

Burada Musician, Performer sınıfından miras almaktadır. Performer sınıfının yanında Musical adındaki mixin'in de belirttiği özellikleri bünyesine eklemiştir. Maestro, Musician'a göre daha donanımlıdır ve başka ek özellikleri olmalıdır ama bunları Performer'a ekleyemeyiz ve Maestro, Musician sınıfından da miras alamaz (Bunun kavramsal olarak neden olamayacağı nesne modelleri ve ilişkileri konusudur. Burada detaya inilmeyecektir.). Maestro'nun içermesi gereken bu ek özellikler farklı mixin'ler olarak eklenmiştir. Görüleceği üzere birden fazla mixin **with** anahtar kelimesi ile miras almada kullanılabilir.

Eğer bir mixin'in kullanımında belli bir sınıfın miras alınması gereklisi ise mixin oluşturulurken **on** anahtar kelimesi ile bu durum belirtilebilir. Bazı fonksiyonların çalışabilmesi için belli özelliklerin tanımlanması gereken durumlar için bu yaklaşım gerekebilir.

```

class Musician {
    // ...
}

mixin MusicalPerformer on Musician {
    // ...
}

class SingerDancer extends Musician with MusicalPerformer {
    // ...
}

```

3.2. Kütüphaneler ve Görünürlük

Dart zengin bir kütüphane yapısına sahiptir ve bu topluluğun aktif katılımı ile giderek artmaktadır. Kütüphaneler **package** olarak **pub.dev** sayfasında paylaşılır. Kütüphaneler aynı zamanda altçizgi() ile başlayan objelerin kütüphane dışından erişimine izin vermediği için bir gizlilik de sağlarlar. Her uygulama aynı zamanda bir kütüphanedir.

Projenize bir kütüphane dahil etmek istediğinizde import anahtar kelimesi ile kütüphane yolunu (URI-Unique Resource Identifier) vermeniz yeterlidir.

```
import 'dart:html';
import 'package:test/test.dart';
```

Yukarıda Dart dağıtım ile gelen html kütüphanesi ve paket olarak yazılmış test kütüphanesinin koda dahil edilmesi örnekleri verilmiştir. pub.dev üzerinden sağlanan paketlerin dahil edilmesinde **package:** ile yol belirtilmektedir.

Bir kütüphane projenize dahil ettiğinizde içerisindeki tüm tanımlamalar projenize dahil olurlar. Bazen farklı iki kütüphanede aynı isimlendirme kullanılabilir. Bu durumda ilk kütüphane içerisindeki tanımlama kabul görecektir. İkinci kütüphanenin takılanması sayesinde çakışma giderilir. Örnekte ikinci kütüphane **as lib2** olarak takma isim verilerek import edilmiştir. Kod içerisinde de bu takma isim kullanılarak içerdeği elemanlara erişilmektedir.

```
import 'package:lib1/lib1.dart';
import 'package:lib2/lib2.dart' as lib2;
// lib1 içerisindeki Element tanımına erişir.
Element element1 = Element();
// lib2 içerisindeki Element tanımına erişir.
lib2.Element element2 = lib2.Element();
```

Bazen de kütüphane içerisinde sadece belli tanımların görünmesi veya gizlenmesi istenebilir. Bunun için **show** ve **hide** anahtar kelimeleri kullanılır.

```
// Sadece foo'yu dahil et
import 'package:lib1/lib1.dart' show foo;
// foo'yu hariç tutarak import et.
import 'package:lib2/lib2.dart' hide foo;
```

Kütüphanelerin paketler şeklinde Internet üzerinden paylaşımı kod geliştirme süreçlerini kolaylaştırmıştır. Artık pek çok fonksiyon için yazılımcılar başkaları tarafından hazırlanmış paketler bulup açık kaynak dağıtım sayesinde kullanabilmektedir. Siz de isterseniz kendi paketinizi üreterek pub.dev altında yayınlanmasını sağlayabilirsiniz. Bununla ilgili adımlar Creating packages sayfasından incelenebilir.

3.3. Genişleme Metotları (Extention Methods)

Dart 2.7 sürümünde eklenmiş olan bu özellik müdahale etmediğiniz sınıf tanımlamalarını genişletmek ve istediğiniz metotların bu sınıflar için tanımlanmasında kullanılır. Kod yazarken IDE'nin sunduğu otomatik tamamlama özelliğinde genişleme metotları da önerilmektedir. Dart'ta temel türler dahi birer obje oldukları için onlara hatta kod içerisindeki sabitlere de genişleme metotları uygulanır.

Örneğin bir String değerinin tamsayı türüne dönürtülmesi için int tanımlamasında parse fonksiyonu vardır. Fakat String tanımında bir fonksiyon bulunmamaktadır. Oysaki işlem yapılan veri String olduğu için String veriye bir metot uygulamak daha akla yatkın bir çözümdür. Bunun için aşağıda birinci satırdaki işlemi ikinci satırdaki gibi yazabiliriz.

```
int.parse('42');
'42'.parseInt();
```

String tanımlaması Dart'in ana kodlamasında tanımlıdır ve parseInt adında bir metodu bulunmamaktadır. Dart geliştiricileri bu genişleme fonksiyonunu sisteme daha sonra bir kütüphane olarak dahil etmiştir. Bunun için aşağıdaki import işlemi yapılmıştır. Burada string_apis.dart dosyası parseInt metodunun String veri tipi için tanımlandığı yerdir.

```
import 'string_apis.dart';
// ...
print('42'.parseInt()); // Genişleme metodu kullanımı.
```

Bu genişleme metodunun tanımlanması için aşağıdaki gibi extension anahtar kelimesi ile başlayan bir tanımlama bloğu açılır. Bu blok sayesinde String sınıfı genişletilmiş ve verilen yeni özelliklere sahip olmuş olur.

```
extension NumberParsing on String {  
    int parseInt() {  
        return int.parse(this);  
    }  
    // ...  
}
```

3.4 Jenerik (Generics)

Veri türlerini incelerken diziler için **List** veri türü kullanıldığını ve bunun aslında diğer dillerde koleksiyonlarla yapıldığını söylemiştık. Aslına bakarsanız her iki tanımlamada da kullanılan yöntem **generics** olarak adlandırılan ve **List<E>** (E burada jenerik olarak sonradan atanacak türü ifade etmektedir.) gösterimi ile tanımlanmıştır.

Jenerik tanımlaması kodun daha iyi derlenmesine katkı sağlar ve gereksiz kod tekrarlarını engeller. Bir dizide sadece String değerlerin saklanması istiyorsak **List<String>** şeklinde bir dizi tanımlaması yaparız. Bu sayede farklı türde verinin bu listede yer almayaceği beyan edilmiş olur.

```
var names = <String>[];  
names.addAll(['Seth', 'Kathy', 'Lars']);  
names.add(42); // Hata: names alanı sadece String türünde değer alır.
```

Kod tekrarını azaltmak için geliştirdiğimiz yapıları jenerik formunda hazırlayabiliriz. Bu sayede aynı kod farklı yerlerde farklı veri türleri ile çalışabilir. Aşağıdaki örneği inceleyelim;

```
abstract class Cache<T> {  
    T getByKey(String key);  
    void setByKey(String key, T value);  
}
```

Burada Cache adındaki sınıf jenerik olarak tanımlanmıştır ve T adı altında henüz belirtilmemiş bir veri türü ile çalışabilir denmektedir. Bu T türü örneğin **Cache<int>** şeklinde bir kullanımda int, **Cache<String>** şeklinde bir kullanımda String veri türü olarak değiştirilecek ve buna göre sınıf içerisindeki fonksiyonların varyantları üretilecektir. **Cache<int>** için durumu düşünürsek **getByKey** metodunun dönüş değerinin int türünde olacak ve **setByKey** metodunun ikinci parametresi de int türünde olacaktır. Sonuçta Cache sınıfının **T:int** olan varyasyonu için bir derleme yapılacaktır. Aynı süreç T yerine gelecek her veri türü için tekrarlanır ve tek bir kod bloğu farklı veri türlerinde tekrar farklı veri türüyle derlenir.

3.5. Asenkron Uygulama Geliştirme Desteği

Gelişmiş programlama dillerinde sunulan en önemli özellikler arasında asenkron kod çalışma desteği gelmektedir. Asenkron kodlama ihtiyacının çoğu dildeki temel çözümü multi-thread (çoklu işlenme) uygulama geliştirmektir. Multi-thread uygulama geliştirmek yazması ve bakımı açısından zor bir süreçtir. Birbirinden bağımsız iş parçalarının ortak alanları kullanım için birçok düzenleme gerekmektedir. Yanlış yazılan bir kod parçası tüm iş parçalarının çalışmasını engelleyebilir.

Dart dili asenkron desteğini basit bir forma sunmaktadır. Multi-Thread kodlama gerektiren yerlerde cevabı sonradan donecek fonksiyon çağrıları kullanmaktadır. Çünkü Multi-Thread kodlamadaki yegâne amaç CPU'nun I/O gecikmelerinden etkilenmemesi ve tam kullanılabilmesidir. Dart, bir fonksiyonun yüksek bir I/O gerektirmesi durumunda kodun bu fonksiyon sonucunu beklemeksizin çalışması için **Future** ve **Stream** adında iki veri türü kullanmaktadır.

Kod içerisinde sonradan gelecek veri kısmının asenkron çalıştırılması ve icap ettiğinde gelecek sonucun beklenmesi sırasıyla **async** ve **await** anahtar kelimeleri ile sağlanmaktadır. İçerisinde async ve await anahtar kelimeleri yer alan bir kod asenkron bir yapıdadır fakat senkron çalışan bir kod ile aynı görünümde dir.

```
Future<void> checkVersion() async {
  var version = await lookUpVersion();
  // versiyon bilgisi ile işlemler.
}
```

Yukarıda **checkVersion** adında bir fonksiyon tanımlanmaktadır. Bu fonksiyonun çağrılmacağı yerde asenkron çalışmasını istiyoruz. Yani ben main içerisinde **checkVersion** fonksiyonunu çalıştırıyma ama sonucunu beklemeden bir sonraki komutu çalışmaya geçeyim. Bunu sağlamak için **checkVersion** fonksiyonunun tanımında **async** anahtar kelimesi kullanılır. **async** kullandığımız için fonksiyon geri dönüş değeri olarak **Future** veya **Stream** türü bir değer dönmelidir. **checkVersion** fonksiyonunun herhangi bir değer dönmesini istemesek de Future dönmesi gereklidir. Bu nedenle **Future<void>** formunda boş bir **Future** dönülür.

Asenkron çalışan bir fonksiyon içerisinde başka asenkron veya senkron fonksiyonlar çağrılabılır. Asenkron fonksiyon çağrısında eğer devam eden komutlarda bu fonksiyonun işini bitirmesi gerekiyorsa bekletme için **await** komutu kullanılır. Yukarıdaki kodda **lookUpVersion** fonksiyonu da asenkron çalışan başka bir fonksiyon olsun. **checkVersion** fonksiyonunun devamında **lookUpVersion**'dan dönen **version** bilgisine ihtiyaç duyuyorsam **lookUpVersion** fonksiyonu önüne **await** koyularak çağrılr. **await** önüne geldiği fonksiyonun işi bitmeden sonraki kodun icra edilmesini engeller. Bir nevi senkron çalışmaya zorlar. Böylece **async** blokları içerisinde senkron olarak yapılması gereken işlemler düzenlenirler. Bir asenkron (**async**) fonksiyon çağrısi içerisinde birden fazla **await** kullanımını olabilir.

Bazı yerlerde **await** kullanma ihtiyacınız olduğu halde hata alıyorsanız (örneğin main içerisinde asenkron bir fonksiyon çağırıcasanız ve senkron iş görmesi gerekiyorsa) **await** yazılan fonksiyon gövdesi **async** ile kuşatılır. Aşağıdaki örnekte **lookUpVersion** fonksiyonu main içerisinde konsola veri yazmak için çağrılmaktadır. Konsola yazan **print** metodu **lookUpVersion** sonucunu beklemelidir bu nedenle **await** kullanılır. **await** kullanımından dolayı main fonksiyonu **async** olmalı ve bundan dolayı da **Future<void>** geri dönüş değerine sahip olmalıdır.

```
Future<void> main() async {
  checkVersion();
  print('In main: version is ${await lookUpVersion()}');
}
```

Future dönüş değeri eğer asenkron çalışacak kod bloğu işlem sonunda tek seferde sonuç döneceksse kullanılmaktadır. Eğer asenkron kod bloğu işlem yaparken sürekli bir veri akışı olacak ve bu veri akışının eş zamanlı olarak işlenmesi gerekiyorsa **Future** yerine **Stream** türü jenerik bir dönüş değeri kullanılır. Future ve Stream kullanımı ile ilgili alıştırmaya Asynchronous programming: futures, **async**, **await** ve Asynchronous programming: Streams bağlantılarından erişilebilir.

async fonksiyon tanımlamak asenkron kod yazmanın kolay yoludur ve senkron bir kodlama yaklaşımı içerisinde asenkron çalışan uygulama geliştirmemizi sağlar. Bunun haricinde Dart multi-thread yaklaşımı için **isolate** olarak adlandırılan bir modeli desteklemektedir. Klasik multi-thread yaklaşımında tüm thread'ler ortak erişimli bir bellek bölgesi kullanırlar. Bu kodun kararsız olmasına neden olabilir ve uygulamanın çalışmasında çeşitli komplikasyonlar oluşturur. Dart **isolate** kullanarak ayrı koşacak her bir kod parçasını kendine ayrılmış bellek bölgesine erişimle çalıştırır. Böylece **isolate**'ler birbirini tutarsız hale getirme tehlikesine girmezler. Arkaplan işlemleri ile ilgili konu işlenirken **isolate** çalışma mantığı inceleneciktir. Konu ile ilgili örneğe Dart asynchronous programming: Isolates and event loops | by Kathy Walrath sayfasından bakılabilir.

Bölüm Özeti

Bu bölümde Dart programlama dilinin nesneye yönelikli programlamadaki kullanım şekline diğer dillere göre farklı miras alma modeline implements, extends, with anahtar kelimeleri ile miras alma yaklaşımına

bakılmıştır. C# ve Java ile C++ arasındaki çoklu miras alma modeli farklılığında Dart mixin-based bir yaklaşım sunarak ikisi arasında bir model kullanmaktadır. public, private ve protected gibi erişim belirteçleri Dart programlama dilinde kullanılmamıştır. Bunun yerine private ve protected manasına gelecek şekilde altıçizgi(_) ile tanımlama yapılması benimsenmiştir. Asenkron uygulama geliştirmek için Dart basit fonksiyon tabanlı asenkron kod çalıştırmayı async ve await anahtar kelimeleri ile desteklemektedir. Bu fonksiyonlar Future veya Stream türü dönüş yapmaktadır ve dönülen değer daha sonra hazır hale gelmektedir. Genişletme metodları ile üzerinde değişiklik yapma imkanımız olmayan kütüphanelerdeki sınıfların davranışlarını zenginleştirme imkanımız vardır. Dilin bu özelliklerinin yanında tam bir içerik Dart.dev internet adresi üzerinden incelenebilir.

Kaynakça

<https://dart.dev/guides/language/language-tour>

<https://dart.dev/guides/language/specifications/DartLangSpec-v2.10.pdf>

<https://dart.dev/guides/libraries/library-tour>

4. WIDGET KAVRAMINA GİRİŞ

Giriş

Bu bölümde ilk gerçek mobil uygulamamızı gerçekleştireceğiz. Bölüm içerisinde verilen işlemleri yerine getirmek için ilk bölümde verilen kurulum işlemlerini tamamlamış olmanız gerekmektedir. Konunun anlaşılmasına için Dart diline tam bir hakimiyet gerekmese de ikinci ve üçüncü dersteki konulara yer yer atıfta bulunulacaktır. Görsel arayüz tasarımları özel bir meziyettir. Bu derste profesyonel bir arayüz tasarımları değil arayüz tasarımlında kullanılabilecekler öğretilecektir. Sonraki derste görsel elemanların bir araya getirilmesi konusunu çalışıktan sonra görsel tatminliği sağlayacak arayüz tasarımları ancak bolca deneme yaparak olur.

4.1. İlk Proje Denemesi

Birinci dersi örnek proje oluşturma adımıyla bitirmiştik. Bu derste örnek proje ve projenin VS Code üzerinde görünümüne bakacağız. Flutter projesi oluşturmak için terminalde istediğiniz klasör altında

```
> flutter create ders4_1
```

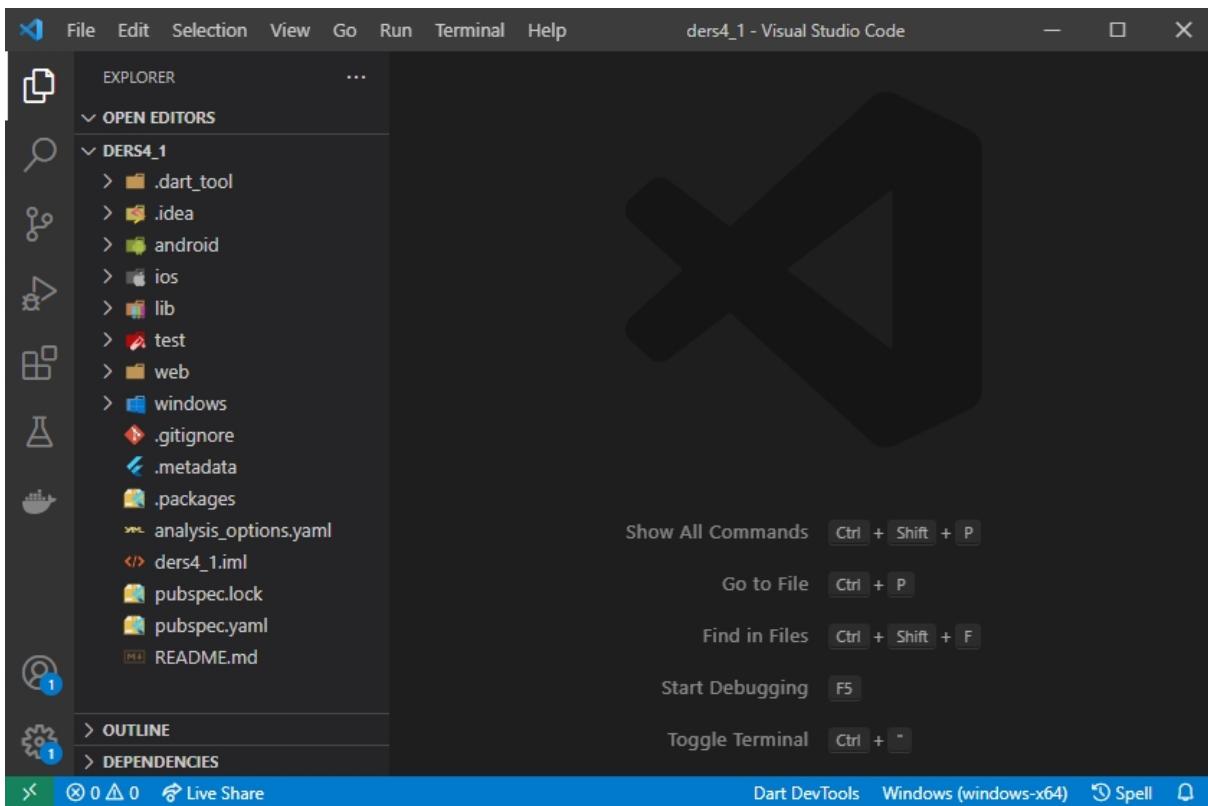
komutu ile ders4_1 adında bir proje oluşturabiliriz. Sonrasında

```
> cd ders4_1
```

```
> code ./
```

Komutlarını sırası ile çağırarak proje klasörünü VS Code editöründe açabilirsiniz.

VS Code sizi aşağıdakine benzer bir klasör yapısı ile karşılaşacaktır.

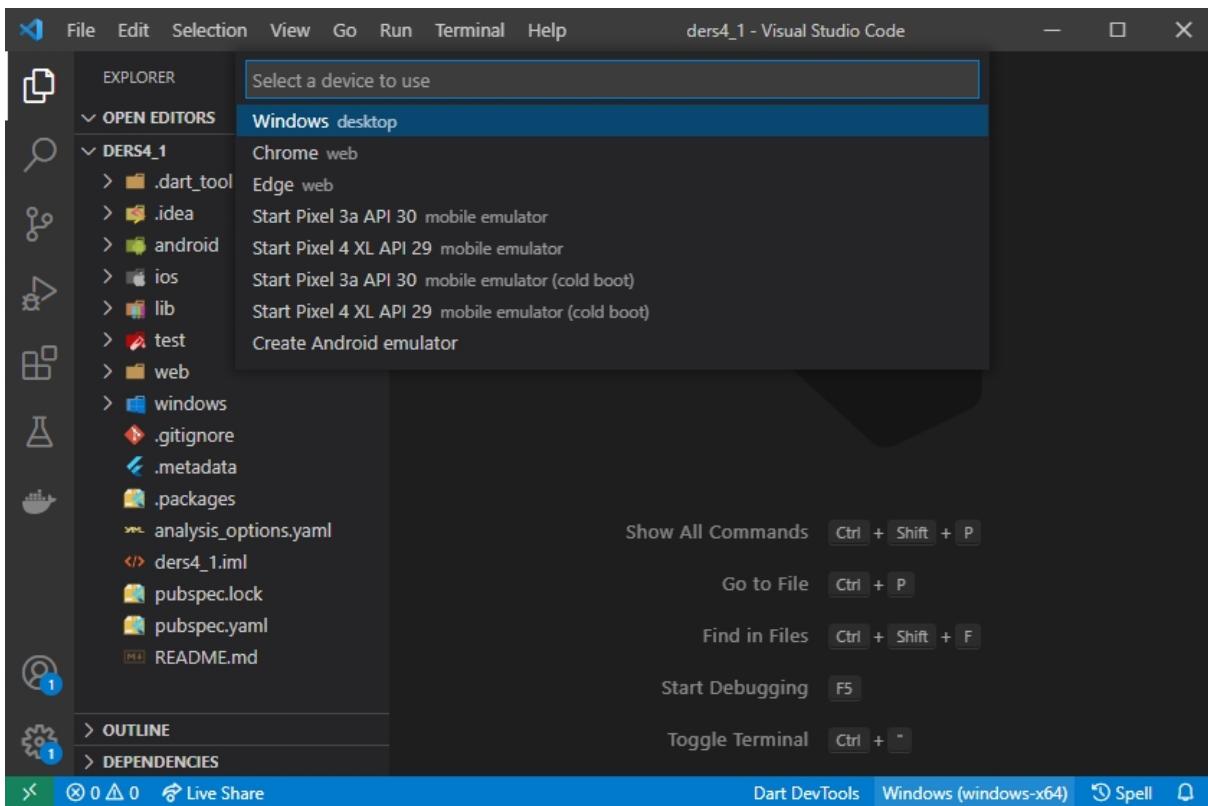


Kurulumda seçtiğiniz kanal ve kullandığınız işletim sisteme göre bazı klasörleri görmeyebilirsiniz. Bu aşamada her şeyin yolunda olup olmadığını test etmek için sıfır projemizi çalıştırmayı deneyeceğiz. Sağ alt kısımda DartDevTools ve Windows (windows-x64) yazılarına dikkat ediniz.

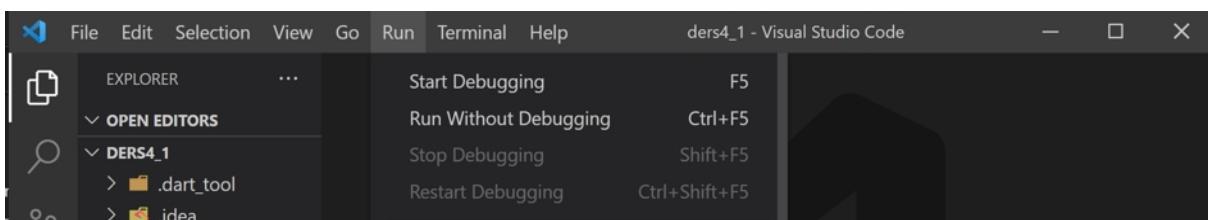
DartDevTools VS Code içerisinde Flutter uygulamalarını debug etmek için kullanacağımız bir eklentidir ve VS Code'da uygulama geliştirirken çokça başvuracağımız bir özelliklektir. Eklenti yönetiminde kurulu değilse aratıp kurunuz. Nasıl kurulacağı ile ilgili anlatımı Install and run DevTools from VS Code adresinden inceleyebilirsiniz.

İkinci kısımda yazan Windows (Windows-x64) sizin kurulumunuzda farklı görünebilir. Bu aktif çalışma seçeneğini belirtmektedir. Bu kurulumda masaüstü(desktop) desteği olduğu, Windows bir işletim sistemi kullanıldığı ve başka bağlı bir cihaz veya çalışır emülatör olmadığı için varsayılan çalışma şekli masaüstü uygulaması olarak gelmektedir. Akıllı telefonlara has özellikler işlenmediği sürece dersteki örnekler masaüstü uygulaması olarak gerçekleştirilecektir. Bilgisayar kaynaklarınızı fazla zorlamamak için size de masaüstü desteğini aktifleştirmeyi öneririm. Nasıl yapılabileceği Desktop support for Flutter adresinden incelenebilir.

Windows(Windows-x64) yazısı üzerine tıkladığınızda VS Code size alternatif cihaz bilgisini getirecektir.

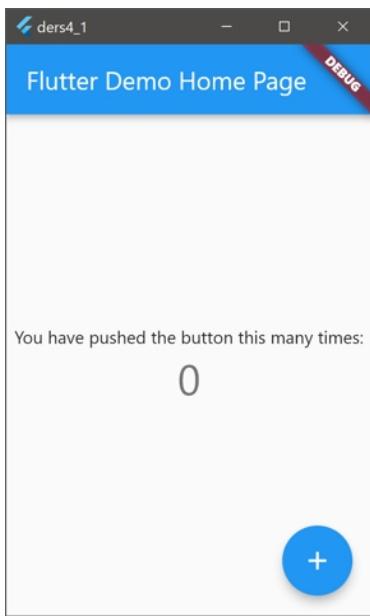


Bu listede kurulu Emülatörler, web ve desktop çalışma kipleri görülmektedir. Herhangi birinin seçilmesinden sonra uygulamayı çalıştırmak için **Run (Çalıştır)** menüsü altından **Start Debugging** (Kısayolu F5) veya **Run Without Debugging** (Kısayolu Ctrl+F5) seçeneklerinden biri ile uygulama çalıştırılır.



“Start Debugging” uygulamanın hata ayıklama kipinde açılmasını sağlar. Uygulama olması gerekenden çok daha yavaş çalışacaktır ama hata ayıklama gereçleri kodu takip edip olan hataların detaylarını raporlayabilmektedir. “Run Without Debugging” seçeneği ile uygulama其实te çalışması gereken şekilde çalışır ama hata ayıklama ve performans takip işlemlerini yerine getiremeyiz.

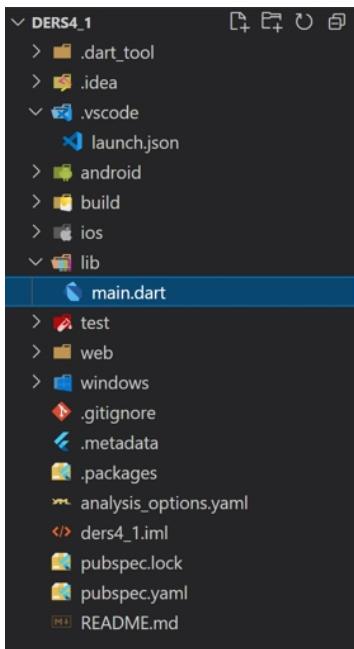
Uygulamayı çalıştırmak için Proje klasör yapısından lib klasörü içerisindeki main.dart dosyasını seçtikten sonra çalıştırabilirsiniz. Diğer bir alternatif de terminal penceresinde “**flutter run**” komutunu çağrımaktır. Uygulamanız çalıştığında aşağıdaki gibi bir ekranla karşılaşacaksınız.



Bu Flutter’ın şablon projesinin çalışan ekran görüntüsüdür. Flutter şablon olarak bir adet Foating Action Button (Sağ alt köşede mavi +), Başlık kısmı ve içerisinde + butonuna tıklandığı kadar birer birer artan bir sayaç bulunmaktadır. Tüm bu kodlar sadece lib\main.dart dosyası içerisindeindedir. Projenin bunun haricinde bir kod dosyası bulunmamaktadır. Hem arayüz hem de arka plan kodlaması aynı dart dosyasında yer almaktadır. İlerleyen örneklerde kodun hiyerarşik olarak düzenlenmesi için birden fazla dart dosyası kullanacağız.

4.2. Proje Klasör Yapısı

Projenin kodlarını incelemeye geçmeden önce şablon projemizdeki klasör ve dosyalara bir göz atalım. Şablon projede aşağıdaki proje klasörü görünümünü göreceksiniz.



Bu görünümde nokta ile başlayan klasör ve dosyalar genelde IDE’nin ve diğer araçların kendi ihtiyaçları için oluşturduğu klasör ve dosyalardır. Çoğunlukla bunlara müdahale etmemiz gerekmekz hatta bunlardaki yanlış bir satır IDE’nin projeyi açamamasına veya derleyememesine neden olabilir. Ama bazen de bu dosyalarda düzenleme yapmak gerekecektir. Örneğin **.vscode** klasörü adından anlaşılacağı üzere Visual Studio Code IDE’si için ayarları saklamaktadır. İçerisinde **launch.json** dosyası yer alır ve bu dosya IDE’nin uygulamayı nasıl ve ne ile çalıştıracağı ile ilgili ayarları saklar. Şu an için bu dosyada işlemimiz olmamıştır. IDE içerisinde de değiştireceğiniz ayarlar veya proje ile ilgili yaptığınız çalışma denemeleri bu dosyanın değişmesine yol açacaktır.

android, ios, web ve **windows** klasörleri adlarını taşıdıkları ortamlar için Flutter projesinin derlenmesinde gerekli sarıcı (**wrapper**) kodları ve ayarları içerirler. Eğer platforma özgü bir düzenleme gerekmiyorsa buradaki dosyalarla işimiz olmayacağıdır. Ama Google markete bağlanma, belli servisleri çalıştırma ve benzeri durumlarda bu klasörlerde (özellikle android ve ios) düzenleme yapmamız gerekecektir.

build klasörü Flutter ile derlenmiş uygulama dosyalarının bulunduğu klasördür. Farklı platformlar için derleme sonucu oluşan yükleme dosyaları burada yer olacaktır.

test klasörü uygulama için Unit test senaryolarının yazıldığı yerdir. Flutter şablon projesinde basit sayaç uygulaması için Unit testler de üretilmiş ve bu klasörde **widget_test.dart** altında saklanmıştır. Uygulamanın test edilmesi konusu işleneceği zaman bu dosya üzerinde farklı test senaryolarını nasıl gerçekleştireceğimize bakacağız.

lib klasörü uygulama kodlarının bulunduğu klasördür. Uygulamanın tüm aktif kodları bu klasör içerisinde yer almalıdır.

Diğer dosyalar arasında bizim için şu an en önemli **pubspec.yaml** dosyasıdır. Bu dosya Flutter uygulamasının temel konfigürasyon dosyasıdır. Uygulama ile ilgili yapılacak ayarlar bu dosya üzerinden belirtilir. Bir Flutter projesinin derlenebilmesi için kök klasörde **pubspec.yaml** dosyası olmak zorundadır.

4.3. Şablon Projedeki main.dart Dosyasına Bir Bakış

Şablon projedeki lib\main.dart dosyasının içeriğini inceleyelim.

```
import 'package:flutter/material.dart';
void main() {
  runApp(const MyApp());
}
```

Kod dosyası ‘package:flutter/material.dart’ ile tanımlı kütüphaneyi dahil etmekle başlamaktadır. Bu kütüphane neredeyse tüm örneklerimizde projelerimize dahil edeceğimiz bir kütüphanedir. Arayüz çizimi ile ilgili bileşenlerin Google Material Design kurallarına ve temalarına göre kullanımı için Flutter içerisinde gelen bir pakettir.

Sonraki satırda ise uygulamanın giriş noktası olan main fonksiyonu görülmektedir. Her Dart uygulaması global olarak tanımlanmış bir main fonksiyonuna sahip olmalıdır. Uygulama yaşam döngüsü main ile başlar ve main sonuna gelince biter. Flutter uygulamaları görsel bir arayüze sahiptir ve arayüz kapatılmadığı sürece çalışmaya devam eder. Bunu sağlamak için grafik arayüzünün (GUI) sürekli tazelendiği özel bir çalışma kipi gereklidir. Flutter’da bu runApp metodu ile sağlanmaktadır. runApp metodu bizden bir Widget örneği ister ve bunu uygulama arayüzüne çizmek için kullanır. Şablon projede MyApp adında bir Widget’ın yapıçı fonksiyonu çağrılarak bir örneği runApp metoduna beslenmiştir.

```
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Demo',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: const MyHomePage(title: 'Flutter Demo Home Page'),
    );
  }
}
```

```
class MyHomePage extends StatefulWidget {
  const MyHomePage({Key? key, required this.title}) : super(key: key);
  final String title;
  @override
  State<MyHomePage> createState() => _MyHomePageState();
}
```

```
class _MyHomePageState extends State<MyHomePage> {
  int _counter = 0;
  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text(widget.title),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            const Text(
              'You have pushed the button this many times:',
            ),
            Text(
              '_$_counter',
              style: Theme.of(context).textTheme.headline4,
            ),
          ],
        ),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        tooltip: 'Increment',
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

Devamında MyApp sınıfının tanımlanması ve onun içerisinde kullanılan MyHomePage sınıfının tanımlanması şeklinde kod devam etmektedir. Bu kod parçasını anlamak şu an için zor gelebilir. Şimdi biz sıfırdan bir merhaba dünya uygulaması ile başlayalım.

4.4. Merhaba Dünya Uygulaması

Şablon projedeki main.dart dosyasının içeriğini silerek aşağıdaki gibi düzenleyelim.

```
void main() {
  for (int i = 0; i < 5; i++) {
    print('Merhaba Dünya');
  }
}
```

Bu uygulama konsola 5 defa “Merhaba Dünya” yazacaktır ama GUI boş beyaz bir sayfa olarak gelecek ve uygulama kilitlenecektir. Bunun sebebi uygulamamızın bir arayüz çizebilmek için Flutter’ın beklediği runApp metodunu çağrırmamış olmasıdır. Yukarıdaki kod geçerli bir Dart programıdır ama Flutter çerçevesi ile uyumlu bir yapıda değildir. Bu noktada şunu tekrar hatırlatmakta fayda vardır. Flutter çerçevesi bir uygulama geliştirme dili değil bir GUI geliştirme çerçevesidir ve kendine özgü bazı kurallara dayanır. Dart ise cross-platform uygulama geliştirebileceğimiz zengin bir dildir ve yukarıdaki kod Dart diline göre uygun, derlenebilir bir koddur. Peki bizim uygulamamız neden kilitlenmektedir? Çünkü biz bir Flutter projesi oluşturduğumuzda gerekli sarıcı (wrapper) kodlar bir arayüz çizecek şekilde kurgulanmıştır ve bize buna uygun android, ios, web, windows klasörleri oluşturulmuştur. Eğer bu şekilde basit Dart uygulamaları yazıp denemek isterseniz çevrimiçi deneme yapabileceğiniz <https://dartpad.dev/> adresini kullanabilirsiniz.

4.5. Widget Kavramına Giriş

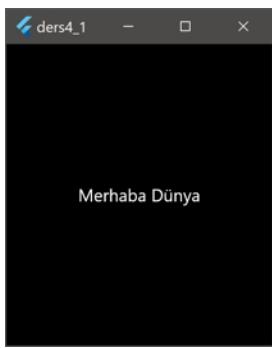
Flutter’da tüm arayüz bileşenleri **Widget** temel sınıfından türemektedir. Arayüzü oluşturan çerçeve, içerisindeki görünmeyen düzenleyici bileşenler ve görünen bileşenlerin hepsi **Widget** sınıfından miras alarak tanımlanmaktadır. Flutter’ın bu yaklaşımı GUI tasarımda çok esnek olmasını sağlamaktadır. Ayrıca çerçevenin karmaşık bir hiyerarşi içermesini ve öğrenme zorluğunu ortadan kaldırmaktadır. Yukarıdaki “Merhaba Dünya” örneğini Flutter uyumlu hale getirelim.

```
import 'package:flutter/material.dart';
void main() {
  runApp(
    const Center(
      child: Text(
        'Merhaba Dünya',
        textDirection: TextDirection.ltr,
      ),
    ),
  );
}
```

main fonksiyonunun içerisinde doğrudan runApp metodu çağrılmakta ve runApp'a parametre olarak bir Widget olan Center sınıfının yapıçı fonksiyonu beslenmektedir. Center sınıfı adından da anlaşılacağı üzere içerisindeki bileşenleri ekranın ortasına getirmek için kullanılan düzenleyici bileşenlerdir. Center sınıfı arayüzde görünmez ve kullanıcı ile etkileşimi yoktur.

Arayüzde görselliği olan bileşenler dışında her Widget içerisinde child veya children adında bir parametre bulunur. Bu parametre sayesinde bir ağaç formunda Widgetlar iç içe tanımlanır. Arayüz çizimi bu ağacın taranması ve gerekli hesaplamaların yapılması sonucu dinamik olarak sağlanır. Fark edeceğiniz üzere örnek uygulamamızda GUI bileşenlerinin nerede duracağı ile alakalı bir piksel veya ölçü verisi kullanılmamıştır. Flutter GUI bileşenlerinin özelliklerine ve birbirleri ile etkileşimlerine göre arayüzü dinamik olarak oluşturur. Bu sayede farklı ekran genişlikleri ve modellerinde aynı görselliği tek bir kod üzerinden üretebilir. Bu açıdan Flutter pek çok alternatifine göre kod yazma sürecini azaltmaktadır.

Center yapıçı fonksiyonu içerisinde isimlendirilmiş **child:** parametresinin kullanımını görmekteyiz. Bu parametreye görsel bir GUI bileşeni (Yani bir Flutter Widget’ı) olan Text sınıfının yapıçı fonksiyonu beslenmiştir. **Text** sınıfının yapıçı fonksiyonu da parametre olarak gösterilecek metni String olarak, metnin hangi yöne (ltr: left to right: Latin dilleri için soldan sağa) yazılacağı bilgilerini almaktadır. Flutter çoklu dil desteği olan bir çerçeve olduğu için arayüzdeki metin gösteriminde yön bilgisini sormaktadır. Şablon projede kullanılan Material teması sayesinde bu bilgiyi vermek zarunda kalmadığımızı sorgulayabilirsiniz. Sonuca aşağıdaki gibi bir uygulama ekranına sahip oluruz.



Şablon projedeki mavi başlık ve beyaz arka plan yerine siyah arka planda ortalanmış “Merhaba Dünya” yazısını görmekteyiz. Material Design’da yönelik gerekli Widget tanımlamaları (MaterialApp) bu kodda olmadığı için Material Design’ın temel Mavi-Beyaz görünümü bu uygulanmamıştır.

4.6. Temel Widget’lar

Flutter arayüz bileşenleri (Widget) açısından oldukça zengin bir içerik sunar. Kendi arayüz bileşenlerimizi Widget’tan miras alarak oluşturabileceğimiz, üçüncü parti paketlerle sunulan arayüz bileşenlerini kullanabileceğimiz gibi Flutter neredeyse tüm arayüzde çizim ihtiyaçlarını karşılayacak kadar çok hazır Widget türevi ile gelmektedir. Bunları Widget Catalog bölümünden inceleyebilirsiniz.

Flutter’ın sunduğu Widgetları iki türde ele alabiliriz. Bunlardan ilki görsel Widgetlar ikincisi ise düzenleyici Widgetlardır. Görsel Widgetlar arayüzde bir hacme sahip kullanıcı ile etkileşen, yer tutan veya bilgi verme amaçlı Widgetlarken düzenleyici Widgetlar arayüzdeki diğer bileşenlerin birbiri ile olan etkileşiminini ayarlarlar. Örneğin Text görsel bir Widget iken Center düzenleyici bir Widgettir.

En fazla kullanacağımız Widgetlara aşağıdakileri örnek verebiliriz;

Text: Ekrana bir metin yazdirmak için kullanacağımız görsel Widgettir.

Row, Column: Ekrana yerleşecek bileşenlerin dikey (Column) ve yatay (Row) düzende sıralanmasını sağlayan düzenleyici Widgetlardır.

Stack: Altındaki elemanların (children) üst üste yerleştirilmesini sağlayan düzenleyici bir Widgettir. Stack mantık olarak web geliştirmedeki absolute yerleşimine denk gelmektedir.

Container: İçerdiği Widget silsilesi üzerinde çeşitli yerleşim ayarları için kullanılır. Web geliştirmedeki div kullanımına benzer bir düzenleyici Widget’tır. Margin, padding, constraints gibi özelliklerle içeriği Widgetların nasıl yerleşeceği ayarlanır.

4.7. Kendi Arayüz Bileşenimizi Oluşturalım

Flutter arayüz tasarımlı için zengin bir Widget kütüphanesi ile gelmiş olsa da bazen kendi arayüz bileşenimizi oluşturmak isteyebiliriz. Flutter’ın ekran çizme modeli ekrandaki bölümlerin bu şekilde alt kümelere bölünmesini ve her alt kümeyi kendi içerisinde bir dinamiği olmasını sağlamaktadır. Diğer bir deyişle belli görevi gerçekleştirecek bir arayüz bileşenini gerekli arka plan koduyla beraber bütünsel olarak tasarlayabiliriz. Ayrıca bu sayede arayüzde tekrar eden aynı görsel bileşenlerin bir kez kodlanması ve kodun yönetiminin kolaylaştırılması sağlanır. Kendi arayüz bileşenlerimizi oluşturmak için Flutter’daki Widget sınıfından miras almamız gerekmektedir. Widget sınıfından doğrudan miras almak yerine Flutter’da Widgettan miras yoluyla tanımlanmış iki sınıfтан (StatelessWidget, StatefulWidget) birinden miras alarak kendi arayüz bileşenimizin sınıfını oluştururuz.

4.7.1. StatelessWidget

Kendi arayüz bileşenimizi oluşturmak için en basit yöntem StatelessWidget’tan miras alarak bir sınıf tanımlamaktır. **StatelessWidget**, kullanıcı ile etkileşime girmeyecek veya kullanıcı etkileşimi ile arayüzün

çizimi tetiklenme ihtiyacı olmayacak durumlar için tercih edilir. Genel olarak arayüzün yenilenmesi için belli bir tetikleme gerekmiyorsa StatelessWidget ile arayüz bileşeni türetmek tercih edilir. Alternatifi StatefulWidget tanımlamasına göre daha basit yapıdadır ve sistemi daha az yorar. StatefulWidget sınıfından miras aldığınızda sizi Widget türünde dönüş değeri olan build adındaki fonksiyonu yeniden yazmaya (override) etmeye zorlar. build fonksiyonu arayüz çizimi için çağrılan fonksiyondur ve döndüğü Widget değeri arayüzdeki bileşeni ifade eden bir referanştır.

```
class GreenFrog extends StatelessWidget {  
  const GreenFrog({ Key? key }) : super(key: key);  
  @override  
  Widget build(BuildContext context) {  
    return Container(color: const Color(0xFF2DBD3A));  
  }  
}
```

Yukarıdaki örnekte GreenFrog, StatelessWidget sınıfından miras alarak türetilen bir arayüz bileşenidir. Her arayüz bileşeninin build adındaki metodu Flutter çerçevesine ekranı nasıl çizeceğini bildirmektedir ve dönüş değeri bir arayüz elemanı olacak şekilde Widget türündedir. Bu yapı Flutter'ın kendi sağladığı Widget kataloğundaki Widget'ları kendi oluşturduğumuz arayüz bileşenlerinde çağrıma ve benzer şekilde tersini de mümkün kılmaktadır.

build metodu içerisinde arayüz çizimi ile ilgili gerekli düzenlemeler yapılarak en son arayüzde çizilmek istenen kısım geri dönülür. Bu örnekte doğrudan Flutter'ın sunmuş olduğu bir Container Widget'i doldurulmuştur. Dikkatli bakınca aslında içini dolduran rengi belirlemiş olduğumuz bir özelleşmiş Container yapısı oluşturmuş ve adını GreenFrog koymuş olduk. build metodu parametre olarak çizim süreci ile ilgili ortam bilgisini taşımak için context adında bir parametre alır. Bu parametreyi ileriki örneklerde kullanacağız. Yine key adındaki parametrenin alınarak miras alınan StatelessWidget yapıcı fonksiyonuna iletildiğini görmekteyiz. Key arayüzdeki bileşenleri ayırt etmek için kullanacağımız bir değerdir. Sonraki örneklerde kullanımını göreceğiz.

Aynı yapıda kullandığımız renk ve Widget'i yerel değişken olarak saklayan aşağıdaki örneği de yazabiliriz. Burada Frog yapıcı fonksiyonu aracılığı ile renk ve Widget verilerini dış ortamdan alarak kendi bünyesinde birleştirmektedir. Daha esnek bir yapı oluşturulmuştur.

```
class Frog extends StatelessWidget {  
  const Frog({  
    Key? key,  
    this.color = const Color(0xFF2DBD3A),  
    this.child,  
  }) : super(key: key);  
  final Color color;  
  final Widget? child;  
  @override  
  Widget build(BuildContext context) {  
    return Container(color: color, child: child);  
  }  
}
```

Dikkatinizi çekecek bir nokta color ve child (alt bileşen için Flutter'da kullanılan genel jargondur) için final anahtar kelimesi kullanılmıştır. Yani bunlar Frog'un yapıcı fonksiyonunda atandıktan sonra değiştirilemezler. Arayüz bileşenlerinde bu sabit bileşen düzeni hep kullanılacaktır. Aksini düşünürsek arayüze çizilen bir bileşenin sonradan değiştirilmesi Flutter çerçevesinin kararsız arayızlar üretmesine neden olur. Eğer arayüzde bir bileşen değişecekse ilgili yerden başlayarak içe doğru tüm bileşenlerin yeniden oluşturulması ve build metodlarının üzerinden geçilmesi gereklidir. Flutter çerçevesinin hızı bunu fark ettirmeden yapmaktadır. Şunu da unutmamak gereklidir build içerisinde yazılacak kodların 16ms'de tamamlanması gereklidir. Bu değer 60 fps görüntü tazeleme için gerekli limitidir. Test kısmında arayüz tepkime sürelerinin nasıl inceleneceği anlatılacaktır.

4.7.2. StatefulWidget

StatefulWidget, StatelessWidget'ın yapamadığı durum (state) takibi için özelleşmiş bir yapı sunar. Flutter mantık olarak arayüz çiziminin gerektiği zamanlarda yenilemeye ve Widget ağacını baştan çizmektedir. Bu durumun tetiklenmesi için ya arayüz bileşenlerinin belli bir noktadan sonrasının yeniden oluşturulması (build metodu çağrıısı) ya da durumunda bir değişiklik olması gerekmektedir. StatefulWidget bunu çok basit bir şekilde kendi türünde jenerik bir State<> objesini oluşturmak için createState() metodunu aşırı yükleyerek sağlamaktadır.

Durum değişikliğinde arayüz çiziminin yenilenmesi sağlanacaktır. Bunun için State<> objesi içerisinde işlemelerde hangi aşama bitince arayüzün çizilmesi isteniyorsa o bölge setState() ile çevrilir.

```
class Bird extends StatefulWidget {
  const Bird({
    Key? key,
    this.color = const Color(0xFFFFE306),
    this.child,
  }) : super(key: key);
  final Color color;
  final Widget? child;
  @override
  _BirdState createState() => _BirdState();
}

class _BirdState extends State<Bird> {
  double _size = 1.0;
  void grow() {
    setState(() { _size += 0.1; });
  }
  @override
  Widget build(BuildContext context) {
    return Container(
      color: widget.color,
      transform: Matrix4.diagonal3Values(_size, _size, 1.0),
      child: widget.child,
    );
  }
}
```

Bu örnekte Bird sınıfı StatefulWidgettan miras alınarak oluşturulmuştur. Bird içerisinde durum takibi için createState() metodunu aşırı yüklenerek _BirdState türünde bir durum sınıfı oluşturulur. StatefulWidget tanımlamasında eğer setState metodunu tetiklenirse durum değişikliği olacak durum içerisindeki build metodunu yeniden tetiklenerek arayüz bileşenleri baştan çizilecektir. grow() adındaki metodun çağrılması bu örnekte bir durum değişikliği oluşturur.

StatefulWidget kullanırken dikkatli olunmalıdır. Gerekmedikçe StatefulWidget tanımlanmamalıdır. Eğer bir kullanıcı etkileşimi sonucu arayüzün değişmesi gerekiyorsa StatefulWidget ve setState'in konumu Widget ağacında değişikliğin olması gereği en üst yerde olmalıdır. Widget ağacında daha üst seviyede StatefulWidget barındırmak ekranada değişimmeyecek bileşenlerin baştan çizilmesini tetikleyeceği için performans kaybına neden olur.

4.8. Tam Bir Mobil Uygulama Arayüzü Örneği

Aşağıdaki örnekte Android veya iOS içerisinde görmeye alışkin olduğunuz başlık ve içerik kısımlarından oluşan tam bir uygulama arayüzü oluşturalım.

```
import 'package:flutter/material.dart';
class MyAppBar extends StatelessWidget {
  const MyAppBar({required this.title, Key? key}) : super(key: key);
  // Widget içerisinde tanımlı alt elemanlar daima final olmalıdır.
  final Widget title;
  @override
  Widget build(BuildContext context) {
    return Container(
      height: 56.0, // mantıksal piksel formunda
      padding: const EdgeInsets.symmetric(horizontal: 8.0),
      decoration: BoxDecoration(color: Colors.blue[500]),
      child: Row(
        children: [
          const IconButton(
            icon: Icon(Icons.menu),
            tooltip: 'Navigation menu',
            onPressed: null, // şu an tıklama desteklemiyor
          ),
          Expanded(
            child: title,
          ),
          const IconButton(
            icon: Icon(Icons.search),
            tooltip: 'Search',
            onPressed: null,
          ),
        ],
      ),
    );
  }
}
```

```

class MyScaffold extends StatelessWidget {
  const MyScaffold({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return Material(
      child: Column(
        children: [
          MyAppBar(
            title: Text(
              'Example title',
              style: Theme.of(context) //.
                .primaryTextTheme
                .headline6,
            ),
          ),
          const Expanded(
            child: Center(
              child: Text('Hello, world!'),
            ),
          ),
        ],
      );
    );
  }
}

void main() {
  runApp(
    const MaterialApp(
      title: 'My app', // işletim sisteminin kullandığı isim.
      home: SafeArea(
        child: MyScaffold(),
      ),
    ),
  );
}

```

main fonksiyonundan başlayarak örneğimizi inceleyelim. Flutter çerçevesinin çalışabilmesi için uygulama runApp metodu çağrısına bir Widget objesini (burada MaterialApp) parametre olarak vererek başlamaktadır.

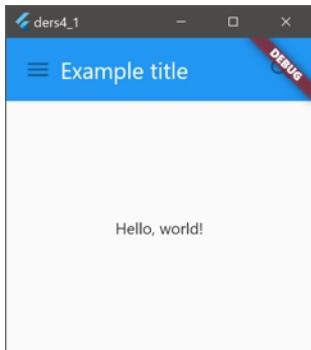
MaterialApp, ‘package:flutter/material.dart’ kütüphanesi ile projede aktif edilen Flutter çerçevesinin “Material Design” temasını uygulamada kullanmayı sağlayan bir bileşendir. Kökte olmasının sebebi ilgili temanın Widget ağıacı içerisinde oluşturulan tüm alt elemanlara uygulanmasını sağlamaktır. MaterialApp kendine özgü title (başlık), home (ev) gibi isimlendirilmiş parametreleri vardır. home parametresi arayüz çizimi için verilecek Widgetı aldığı parametredir. Burada başka bir Flutter Widgetı olan SafeArea kullanımını görmekteyiz.

SafeArea farklı telefon modelleri ve işletim sistemleri için arayüzde görülemeyen alanlar olmasını engellemektedir. Bu nedenle Widget ağıacı seviye olarak görsel bileşenler başlamadan gelmesi gereklidir. SafeArea, çoğu Widgettan aşina olduğumuz child isimlendirilmiş parametresini kullanmaktadır. child parametresine arayüz çizimini oluşturmak için bizim hazırladığımız MyScaffold türünde bir obje beslenmektedir.

Alt eleman içerecek Widgetların parametre listelerinde eğer tek alt elemana izin veriyorsa child, birden fazla alt elemana izin veriyorsa children olarak isimlendirilmiş bir parametre tanımlanır. child parametresi tür olarak Widget, children parametresi de tür olarak Widget türü bir dizi <Widget>[] almaktadır.

Kod içerisinde Material, Expanded, IconButton Widgetları kullanılmıştır. Material, Material Design temasının kullanımı için kavrayıcı bir Widgettir. Expanded, flex adlı özelliğine göre ekranda kalan boşluğu paylaştırır. IconButton ise tıklanabilir ikon oluşturmak için yine Flutter Widget kataloğu bulunan bir Widgettir.

Çalışan uygulama aşağıdaki gibi bir ekran oluşturacaktır. Burada üst bar da menü ve ara butonları Material kütüphanesindeki ikonlardan getirilmiştir. Uygulama arayüzünde kendi görsellerinizi oluşturmak yerine genel geçer görselleri kullanmak kullanıcıların uygulamaya daha hızlı uyum sağlayacaktır.



4.9. Material Kütüphanesindeki Bileşenleri Kullanma

Bir önceki örnekte oluşturduğumuz MyScaffold ve MyAppBar yerine Material kütüphanesinde sunulan hazır Widgetları kullanabiliriz. Flutter Widget kataloğu bu açıdan zengin bir içerik sunmaktadır. Material kütüphanesi Android işletim sisteminden aşina olduğumuz görselliği bize hızla sunacaktır. Flutter, iOS için de Cupertino kütüphanesinde benzer Widgetları iOS görselliği ile sunmaktadır.

```
import 'package:flutter/material.dart';

void main() {
  runApp(
    const MaterialApp(
      title: 'Flutter Tutorial',
      home: TutorialHome(),
    ),
  );
}

class TutorialHome extends StatelessWidget {
  const TutorialHome({Key? key}) : super(key: key);
```

```
@override
Widget build(BuildContext context) {
    // Scaffold is a layout for
    // the major Material Components.
    return Scaffold(
        appBar: AppBar(
            leading: const IconButton(
                icon: Icon(Icons.menu),
                tooltip: 'Navigation menu',
                onPressed: null,
            ),
            title: const Text('Example title'),
            actions: const [
                IconButton(
                    icon: Icon(Icons.search),
                    tooltip: 'Search',
                    onPressed: null,
                ),
            ],
        ),
        // body is the majority of the screen.
        body: const Center(
            child: Text('Hello, world!'),
        ),
        floatingActionButton: const FloatingActionButton(
            tooltip: 'Add', // used by assistive technologies
            child: Icon(Icons.add),
            onPressed: null,
        ),
    );
}
```

Birçok zaman kendi Widgetlarımızı oluşturmak yerine Flutter'ın bize sunduğu hazır Widgetları kullanmak daha hızlı geliştirme yapma ve görsel açıdan daha kaliteli arayüzler oluşturmamızı sağlar. Material bileşenlerini kullanan bu örnekte dikkat edilirse başlıkta gölge efekti bulunmaktadır. Hazır bileşenlerde bunun gibi pek çok özellik vardır ve ayarlanabilir olarak sunulmaktadır.

4.10. Kullanıcı Etkileşimi Sağlamak

Flutterda kullanıcı etkileşimlerini yakalayacak IconButton, ElevatedButton, FloatingActionButton, gibi hazır widgetlar bulunmaktadır. Bu widgetlarda onPressed adındaki parametre tıklama işlevini yerine getirecek fonksiyonla beslenerek basitçe bu işlev sağlanmaktadır. Önceki örneklerimizde IconButton için bu değeri işlevsiz bırakmak için null ile beslemiştik.

Kendi oluşturacağımız widgetlar için de benzer kullanıcı etkileşimi tanımlamak için GestureDetector adındaki widgeti kullanmamız gereklidir. Kullanıcının etkileşime geçeceği arayüz objelerini bir GestureDetector objesi ile sarmalayarak gereken etkileşim işlevlerini GestureDetector üzerinden sağlayız.

```
import 'package:flutter/material.dart';
class MyButton extends StatelessWidget {
  const MyButton({Key? key}) : super(key: key);
  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: () {
        print('MyButton was tapped!');
      },
      child: Container(
        height: 50.0,
        padding: const EdgeInsets.all(8.0),
        margin: const EdgeInsets.symmetric(horizontal: 8.0),
        decoration: BoxDecoration(
          borderRadius: BorderRadius.circular(5.0),
          color: Colors.lightGreen[500],
        ),
        child: const Center(
          child: Text('Engage'),
        ),
      ),
    );
  }
}
```

```
void main() {
  runApp(
    const MaterialApp(
      home: Scaffold(
        body: Center(
          child: MyButton(),
        ),
      ),
    ),
  );
}
```

Bu örnekte MyButton adındaki bizim tanımladığımız widget için onTap (üzerine dokunma) işlevini build metodu içerisinde yerleştirdiğimiz GestureDetector widget sayesinde uygulamaktayız. GestureDetector da child parametresi olarak ekran çizimi için kullanılacak alt bileşenleri almaktadır.

Kullanıcı etkileşimi için tam bir işlev listesine Flutter sitesindeki Gestures bölümünden erişebilirsiniz.

4.11. Flutter Uygulama Örnekleri

Flutter.dev İnternet sitesinde farklı özelliklerini incelememiz için birçok örnek sunulmaktadır. Bu noktaya kadar anlatılan içeriklerin hepsini bir araya getiren ShoppingList örneğini Flutter | Introduction to widgets bölümünde “Bring It All Together” başlığında inceleybilirsiniz.

Flutter örnekleri Gallery, GitHub Repo, Cookbook, Codelabs ve Tutorials başlıkları altında sunmaktadır. Bunları geldiğiniz seviyeye göre yer yer inceleyeceğiz. Tüm listeye ulaşmak için Flutter dokümantasyon sayfasındaki “Samples&tutorials” başlığını inceleyebilirsiniz.

Bölüm Özeti

Flutter çerçevesi arayüz oluşturmak için Widget adı verilen sınıfından türetilmiş objeler kullanmaktadır. Widgetların en büyük avantajı iç içi geçerek bir arayüz ağacı oluşturabilmeleri ve arayüz için başka bir kök kavram olmamıştır. Yani Flutter'da her şey bir Widgettir. Flutter zengin bir Widget kataloğu ile gelmektedir. Bunun dışında StatelessWidget veya StatefulWidget sınıflarından miras alarak kendi özel Widget sınıfımızı da oluşturabilmekteyiz. StatelessWidget basit bir arayüz düzenlemesi sağlarken StatefulWidget kullanıcı etkileşimi sonucu yeniden çizim gerektiren durumlarda çözüm sunmaktadır. Kullanıcı etkileşimlerini yakalamak için kendi oluşturacağımız Widgetlarımızda GestureDetector kullanırız. Flutterda tanımlı hazır widgetlarda GestureDetector kullanılmaktadır ve bize işlev için onPressed, onTap gibi beslememiz gereken parametreler sunmaktadır. Flutter hakkında ek örnekler için Flutter.dev sayfasındaki dokümanlar kısmında yönlendirmeler mevcuttur. Bunların bir kısmı eğitim materyali içerisinde kullanılırken bir kısmı sadece dilin özelliklerini ifade etmek için GitHub repolarında verilmektedir.

Kaynakça

<https://flutter.dev/docs/development/ui/widgets-intro>

<https://api.flutter.dev/flutter/widgets/Widget-class.html>

<https://api.flutter.dev/flutter/widgets/Widget-class.html>

<https://api.flutter.dev/flutter/widgets/SafeArea-class.html>

5. FLUTTER'DA ARAYÜZ DÜZENLEME

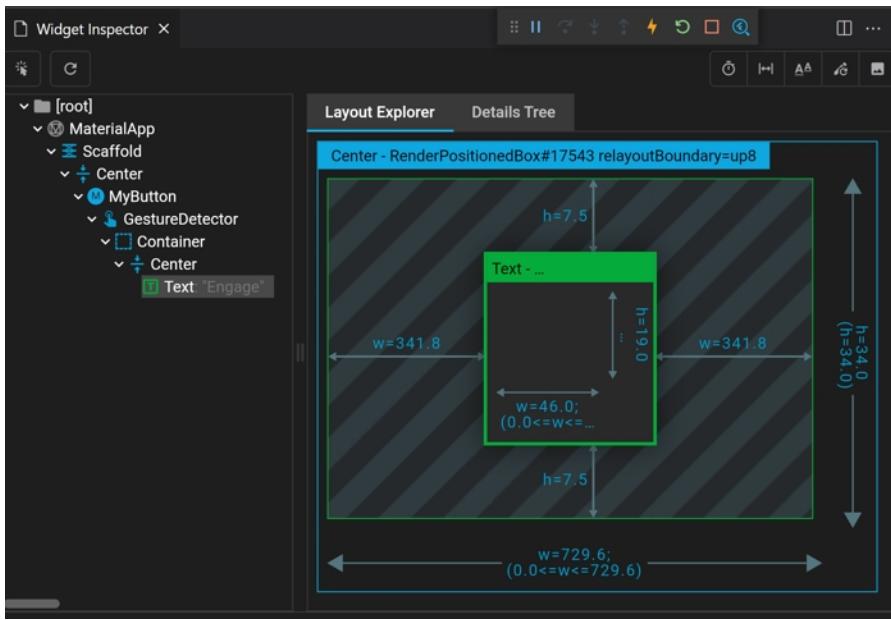
Giriş

Bu bölümde Flutter'ın arayüz yönetim modeli anlatılacaktır. Flutterdaki arayüz yerleşimi web tasarıma benzemektedir. Arayüzde nesnelerin yerleşimi diğer platformlardan farklı gelebilir. Flutter arayüz tasarımı için doğrudan bir araç sunmamaktadır. Bunun nedeni olarak arayüzün nasıl görüneceğini Flutter tarafından çalışma zamanında karara bağlanmasıdır diyebiliriz.

5.1. Arayüz Düzeni

Flutter'da her objenin bir Widget olduğunu söyledik. Widgetları iki kümeye ele alabiliriz. İlk görsel olan Text, Image, Icon gibi widgetlar, ikincisi ise Row, Column, Container gibi düzenleyici widgetlardır. Arayüzü oluştururken tüm bu widgetlar bir ağaç şeklinde iç içe geçerek arayüzde istenilen düzen oluşturulur. Görselliği olmayan widgetlar **layout** (düzen) Widgetları olarak adlandırılırlar. Genellikle layout widgetları ağaç içerisindeki hiyerarşiyi düzenlerken görsel widgetlar ağaç yapraklarını oluşturur.

Tasarımınızda nesnelerin ağaç içerisindeki yerini ekrandaki konumlanması için **“DartDevTools”** kullanabiliriz. **“Widget Inspector”** ile ekranı oluşturan widget hiyerarşisini hem ağaç hem de yerleşim olarak inceleyebiliriz.



Aşağıdaki görseli oluşturacak bir widget ağacını nasıl oluşturacağımıza bakalım.



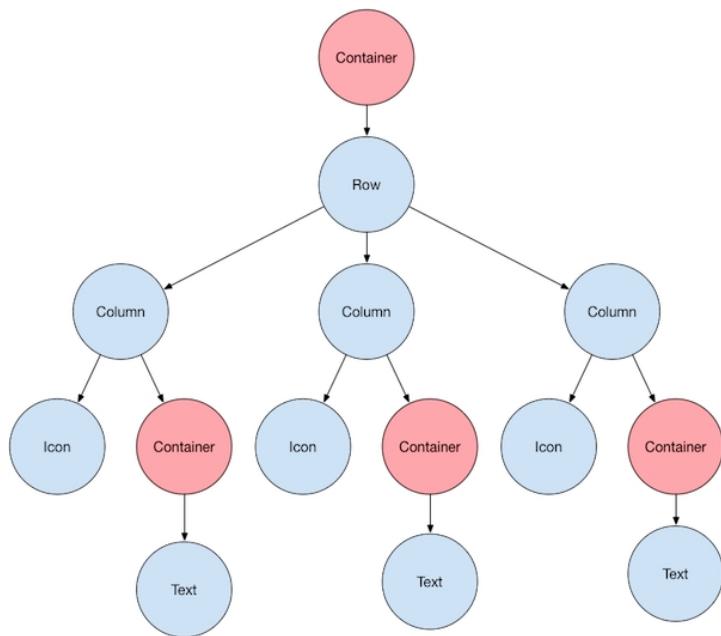
Widget Inspector üzerinden “**Overlay Guidelines**” seçimini aktif ederseniz görülmeyen widgetların ekrandaki konumlarını işaretleyebilirsiniz.



Yukarıdaki ikon ve metinlerden oluşan görselin tasarım çizgileri aşağıdaki gibi çizilmektedir. Ekran yatayda eşit aralıklla 3 parçaaya bölünmüştür. Bunun için birden fazla alt widget barındıran Row widgeti kullanılır. Bu üç parçanın her biri de bir ikon ve bir metni alt alta göstermektedir. Bunun için de her birinde bir Column widget kullanılır.



Bu hiyerarşinin ağaç şeklinde gösterimi aşağıdaki şekildedir. Container tek alt widget (child) alabilen bir düzenleyici, Row ve Column ise birden fazla alt widget (children) alabilen düzenleyicilerdir. Icon ve Text görsel özelliği olan ve alt widget almayacak widget türleridir. Container çokça başvuracağımız düzen widgetlarındandır. Arayüzdeki nesnelerin yerleşimi arkaplan renklendirmesi, kenarlıkların düzenlenmesi, kenar boşlukları gibi özellikleri ayarlanır.



5.2. Adım Adım Arayüz Elemanlarını Oluşturalım

Arayüz tasarımda önce kağıt üzerinde eskiz çalışması yapılmalıdır. Görsel olarak bulunacak nesnelerin nasıl konumlanacağına karar verdikten sonra widget ağaçları tepeden aşağıya doğru düzenlenir. Şimdi olayı basitten genele anlamak için tersi bir bakış açısı ile önce görsel elemanları sonra bunları birleştiren düzen widgetlarını oluşturalım.

Ağacın yapraklarındaki Icon ve Text widgetları basitçe aşağıdaki gibi kodlanmaktadır.

```
Text('Hello World'),
```

```
Icon(
  Icons.star,
  color: Colors.red[500],
),
```

Bu görsel elemanların düzenleyici bir widget ile sarmalanmasına aşağıdaki kodu örnek verebiliriz. Bu örnekte bir metin ekranı ortalamak için Center widgetine child olarak verilmiştir.

```
const Center(
  child: Text('Hello World'),
),
```

Daha üst seviyede bu düzen widgeti eğer başka bir düzen widgeti ile sarmalanacaksa onun child veya children parametresi olarak verilecektir. Eğer kendi oluşturacağımız bir widget içerişine ekleneneceğse build metodu içerisinde yer olacaktır. Doğrudan Material bir sayfaya yerleştirilecekse buna uygun parametreye beslenmelidir. Örneğin Scaffold widgeti için bu body parametresidir. Aşağıda bu kullanımlar örnek MyApp yapısı altında MaterialApp -> Scaffold -> Center -> Text silsilesinde bir yerleşim verilmektedir.

```
class MyApp extends StatelessWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter layout demo',
      home: Scaffold(
        appBar: AppBar(
          title: const Text('Flutter layout demo'),
        ),
        body: const Center(
          child: Text('Hello World'),
        ),
      ),
    );
  }
}
```

Material tasarımlı ve Material tasarımlı olmayan örnekleri repodan inceleyebilirsiniz.

5.3. Yatay ve Dikey Düzende Arayüz Bileşenlerini Yerleştirme

Arayüz bileşenlerini yatay olarak yerleştirmek için Row, dikey olarak yerleştirmek için Column düzen widgetları kullanılır. Bu düzen widgetları elemanların nasıl duracağı ile ilgili mainAxisAlignment ve CrossAxisAlignmentAlignment özellikleri bulunmaktadır.

```
Row(
  mainAxisAlignment: MainAxisAlignment.spaceEvenly,
  children: [
    Image.asset('images/pic1.jpg'),
    Image.asset('images/pic2.jpg'),
    Image.asset('images/pic3.jpg'),
  ],
);
```

Bu örneğimiz yatayda üç resmi **spaceEvenly** seçeneği ile eşit aralıklı olarak yerleştirmektedir. Resim dosyalarının projeye dahil olması için pubspec.yaml dosyasında resim kaynağının belirtilmesi gerekmektedir. Buna sonraki ders bakacağınız. Row-Column kullanımıyla ilgili tam örnek GitHub row_column adresinden incelenebilir.

Bazen kullandığınız görseller arayüze belirlenen alana sığmayacak kadar büyük olabilir. Bu gibi durumlarda görselin yeniden boyutlandırılması ve alana uygun hale getirilmesi için **Expanded** widgeti kullanılabilir. Flutter ekrana sığmayan bölümlerde sarı-siyah uyarı bandı göstermektedir.

```

Row(
  mainAxisAlignment: MainAxisAlignment.center,
  children: [
    Expanded(
      child: Image.asset('images/pic1.jpg'),
    ),
    Expanded(
      child: Image.asset('images/pic2.jpg'),
    ),
    Expanded(
      child: Image.asset('images/pic3.jpg'),
    ),
  ],
);

```

Bu örneğin tam hali GitHub sizing adresinden incelenebilir.

Expanded widgetları aynı üst bileşen içerisindeki alanı eşit bölüşürler ve boş alana yayırlar. Eğer bu bölüşmenin belirli oranlarda olmasını isterseniz **flex** değeri ile bunu ayarlayabilirsiniz. Flex değeri varsayılan olarak bir alınır. Yukarıdaki örnekte ortadaki resmin yatayda diğerlerinden iki kat fazla yer kaplamasını istiyorsak kodu şu şekilde güncelleyebiliriz.

```

Row(
  mainAxisAlignment: MainAxisAlignment.center,
  children: [
    Expanded(
      child: Image.asset('images/pic1.jpg'),
    ),
    Expanded(
      flex: 2,
      child: Image.asset('images/pic2.jpg'),
    ),
    Expanded(
      child: Image.asset('images/pic3.jpg'),
    ),
  ],
);

```

Row, Column widgetları normalde kullanabildikleri alana tam olarak yayırlar. Yani Row en üst seviyede kullanılıyorsa ekranın solundan sağına kadar olan alanın tamamını temsil edecektir. Row için **mainAxisSize** isimlendirilmiş parametresine mainAxisSize.min değeri atanarak düzenleyici widgetin sadece içerisindeki görsel elemanlar kadar yer kaplaması sağlanır. GitHub pavlova adresinde bununla ilgili örnek bulunmaktadır.

Pavlova örneği aşağıdaki görseli sağlamaktadır.



Bu görselin oluşmasında Row ve Column widgetları iç içe yerleştirilerek ekrandaki görsel objelerin birbirlerine göre yerleşimleri ayarlanmıştır. Bu düzenlemeyi web sayfası geliştirmedeki div tabanlı düzenlemeye benzetebiliriz. Flutter'da iç içe gelecek parçalar için oluşan ağaç belli alt kümelerde oluşturup değişkenlere atayabilir veya sonradan çağrılabilecek fonksiyonlar olarak yazabilir. Pavlova örneğindeki aşağıdaki kod parçalarında bunlar örneklenmiştir.

```
var stars = Row(
  mainAxisSize: MainAxisSize.min,
  children: [
    Icon(Icons.star, color: Colors.green[500]),
    Icon(Icons.star, color: Colors.green[500]),
    Icon(Icons.star, color: Colors.green[500]),
    const Icon(Icons.star, color: Colors.black),
    const Icon(Icons.star, color: Colors.black),
  ],
);

final ratings = Container(
  padding: const EdgeInsets.all(20),
  child: Row(
    mainAxisAlignment: MainAxisAlignment.spaceEvenly,
    children: [
      stars,
      const Text(
        '170 Reviews',
        style: TextStyle(
          color: Colors.black,
          fontWeight: FontWeight.w800,
          fontFamily: 'Roboto',
          letterSpacing: 0.5,
          fontSize: 20,
        ),
      ),
    ],
  ),
);
```

stars adındaki değişken, bir yıldız ikon kümesi tutmakta ve bu küme ratings adlı başka bir değişkende yapılan tasarımda kullanılmaktadır.

5.4. Genel Arayüz Düzenleyici Bileşenler

Flutter arayüz düzeni için çok farklı widget seçenekleri sunmaktadır. Bunları kendi kullanım alışkanlıklarınız, performans gereksinimleri veya görsellikle ilgili ihtiyacı karşılaması açısından değerlendirebilirsiniz. Bu noktada en büyük sorun belki de aynı arayüzü çok farklı widget ağaçları oluşturarak da düzenleyememizdir. Bu açıdan arayüz düzenleyici bir widget seçeceğinizde widget kataloğuundan ve ilgili widgetin nasıl kullanıldığı ile ilgili API dokümantasyonunu incelemekte faydalıdır.

Şimdi genel amaçlı en fazla kullanacağımız düzenleyici arayüz bileşenlerine bakalım.

5.4.1. Standart Düzenleyici Widgetlar

Container: İç boşluk(padding), dış boşluk(margin), kenarlık(border), arkaplan rengi(background color) gibi özellikleri atamayı sağlar.

GridView: Kaydırılabilir bir ızgara (tablo) görünümü sağlar.

ListView: Kaydırılabilir bir liste görünümü sağlar

Stack: İçeriğin üst üste bindirilmesini sağlar.

5.4.2. Material Düzenleyici Widgetlar

Card: Önemli noktaları vurgulamak için köşeleri yuvarlatılmış ve gölge efekti olan bir çerçeve sağlar.

ListTile: İçerisinde 3 satır yazı alanı olan ve istege göre başlık ve dip ikonları konulabilen bir satır sağlar.

Burada verilen düzenleyici arayüz bileşenlerinin kullanımı ile ilgili örnekler; container, grid_and_list, card_and_stack adreslerinde bulunmaktadır.

5.5. Responsive(Esnek) ve Adaptive(Uyabilen) Uygulama Geliştirme

Mobil platformlar için uygulama geliştirmede karşımıza çıkan en önemli arayüz tasarım kriterleri esnek ve uyum sağlayabilen uygulamalar geliştirmektir. Flutter'ın geliştirilmesinde de bu hususlar ön planda tutulmuştur.

Responsive(Esnek) bir uygulama demek platformdaki durum, konfigürasyon değişikliklerine uygulamanın karşılık vererek ekranı kullanıcının maksimum konfor alabileceği şekilde sunmasıdır. Bunun en başında telefonun dik konumdan yatay konuma çevrilmesinde arayüz bileşenlerinin düzenlerinin değiştirilmesini örnek verebiliriz.

Adaptive(Uyabilen) bir uygulama ise farklı işletim sistemleri ve farklı donanımlar için aynı kullanıcı deneyimini yaşatacak şekilde çalışabilen bir uygulama demektir. Örneğin çok geniş ekranlı ama düşük çözünürlüklu bir tablette kullanılan uygulamanın yüksek çözünürlüklu ama dar bir telefonda kullanılırken görsellerin kullanıcının etkileşim sağlayabileceği boyutta ve çözünürlükte olması gereklidir. Benzer şekilde ekran köşelerinin düz veya yuvarlak olması, fiziksel klavye olup olmaması, dokunmatik ekrana sahip olup olmaması durumunda kullanıcının uygulamayı benzer deneyimle kullanabilmesi gereklidir.

Flutter çerçevesinde bulunan bileşenler bu iki özelliğin de desteklenmesini sağlayacak şekilde tasarlanmıştır. Bazı durumlarda uygulamanın çalıştığı cihazın donanım özellikleri veya konfigürasyonu hakkında bilgi edinmek gerekebilir. Çünkü ekrandaki objelerin nasıl dağılacığını buradan yola çıkarak karar vermek isteyebilirsiniz. Örneğin bir e-posta kutusu uygulaması yatayda çok alanı olan bir pencerede hem e-posta listesini hem de açılan e-posta içeriğini yan yana göstermek isterken dar bir ekranda bunları iki ayrı pencerede ileri geri giderek açmak isteyebilir.

Bunun için widgetların build metodu içerisinde **MediaQuery.of()** metodunu çağırarak cihaz bilgisine erişebiliriz. Bu metot bize cihaz hakkında ekran boyu, çözünürlüğü gibi bilgileri sağlar. Uygulamanın genelindeki davranış için **MediaQuery.of()** metodundan elde edilecek değerlere göre davranış mak gerekebilir. Bir widgetin çizimi ile alakalı düzenlemeler için Flutter'ın sunduğu özel düzenleyici widgetların kullanılması çoğu zaman daha pratik ve yerinde bir çözüm olacaktır. Aşağıda bunların özet bir listesi bulunmaktadır.

AspectRatio

CustomSingleChildScrollView

CustomMultiChildScrollView

FittedBox

FractionallySizedBox

LayoutBuilder

MediaQuery

MediaQueryData

OrientationBuilder

5.6. Arayüz Tasarımında Kısıtları Anlama

Flutter arayüz bileşenlerini belirli konumlara yerleştirmek yerine daha çok bileşenlerin birbiri ile ve alt üst ile ilişkilerini belli kısıtlara (constraints) göre tanımlamaktadır. Bu yaklaşımın handikabı arayüz tamamen çizilmeden nasıl görüneceğinin bilinmemesidir. Avantajı ise belli bir ekran boyutuna veya çözünürlüğe göre değil bileşenlerin birbirleri ile ilişkisine göre arayüze yerleşmesi bu sayede farklı ekranlarda belirtilen kısıtlara göre değişik arayüz düzenleri sağlarasıdır. Flutter, Web sayfalarındaki html objelerinin yerleşim kurallarından çok farklı bir yerleşim modeli kullanmaktadır.

Flutter sayfasında bu düzenle alakalı şu uyarı yapılmaktadır. “Constraints go down. Sizes go up. Parent sets position.” Bu uyarıyı şöyle açıklayabiliriz. Arayüz bileşenlerinin oluşturduğu ağaç yapısında kısıtlar aşağı doğru aktarılır. Boyutlandırmalar tersine aşağıdan yukarı doğru aktarılır ve ebeveyn obje arayüz bileşeninin ekrandaki konumunu belirler. Yani bir bileşen alt bileşenlerden ebatlarının ne olması gerektiğini üst bileşendense ekranda ne kadar alana yayılabileceğini öğrenir. Bunun üzerine kendisinin ne kadar boyutta olacağını ebeveyn bileşene bildirir ve ebeveyn bileşen hangi konumdan çizime başlanacağını tespit eder.

Flutter’ın arayüz düzenlemesinde koyduğu kısıtlamaları şöyle sıralayabiliriz;

Bir widget kendi boyutunu ancak üst seviye bileşenin verdiği kısıtlar arasında seçebilir. Yani bir widget hiçbir zaman kendi istediği boyutlarda olamaz.

Bir widget ekranda hangi konumda olacağını asla bilemez ve buna karar veremez. Konumlandırmasında söz sahibi üst seviyedeki widgettir.

Bir üst seviyedeki widgetin boyutu belirleyebilmesi de onun üstündeki widgeta bağlı olacağından tam widget ağaçları işlenmeden arayüzün nasıl olacağı bilinemez.

Bir widget üst seviyeden daha fazla bir boyutta olmak ister ve üst seviyenin alttaki widgetları nasıl yerlestireceği ile ilgili bilgisi yoksa alttaki elemanın verdiği ölçü değerleri yok sayılır.

Flutter Constraints sayfasında bir Container arayüz bileşeninin farklı özellikle ve widget ağaç hiyerarşisinde davranışları verilmiştir. Bunlardan birkaçı burada inceleyelim.

```
Container(color: red)
```

Birinci örnekte sadece arkaplan rengi kırmızı olarak belirtilen bir Container vardır. Bu Container üstte doğrudan ekranı simgeleyen bileşenini almaktadır. Ekran, Container'a kısıt olarak genişlemesini ve içeriği doldurmasını istemektedir. Neticede Container tam ekran kırmızı görünür.

```
Container(width: 100, height: 100, color: red)
```

İkinci örnekte Container için width(en), height(boy) verileri tanımlanır. Ama üst seviye kısıt yine aynıdır. Container ilk örnekteki gibi tam ekran olacaktır.

```
Center(  
    child: Container(width: 100, height: 100, color: red),  
)
```

3. örnekte Container, Center adlı başka bir düzenleyici widget içerisinde konulmuştur. Center ekrandan gelen genişleme kısıdını iptal eder ve tam tersi ortaya doğru objeyi öteler. Center’ın ötelemesi sonucu Container kendi verdiği width ve height değerleri kadar yer kaplar.

```
Align(  
  alignment: Alignment.bottomRight,  
  child: Container(width: 100, height: 100, color: red),  
)
```

4. örnekte Container bu sefer Align ile sarmalanmıştır. Align, Center ile aynı mantıkta çalışmaktadır. Sadece fark olarak ekranın neresine doğru yerleşilmesi isteniyorsa o alignment parametresi ile belirtilir.

Flutter Constraints sayfasında bu şekilde 29 farklı örnekle kullanılan arayüz düzenleyici widgetların birbirlerine olan etkileşimi incelenmektedir. Sonraki bölmelere geçmeden önce tüm örnekleri çıktılarındaki yerleşim gerekçesini yorumlayacak şekilde inceleyiniz. Bu daha önceden div ile web sayfası tasarımlı yapan yazılımcılar için biraz sıra dışı gelebilir.

5.7. Alıştırma

Flutter.dev sayfasında bu ana kadar işlediğimiz konuları adım adım deneyeceğiniz bir alıştırma bulunmaktadır. “Building Layouts” adlı alıştırma sayfasındaki yönergelerle anlatılanları örnek projede tatbik ediniz.

Bölüm Özeti

Flutter arayüz düzenlemesi için birçok bileşen sunmaktadır. Bu bileşenler farklı amaçları yerine getirmek ve değişik hiyerarşiler oluşturmak için kullanılır. Arayüz düzeni için web sayfasına benzer düzenleyici widgetlar vardır. Fakat Flutter’ın widgetları ekrana çizme şekli ve bir widgetin ekranda nasıl yerleştirileceğinin mantığı farklıdır. Flutter widgetlar arası hiyerarşide kısıtlar (constraints) uygular. Bu kısıtlar alt üst ilişkisi ile arayüz bileşeninin konumunu ve boyutlarını belirlemek için kullanılır.

Kaynakça

<https://flutter.dev/docs/development/ui/layout>

<https://flutter.dev/docs/development/ui/layout/adaptive-responsive>

6. KULLANICI ETKİLEŞİMİ VE MATERİYAL YÖNETİMİ

Giriş

Bu bölümde Flutterda kullanıcı etkileşimi yapabilen bir uygulama yazmak için StatefulWidget ile basit durum yönetimi ve resim gibi kod harici dosyaların projeye nasıl dahil edileceği inceleneciktir. Önceki bölümlerde verilen örneklerin başlamadan önce yapılması bu bölümün kolay anlaşılmasına önemlidir. Öğrencilerin bir önceki bölümde Flutter.dev sayfasındaki örnekleri denemeleri önerilir.

6.1. Flutter Uygulamasına Kullanıcı Etkileşimi Verme

Onceki derslerimizde bir Flutter uygulamasının kullanıcı etkileşimi sağlaması için IconButton gibi Widgetların, onPressed gibi parametrelerine fonksiyon beslemesi vermemizin yeterli olacağını söylemiştim. Ayrıca kendi yazacağımız arayüz bileşenlerinde GestureDetector kullanarak benzer şekilde onTap, onPressed gibi olayları arayüz bileşenimize kazandırabiliriz. Bu sıraladığımız iki yöntem kullanıcıdan girdi almak için yeterlidir. Fakat Flutter’da bir butona tıkladığınızda ilgili fonksiyon içerisinde yeni pencere aç gibi bir kod yazamamaktayız. Flutter kullanıcı arayüzüne ancak bir durum veya konfigürasyon değişikliği olduğunda yenilemektedir. Yani GestureDetector ile onTap oyununu hazırladık ve kullanıcı da bizim arayüz bileşenimize dokundu. İlgili fonksiyon çalışmıyor ama arayüzde bir değişiklikle bunu kullanıcıya göstermek için

sadece bu yeterli değildir. StatelessWidget & StatefulWidget konularını anlatırken Flutter'da arayüzde güncelleme yapmak istediğimiz durumlarda StatefulWidget oluşturmalı ve **setState** metodunu kullanarak bir durum değişikliği olduğunu Flutter'a bildirmeli, bu sayede arayüzün yeniden çizilmesini sağlamalıyız demiştim. Bu bilgiler ışığında kullanıcı etkileşimli bir Flutter arayüzü için iki temel unsur gerekmektedir. Birincisi kullanıcının etkileşimi ilgili arayüz bileşeni üzerinden yakalayacak onTap, onPressed gibi bir olay tetikleyicisi hazır edilmelidir. İkinci olarak da arayüzde değiştirilmek istenilen bölge StatefulWidget ile sarmalanarak arayüzü güncelleme gerektiren işlem setState içeresine alınmalıdır.

Eğer kullanıcı etkileşimi sonucunda sadece etkileşimde bulunulan arayüz bileşeni için ekranın yeniden çizilmesi gerekiyorsa örneğin bir IconButton tıklandığında gösterdiği ikonun değişmesi gibi, Flutter'in birçok hazır widgeti kendi üzerindeki arayüz çizimlerini düzenleyecek şekilde kendi durum yönetimine sahiptir.

Biz burada durum yönetimini tamamen bizim vereceğimiz bir senaryo üzerinden farklı yaklaşımıları inceleyeceğiz. Bunun için StatefulWidget sınıfından miras alacak bir arayüz bileşeni ve setState metodunun çağrıldığı bir kod bloğu oluşturacağız.

Widget'ın arayüzde sadece yenilenmesi gereken yeri kapsayacak şekilde yerleştirilmesi ve bu sayede arayüzün sadece gereken bölümlerinin yenilenmesinin performansı olumlu etkileyeceğini söylemiştim. Peki setState nerede yer almmalıdır? İhtiyacımıza göre kurgulayabileceğimiz 3 yaklaşım karşımıza çıkmaktadır;

1. Kullanıcının etkileşimde bulunduğu arayüz bileşeni içerisinde. Bu kendi üzerinde güncelleme yapacak bir arayüz bileşeni oluşturacaksak tercih edeceğimiz yöntemdir.
2. Arayüzü oluşturan widget ağacında kullanıcının etkileşimde bulunduğu arayüz bileşenin üst seviyesinde(ebeveyn). Eğer etkileşimde bulunulan nesne dışında bir yerin arayüzde güncellenmesi gerekiyorsa.
3. İlk iki yöntemin melezi bir yaklaşım. Eğer hem etkileşimde bulunan widget hem de başka bir bölgede güncellenme gerekiyorsa iki yöntem bir arada da kullanılabilir.

Şimdi sırasıyla bu üç yaklaşımı bakalım. Senaryo tıklandığında yeşil(Active) ile gri(Inactive) arasında geçiş yapacak bir Container tasarlamaktır.



6.1.1. Kendi Durumunu Yöneten Arayüz Bileşeni

Birinci örneğimizde TabboxA adında StatefulWidget'tan miras alarak oluşturduğumuz bir arayüz bileşenimiz bulunmaktadır. TabboxA için durum sınıfı _TabboxAState ile tanımlanmış ve setState kullanımı burada tanımlı _handleTap metodunu içerisinde gerçekleştirilmektedir. _handleTap metodunu, TabboxA'nın içerisinde kullanıcı etkileşimi yakalamak için kullanılan GestureDetector widgetındaki onTap parametresine beslenmektedir. Bu örnekte TabboxA arayüz bileşeni kendi dışındaki widget ağacından bağımsız olarak kullanıcı etkileşimi alıp kendi durumunu ve görselini güncelleyebilmektedir.

```
import 'package:flutter/material.dart';

// TapboxA manages its own state.

//----- TapboxA -----
class TapboxA extends StatefulWidget {
  const TapboxA({Key? key}) : super(key: key);

  @override
  _TapboxAState createState() => _TapboxAState();
}

class _TapboxAState extends State<TapboxA> {
  bool _active = false;

  void _handleTap() {
    setState(() {
      _active = !_active;
    });
  }

  @override
  Widget build(BuildContext context) {
    return GestureDetector(
      onTap: _handleTap,
      child: Container(
        child: Center(
          child: Text(
            _active ? 'Active' : 'Inactive',
            style: const TextStyle(fontSize: 32.0, color: Colors.white),
          ),
        ),
        width: 200.0,
        height: 200.0,
        decoration: BoxDecoration(
          color: _active ? Colors.lightGreen[700] : Colors.grey[600],
        ),
      ),
    );
  }
}
```

```
//----- MyApp -----  
  
class MyApp extends StatelessWidget {  
  const MyApp({Key? key}) : super(key: key);  
  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      title: 'Flutter Demo',  
      home: Scaffold(  
        appBar: AppBar(  
          title: const Text('Flutter Demo'),  
        ),  
        body: const Center(  
          child: TapboxA(),  
        ),  
      ),  
    );  
  }  
}
```

Koddaki tüm işlemler `_TabboxA` State içerisinde `_active` adlı Boolean değişkenin aldığı değere göre yapılmaktadır. Kodu dikkatlice incelerseniz `_handleTap` metodu içerisinde arayüz çizimi ile ilgili hiçbir işlem yapılmamaktadır. Sadece `_active` değişkeninin tuttuğu değer değiştirilmektedir ve bu işlem `setState` ile sarmalanmıştır. `setState` sonuna gelindiğinde Flutter `setState`'in içerisinde bulunduğu widget için `build` metodunu tekrar çağrıarak arayüzü güncellemek ister. İşte bu noktada Flutter'ın kendine has kodlama dinamiğini görmekteyiz. Arayüzde direkt olarak renk şu olsun şeklinde bir olay çağrıları yapılmaz. Bunun yerine burada da olduğu gibi belli değişkenlerin aldığı değerlere göre `build` metodu arayüzü nasıl çizeceğine karar verir. Misal `Text` bileşeninde (`_active ? 'Active' : 'Inactive'`,) satırı ile gelecek metin, `Container` bileşeninde de (`color: _active ? Colors.lightGreen[700] : Colors.grey[600]`,) satırı ile arkaplan rengi seçimi yapılmaktadır.

6.1.2. Ebeveyn Bileşenin Durumu Yönetmesi

Bazı durumlarda hazırladığımız arayüz bileşenlerinde dinamiklik için kullanıcı etkileşimi fonksiyonlarını kendimiz kodlamayız. Bunun yerine bu fonksiyonları yapıçı fonksiyon üzerinden parametre olarak almayı tercih ederiz. `IconButton`'un `onPressed` metodunun yapıçı fonksiyonu çağrısında bizim tarafımızdan atandığını anımsayalım. Bu durumda arayüzdeki değişimi yapacak olan kod parçası (kullanıcı etkileşiminde çağrılacak fonksiyon) arayüz bileşeni içerisinde değil daha üst seviye bir bileşen içerisinde bulunacaktır. Doğal olarak da `setState` çağrısının bu kodu sarmalayacak şekilde yazılması gerekecektir. İşte bu durumda arayüzde değişecek olan görselliği kontrol eden `setState` daha üst bir bileşenden çağrılmak durumundadır.

Bir önceki `TobboxA`'da gerçekleştirilen işlevi aynen `TobboxB` sınıfında yapalım. Farklı olarak `TobboxB` içerisinde `setState` çağrılmayacağı için `TobboxB` tanımlaması basitlik açısından `StatelessWidget`'tan miras alınarak yapılacak, tıklanma için çağrılmak ise asıl işlevini yapıçı üzerinden olacaktır.

```
import 'package:flutter/material.dart';

// ParentWidget manages the state for TapboxB.

----- ParentWidget -----


class ParentWidget extends StatefulWidget {
  const ParentWidget({Key? key}) : super(key: key);

  @override
  _ParentWidgetState createState() => _ParentWidgetState();
}
```

```
class _ParentWidgetState extends State<ParentWidget> {
  bool _active = false;

  void _handleTapboxChanged(bool newValue) {
    setState(() {
      _active = newValue;
    });
  }

  @override
  Widget build(BuildContext context) {
    return SizedBox(
      child: TapboxB(
        active: _active,
        onChanged: _handleTapboxChanged,
      ),
    );
  }
}
```

```
----- TapboxB -----


class TapboxB extends StatelessWidget {
  const TapboxB({
    Key? key,
    this.active = false,
    required this.onChanged,
  }) : super(key: key);

  final bool active;
  final ValueChanged<bool> onChanged;

  void _handleTap() {
    onChanged(!active);
  }
}
```

```

@Override
Widget build(BuildContext context) {
  return GestureDetector(
    onTap: _handleTap,
    child: Container(
      child: Center(
        child: Text(
          active ? 'Active' : 'Inactive',
          style: const TextStyle(fontSize: 32.0, color: Colors.white),
        ),
      ),
      width: 200.0,
      height: 200.0,
      decoration: BoxDecoration(
        color: active ? Colors.lightGreen[700] : Colors.grey[600],
      ),
    ),
  );
}
}

```

TobboxB ve onu içerisindeki ParentWidget kodlarını incelediğimizde bir önceki TobboxA'da yapılan işlemin aynısının yapıldığını, fark olarak TobboxB örneğinde görsellik işlevsellik kodlarının ayrıldığını göreceksiniz. Eğer TobboxB gibi bir görsel tasarım yapmak istiyorsak ama işlevsellliğini kodlamayı dinamiklik açısından sonraya bırakacağımızda setState çağrıları ve durum kontrolü üst bir widget içerisinde gerçekleştirilecektir. Burada StatefulWidget kodlamasının mümkün olan en alt seviyede yazılması gerektiğini tekrar hatırlatalım. Tüm ekranı kaplayan bir StatefulWidget kullanmak performans açısından hiç iyi değildir. En ufak bir kullanıcı etkileşiminde tüm ekrandaki bileşenlerin baştan çizilmesine neden olacağını unutmamak gereklidir.

6.1.3. Melez Yaklaşımıla Durum Yönetimi

İlk iki örnekte setState metoduna çağrıının duruma göre değişimin olacağı yerde veya üst seviyede bulunabileceğini gördük. Buna göre StatefulWidget ve setState kodlaması widget ağaçının içerisinde uygun yerde yapıldı. Peki bir arayüz bileşeni oluştururken hem sonradan dinamik verilecek bir kullanıcı etkileşimi fonksiyonumuz hem de kendi içimizde düzenlemek istediğimiz bir kullanıcı etkileşimimiz olsa nasıl bir yaklaşım izlemeliyiz? Bu duruma örnek olarak TobboxC örneğine bakalım. Bu örnekte aynı yeşil(Active)-gri(Inactive) Container üzerine basılı tutulduğunda bunu farklı bir arkaplan rengi ve kenarlıkla kullanıcıya tepki verecek şekilde güncellenecektir. Amacımız onTap (Tıklanma) işlevinin dinamik dışarıdan gelmesi ama bunun dışında kullanıcının basılı tutma ve tıklamadan cayma durumundaki işlevlerin TobboxC kodlamasının kendi içerisinde halledilmesidir. Tüm kullanıcı etkileşimlerinde arayüzde değişiklik gereği için setState hem üst seviyede hem de TobboxC içerisinde kullanılacaktır. Bu durumda TobboxB'de olduğu gibi üst seviyedeki Widget StatefulWidget seçilmeli ve TobboxA'da olduğu gibi TobboxC de StatefulWidget seçilmelidir. Örnek kodlama aşağıdaki gibi olacaktır.

```

import 'package:flutter/material.dart';

//----- ParentWidget -----
class ParentWidget extends StatefulWidget {
  const ParentWidget({Key? key}) : super(key: key);

  @override
  _ParentWidgetState createState() => _ParentWidgetState();
}

```

```
class _ParentWidgetState extends State<ParentWidget> {
  bool _active = false;

  void _handleTapboxChanged(bool newValue) {
    setState(() {
      _active = newValue;
    });
  }

  @override
  Widget build(BuildContext context) {
    return SizedBox(
      child: TapboxC(
        active: _active,
        onChanged: _handleTapboxChanged,
      ),
    );
  }
}
```

----- TapboxC -----

```
class TapboxC extends StatefulWidget {
  const TapboxC({
    Key? key,
    required this.active,
    required this.onChanged,
  }) : super(key: key);

  final bool active;
  final ValueChanged<bool> onChanged;

  @override
  _TapboxCState createState() => _TapboxCState();
}

class _TapboxCState extends State<TapboxC> {
  bool _highlight = false;

  void _handleTapDown(TapDownDetails details) {
    setState(() {
      _highlight = true;
    });
  }

  void _handleTapUp(TapUpDetails details) {
    setState(() {
      _highlight = false;
    });
  }
}
```

```

void _handleTapCancel() {
  setState(() {
    _highlight = false;
  });
}

void _handleTap() {
  widget.onChanged(!widget.active);
}

@Override
Widget build(BuildContext context) {
  // This example adds a green border on tap down.
  // On tap up, the square changes to the opposite state.
  return GestureDetector(
    onTapDown: _handleTapDown, // Handle the tap events in the order that
    onTapUp: _handleTapUp, // they occur: down, up, tap, cancel
    onTap: _handleTap,
    onTapCancel: _handleTapCancel,
    child: Container(
      child: Center(
        child: Text(widget.active ? 'Active' : 'Inactive',
          style: const TextStyle(fontSize: 32.0, color: Colors.white)),
      ),
      width: 200.0,
      height: 200.0,
      decoration: BoxDecoration(
        color: widget.active ? Colors.lightGreen[700] : Colors.grey[600],
        border: _highlight
          ? Border.all(
              color: Colors.teal[700]!,
              width: 10.0,
            )
          : null,
      ),
    ),
  );
}
}

```

6.2. Interaktif Arayüz Bileşenleri

Flutter widget kataloğu üzerinde “Material Design” kurallarına uyan çok zengin bir interaktif arayüz bileşen gurubu sunmaktadır. Ayrıca yine iOS görünümü veren Cupertino grubu bileşenleri de sunmaktadır. Bunların dışında kendi özel tasarımlarımız için GestureDetector ile interaktif arayüzü istediğimiz şekilde tasarlama imkânımız da bulunmaktadır. Çoğu durumda en basit ve etkili yaklaşım hazır bileşenlerden birini tercih etmektir. Bunlardan belli bir kısmının referans dokümanına aşağıdaki bağlantılarından erişebilirsiniz.

Standart Widgetlar

Form

FormField

Material Grubu Bileşenler

Checkbox

DropdownButton

TextButton

FloatingActionButton

IconButton

Radio

ElevatedButton

Slider

Switch

TextField

Konu ile ilgili diğer kaynaklar için Flutter.dev Interactive sayfasındaki kaynaklar kısmını inceleyebilirsiniz. Bir önceki ders sonundaki alıştırma örneğini etkileşimli hale getirmek için alıştırmanın devamına yine aynı sayfadan erişilmektedir.

6.3. Resim ve Diğer Materyallerin Yönetimi

Bir Flutter projesi Dart dilinde yazdığımız kodlardan ve diğer varlıklardan (assets) oluşmaktadır. Varlık (asset) projenizdeki kod dışındaki her tür veri dosyasını (resim, metin, müzik, video) temsil eden bir tanımlamadır. Flutter'da geliştirilen projeler mobil platformlar için yükleme dosyasına dönüştürülmemektedir. Bu yükleme dosyalarının (örneğin Android için .apk) içerisinde hangi dosyaların dahil edileceği pubspec.yaml dosyası içerisindeki assets: başlığında verilir. Şablon projemizde nasıl yazılacığı ile ilgili örnek açıklama metni olarak verilmiştir.

Aşağıdaki örnekte pubspec.yaml dosyası içerisinde iki resim tanımlaması yapılmaktadır.

```
flutter:  
  assets:  
    - assets/my_icon.png  
    - assets/background.png
```

Yukarıdaki örnekte assets adlı klasör altındaki iki png dosyası proje yapısına dahil edilmiştir. pubspec.yaml dosyasında bu belirtme olmaksızın proje içerisinde assets adında bir klasör açıp resimleri içine koymaz bunları projeye dahil etmeyecektir.

Dosyaları tek tek göstermek yerine klasör şeklinde de varlık tanımlaması yapılabilir.

```
flutter:  
  assets:  
    - klasor/  
    - klasor/altklasor/
```

Flutter varlık yönetiminde bazı akıllı klasörleme yaklaşımı uygular. Özellikle farklı temalarda aynı dosyaların birden fazla versiyonları olduğu durumda bu varlık yönetimini kolaylaştırmaktadır.

Örneğin aşağıdaki klasör yapısında pubspec.yaml dosyasında background.png için bulunacak tek kayıt iki background.png dosyasını da projeye dahil edecektir.

```
.../pubspec.yaml  
.../graphics/my_icon.png  
.../graphics/background.png  
.../graphics/dark/background.png  
...etc.
```

```
flutter:  
  assets:  
    - graphics/background.png
```

Benzer ve daha geniş bir etkiyi sadece graphics klasörünü göstererek de sağlayabiliyoruz. Bu durumda graphics klasörü altındaki tüm dosya ve alt klasörlerdeki dosyalar projeye dahil edilecektir.

6.3.1. Varlıkların Kod İçerisinden Erişimi

Flutter'da kod içerisinde varlıklar(**assets**) **AssetBundle** adındaki bir obje üzerinden erişilir. AssetBundle yapısındaki metin türü varlıklara erişmek için **loadString()**, resim ve diğer binary formdaki varlıklara erişmek için **load()** fonksiyonları kullanılır. Fonksiyonlar pubspec.yaml dosyasında belirtilen mantıksal bir anahtarı parametre olarak alır.

Her Flutter uygulamasında **rootBundle** ana varlık yapısına erişim için kullanılır. rootBundle, 'package:flutter/services.dart' içerisinde tanımlanmış global statik formda tanımlanmıştır. Fakat bunun yerine güncel BuildContext (Widgetların build metoduna parametre olarak verilmektedir.) üzerinden DefaultAssetBundle ile varlıklara erişim tavsiye edilmektedir. Böylece anlık olarak konfigürasyon değişikliklerini doğru yansitan bir uygulama yapısı oluşturulur. BuildContext'e erişemeyeceğimiz durumlarda, örneğin widgetların build metotları dışındaki kodlarda rootBundle kullanarak varlıklara erişim sağlamamız icap eder. Config.json adındaki bir varlık dosyasına erişim aşağıdadır. Burada eriştiğimiz yer kalıcı hafıza olduğu ve bu zaman alabilen bir I/O işlemi olduğu için erişim şeklinin asenkron yazıldığına dikkat ediniz.

```
import 'dart:async' show Future;  
import 'package:flutter/services.dart' show rootBundle;  
  
Future<String> loadAsset() async {  
  return await rootBundle.loadString('assets/config.json');  
}
```

6.3.2. Resim Varlıkları

Flutter'da resim varlıklarına erişmek için AssetImage adında özel bir sınıf vardır. AssetImage cihazdaki piksel yoğunluğuna bağlı olarak farklı ebatlardaki resimler arasından otomatik seçim yapabilmektedir. Bunun için farklı ebatlardaki aynı isim ile isimlendirilmiş resimlerin aşağıdaki gibi bir klasör yapısı ile sunulması gereklidir.

```
.../my_icon.png  
.../2.0x/my_icon.png  
.../3.0x/my_icon.png
```

AssetImage sınıfının yapıcısı fonksiyonunu bir widgetin build metodunda içerişinde çağırarak sayfaya resim yükleriz.

```
Widget build(BuildContext context) {  
  return Image(image: AssetImage('graphics/background.png'));  
}
```

Bazen paket olarak gelen bir kütüphane altında tanımlanmış resim varlıklarına erişmek isteriz. Bu durumda AssetImage yapıcısı paket bilgisi ile çağrırlar.

```
AssetImage('icons/heart.png', package: 'my_icons')
```

Flutter projesindeki varlıklara alt yapıda çalışan Android ve iOS için yazılan doğal kod kısmından da erişilebilmektedir. Bununla ilgili yönergeler için “Sharing Assets with the Underlying Platform” başlığındaki içerik incelenebilir.

Bölüm Özeti

Flutter’da kullanıcı etkileşimlerini yakalamak için hazır widgetlar kullanılabileceği gibi kendi widgetlarımızda GestureDetector ile özel olarak da tasarlayabilmekteyiz. Kullanıcı etkileşiminde geriye arayüz üzerinden tepki vermemiz gereklidir. Bunun için Flutter çerçevesinin arayüzde belli bir bölgeyi yeniden çizmesi gerekecektir. Her widgetin içerisinde arayüz çizimini yapan build adında bir metod bulunmaktadır. Bu metodun tetiklenmesi için StatefulWidget kullanımı ve setState ile durum değişikliği Flutter çerçevesine aktarılır. setState metodu arayüz çiziminde değişmesi gereken yeri kapsayacak widget içerisinde olmalıdır. Bu seviye hiyerarşide ne kadar aşağıda tutulursa o kadar iyidir. Çünkü widget ağacında bu noktadan aşağıdaki bileşenler baştan çizileceklerdir.

Flutter projesinde kod dışındaki veri dosyaları asset olarak isimlendirilir ve pubspec.yaml dosyası üzerinden tanıtılmalıdır. Bu tanıtım dosya bazlı veya klasör bazlı yapılabilir. Varlık yönetimi ve kod içerisinde erişim için AssetBundle objeleri kullanılır. Uygulamadaki tüm varlık bilgisine erişim için rootBundle kullanılır. Ayrıca Widgetların build metodunun BuildContext parametresi üzerinden de varlıklara erişilir. Kod içerisinde resimlerin yönetimi için AssetImage adında yardımcı bir sınıf tanımlıdır. AssetImage sınıfı varlıklara erişirken çözünürlüğe göre farklı ebattaki resimleri de otomatik algılayıp uygulamaktadır.

Kaynakça

<https://flutter.dev/docs/development/ui/interactive>

<https://flutter.dev/docs/development/ui/assets-and-images>

7. EKRANLAR ARASI NAVİGASYON VE YÖNLENDİRME

Giriş

Flutterda navigasyon ve yönlendirme, web sayfaları arasında gezintiye benzemektedir. Hatta Flutter uygulamasını web destekli çalıştırırsanız tam olarak bunu internet tarayıcınızın adres çubuğuunda göreceksiniz. Bir web sitesi içerisinde gezinirken URL (Adres çubuğuunda yazan “Unique Resource Locator”) ifadesinin nasıl değiştığını gözlemleyiniz. URL içerisindeki ? # şeklindeki özel sembollerin ne için kullanıldığını anımsayınız.

7.1. Flutterda Uygulama İçi Gezinti

Şimdije kadar yaptığımız uygulamalar hep tek sayfa içerisinde görsel objeler içeren basit uygulamalardı. Daha gelişmiş uygulamaların yazımında genelde kullanıcı arayüzü mantıksal alt parçalara ayrılmak durumundadır. Akıllı Cep Telefonları gibi küçük ekranlı cihazlarda uygulamalar genellikle tam ekran ve tek pencere formunda çalışırlar (Ekranı iki uygulamaya bölmeye veya Picture-in-Picture çalışma kiplerini göz ardı edersek). Bunun sebepleri ekranın küçüklüğü, donanımsal klavye yerine ekran klavyesi kullanma zorunluluğu ve kullanıcı odağını yakalama olarak sıralanabilir. Gelişmiş bir uygulamada birden fazla ekran tasarımı ve bunların arasında bir gezinme (navigasyon) sisteminin olması gerekmektedir.

Flutter’da uygulama içi gezinti için iki farklı yöntem sunulmaktadır. Bunlar Navigator 1.0 API ve Navigator 2.0 API olarak ifade edilmektedir. Navigator 1.0 API, Flutter’ın ilk çıktığında kullandığı gezinti yöntemidir ve aktif olarak kullanılmaktadır. **Navigator** ve **Route** objeleri bu yöntemin ana çalışma prensiplerini

belirlemektedir. Navigator 2.0 API, daha bildirimsel(declarative) bir yaklaşım sunmak için Flutter'a dahil edilmiştir. **Router** ve **Page** adındaki yapıları baz almaktadır. Navigator 2.0 yapısı daha modern bir yaklaşımındır. Fakat her iki API da Flutter projelerinde kullanılabilir. Şimdi sırasıyla bu yöntemleri ve bileşenlerini inceleyelim.

7.2. Navigator 1.0 API

Mobil uygulamalardaki tam ekran açılan yapılar genelde “screen” ve “page” olarak isimlendirilmektedirler. Flutter’da bu elamanlara “route” adı verilmektedir ve **Navigator** adındaki widget tarafından kontrol edilmektedir. Navigator, Route objelerinden oluşan bir yığını kontrol eder. Bu yığındaki sayfalar bildirimsel olarak Navigator.pages veya emredici şekilde Navigator.push ve Navigator.pop (push ve pop bir yığına eleman ekleme ve çıkışma için kullanılan jargondur.) iki farklı yöntemle yönetilir.

Yığın kullanıcının gezindiği sayfaların geriye dönük bir kaydını tutmak için kullanılır ve kullanıcı geri gitmek istediği yıldızdan ilgili sayfa ekrana getirilir. Çoğu akıllı telefon modelinde bu işlem için donanımsal bir geri butonu bulunmaktadır. Donanımsal butonun olmaması durumunda da Scaffold widgetındaki AppBar üzerinde geri(back) butonu eklenerek bu işlevsellik kullanıcı ekranında yazılımla sağlanır.

Bir Navigator objesini kod içerisinde doğrudan oluşturabiliriz. Fakat genel olarak kullanılan yöntem Navigator objesini bize sağlayan WidgetsApp veya MaterialApp widgetlarından birinin kullanılmasıdır. MaterialApp kullanımı en basit yaklaşımındır. Yazdığımız uygulamaların çoğu uygulama kökünde zaten MaterialApp widgetinin kullanıldığı fark etmişinizdir.

```
void main() {  
  runApp(MaterialApp(home: MyAppHome()));  
}
```

Eğer yığın içerisinde yeni bir ‘Route’ tanımlamak istersek MaterialPageRoute'un bir örneğini oluşturup builder fonksiyonuna ekranda görünmesini istediğimiz widget ağacını verebiliriz. Örneğin aşağıdaki kod parçası Scaffold widgeti ile başlayan yeni bir widget ağacını ekrana yansıtacaktır. Burada Navigator.push verilen MaterialPageRoute objesi ile ekranı çizer. Navigator.pop() metodu ise bu ekrandan bir önceki ekranı geri dönüş sağlar (geri “back” butonuna tıklama gibi).

```
Navigator.push(context, MaterialPageRoute<void>(  
  builder: (BuildContext context) {  
    return Scaffold(  
      appBar: AppBar(title: Text('My Page')),  
      body: Center(  
        child: TextButton(  
          child: Text('POP'),  
          onPressed: () {  
            Navigator.pop(context);  
          },  
        ),  
      ),  
    );  
  },  
));
```

Route altında doğrudan bir widget bulunmaz, bunun yerine builder adındaki metod kullanılır. Çünkü sayfanın yığına atılması ve çekilmesi gibi durumlarda ekranın yeniden inşa edilmesi gerekebilir. Örneğin Navigator.pop() çağrısında olduğu gibi. Navigator.push' çağrısında ekrana gelecek Route tanımlanır ama Navigator.pop'da böyle bir bilgi verilmeye gerek yoktur. Varsayılan davranış Route yığını içerisinde en son eklenen (son ziyaret edilen ekran) getirilir.

7.2.1. İsimlendirilmiş Yol Kullanımı

MaterialApp widgetin kullanımında şu ana kadar uygulamanın giriş ekranını “home” parametresi ile tanımladık. Kullanıcının başka sayfalara geçmesi için yukarıdaki örneğimiz Navigator.push çağrısına bir MaterialPageRoute objesi beslemektedir. MaterialApp widgetinin “**routes**” adlı parametresine Map<String, WidgetBuilder> türünde uygulama içerisinde gezinebileceğimiz sayfaları isimlendirerek verebiliriz. Aşağıdaki örnekte uygulamanın kök dizini dışında /a /b /c şeklinde erişilebilecek üç isimlendirilmiş sayfası bulunmaktadır.

```
void main() {  
  runApp(MaterialApp(  
    home: MyAppHome(), // '/' olan kök dizini ifade edecektir.  
    routes: <String, WidgetBuilder> {  
      '/a': (BuildContext context) => MyPage(title: 'page A'),  
      '/b': (BuildContext context) => MyPage(title: 'page B'),  
      '/c': (BuildContext context) => MyPage(title: 'page C'),  
    },  
  ));  
}
```

Burada her sayfa bir yol tanımlaması (String) ve bu yola karşılık sayfayı oluşturacak bir fonksiyon (WidgetBuilder) şeklinde bir Map oluşturularak routes parametresine beslenmektedir.

Yukarıdaki sayfalardan birini çağırmak için Navigator.pushNamed metodu çağrılır.

```
Navigator.pushNamed(context, '/b');
```

7.2.2. Açılan Sayfadan Değer Döndürme

pop metodu ile bir sayfadan çağrıran sayfaya dönüldüğünde istenilen değerin dönülmesi de sağlanabilir. Bunun için push metodu çağrısında istenilen dönüş değerini belirten bir sayfa için builder oluşturulur. Dönülecek değer bir Future objesidir. Çünkü push metodunun çağrısından pop çağrısına kadar geçecek süre içerisinde bir bekleme süreci olacaktır. Bu nedenle de push çağrısının önüne await anahtar kelimesi kullanılarak dönen sonuç bir değişkene aktarılır. Aşağıdaki örnekte açılan ekrandaki ‘OK’ yazan kutuya tıklanarak önceki sayfaya dönülürse “true” aksi halde (geri butonuna tıklanma durumu) “null” değer dönecek bir tanımlama yapılmıştır.

```
bool value = await Navigator.push(context, MaterialPageRoute<bool>(  
  builder: (BuildContext context) {  
    return Center(  
      child: GestureDetector(  
        child: Text('OK'),  
        onTap: () { Navigator.pop(context, true); }  
      ),  
    );  
  }  
>);
```

Route bir geri dönüş değerine sahip olacaksızın tanımlamasının dönülecek değer türüne göre olması gereklidir. Bu nedenle bu örnekte Route tanımlamamız MaterialPageRoute<void> yerine MaterialPageRoute<bool> olarak verilmiştir.

7.2.3. Açılan Sayfaya Argüman Gönderme

Bir web sayfasında soru işaretri (?) karakterinden sonra gelen argüman listesi gibi Flutter’da da bir sayfanın açılmasında argüman listesi beslenebilmektedir. Bunu Navigator.pushNamed() metodunun **arguments**

parametresi ile sayfaya gönderebiliriz. Sayfa içerisinde **ModalRoute.of()** metodu ile gönderilen argümanlara erişilir. Diğer bir alternatif ise **MaterialApp** veya **CupertinoApp** widgetlarında bulunan **onGenerateRoute** parametresi ile gerçekleştirilebilir.

Bir örnek üzerinde bu iki yöntemin nasıl uygulanacağını inceleyelim. Öncelikle gönderilecek argümanlar için bir yapı oluşturulur. Bu yapımız title ve message adında iki alan içermektedir.

```
class ScreenArguments {  
    final String title;  
    final String message;  
    ScreenArguments(this.title, this.message);  
}
```

Bu argüman listesini alıp işleyen bir widget tasarımlı aşağıdaki gibidir.

```
class ExtractArgumentsScreen extends StatelessWidget {  
    const ExtractArgumentsScreen({Key? key}) : super(key: key);  
    static const routeName = '/extractArguments';  
    @override  
    Widget build(BuildContext context) {  
        final args = ModalRoute.of(context)!.settings.arguments as ScreenArguments;  
        return Scaffold(  
            appBar: AppBar(  
                title: Text(args.title),  
            ),  
            body: Center(  
                child: Text(args.message),  
            ),  
        );  
    }  
}
```

Oluşturulan bu widgetin MaterialApp'da routes altında kaydedilmesi;

```
MaterialApp(  
    routes: {  
        ExtractArgumentsScreen.routeName: (context) =>  
            const ExtractArgumentsScreen(),  
    },  
)
```

Son olarak bu sayfaya argüman listesinin gönderilmesi kısmını ele alırsak aşağıdaki gibi Navigator.pushNamed() fonksiyonuna arguments parametresi tanımlanarak sayfa açılır.

```
ElevatedButton(
  onPressed: () {
    Navigator.pushNamed(
      context,
      ExtractArgumentsScreen.routeName,
      arguments: ScreenArguments(
        'Argümanla Açılan Sayfa',
        'Bu mesaj build metodu içerisinde işlenir.',
      ),
    );
  },
),
child: const Text('Argüman listesini gösteren sayfayı aç'),
),
```

Alternatif yöntem olarak MaterialApp widgetındaki onGenerateRoute parametresini kullanalım. onGenerateRoute parametresi RouteSettings olan bir fonksiyon alır.

```
MaterialApp(
  onGenerateRoute: (settings) {
    if (settings.name == PassArgumentsScreen.routeName) {
      final args = settings.arguments as ScreenArguments;
      return MaterialPageRoute(
        builder: (context) {
          return PassArgumentsScreen(
            title: args.title,
            message: args.message,
          );
        },
      );
    }
    assert(false, 'Need to implement ${settings.name}');
    return null;
  },
)
```

Çalışır tam örneği Interactive Example başlığı altından inceleyebilirsiniz.

7.3. Navigator 2.0 API

Flutter’ın sayfalar arası gezintiyi daha bildirimsel hale getirmek için var olan API’sini çerçeveye yeni sınıflar ekleyerek genişlettiği halidir. Amaç web sayfalarındaki URL mantığında gezintiyi daha kolay inşa etmektir. Bu amaçla çerçeveye eklenen sınıfların açıklamaları aşağıdaki gibidir. Eklenen bu yeni özellikler Flutter Design Docs altında Navigator 2.0 and Router başlığıyla paylaşılmıştır.

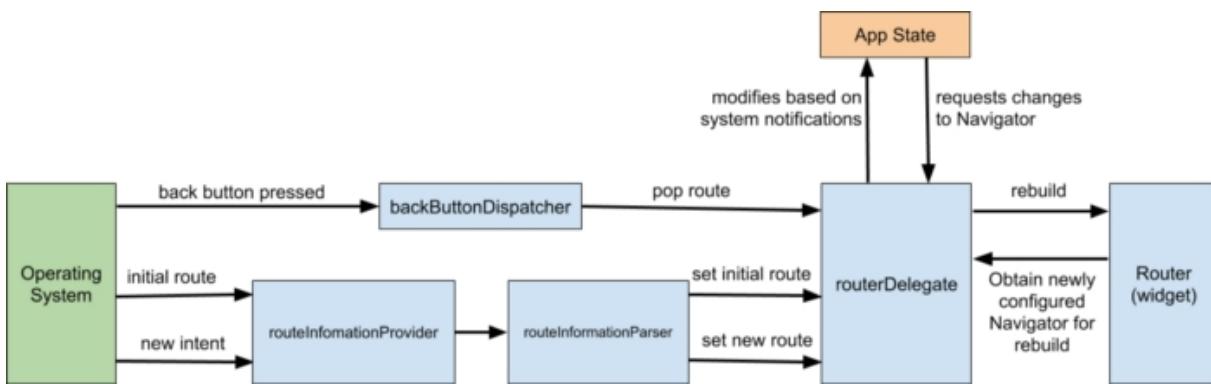
Page: Navigator’ın geçmiş yiğinını oluşturmak için kullandığı değiştirilemez objeyi temsil eder.an

Router: Navigator tarafından gösterilecek sayfaları(Page) kontrol eder. Genellikle sayfalar alttaki platforma bağımlı ve uygulamanın durumuna bağlı olarak değişir.

RouterInformationParser: RouterInformationProvider bileşeninden RouterInformation bilgisini alır ve bunu kullanıcı veri türüne çevirir.

RouterDelegate: Router’ın uygulama içerisindeki durum bilgisindeki değişikliği anaması ve tepki vermesini tanımlar. Temel görevi RouterInformationParser’ı dinlemek ve uygun Page listesi ile Navigator’u oluşturmaktır.

Aşağıdaki diyagramda bu bileşenler arasındaki işleyiş verilmektedir. Navigator 2.0 API'ı kullanım örneği tasarım dokümanından ve özeti John Ryan'ın Medium yazısından incelenebilir.



Bölüm Özeti

Flutter tamamen widgetlardan kurulu bir sayfa düzeni sunsa da sayfalararası geçişleri ve uygulama içerisindeki bölümlemeyi yönetecek mekanizmaları ihtiyaç duyulmaktadır. Flutter'da bu işlemi gerçekleştiren kısma Navigator API adı verilmektedir. MaterialApp ve CupertinoApp (Android ve iOS) görünümü uygulama geliştirmek için temel widgetlar) widgetları bünyelerinde bu Navigator API davranışını taşımaktadırlar. Bunun haricinde Navigator objesi üzerinden tamamen özelleşmiş bir sayfa yönetimi de gerçekleştirilebilir. Sayfa yönetimi akıllı telefonlarda özellikle geri tuşuna basıldığında bir yığından önceki sayfaya dönenbilme, açılan sayfadan bilgi alabilme ve bu sayfaya bilgi iletebilme gibi özellikleri uygulamaya kazandırmaktadır. Navigator API geliştirilerek daha bildirimsel 2.0 versiyonu da hizmete sunulmuştur. Bu versiyonda web sayfalarındaki gibi bir gezinti deneyimi sunmak amaçlanmıştır. Bir uygulama içerisinde hem 1.0 hem de 2.0 API özellikleri aynı anda kullanılabilmektedir.

Kaynakça

<https://flutter.dev/docs/development/ui/navigation>

<https://flutter.dev/docs/cookbook/navigation>

<https://api.flutter.dev/flutter/widgets/Navigator-class.html>

<https://medium.com/flutter/learning-flutters-new-navigation-and-routing-system-7c9068155ade>

https://docs.google.com/document/d/1Q0jx0l4-xymph9O6zLaOY4d_f7YFpNWX_eGbzYxr9wY

8. BÖLÜM

Giriş

Animasyon hazırlamak başlı başına bir geliştirme yüküdür. Bu açıdan çoğu program geliştirici uygulama içerisinde animasyon sunmaktan kaçınır. Flutter bu açıdan ciddi bir avantajla gelmektedir. Arayüzü oluşturan klasik bileşenlerin bir çoğunu animasyonlu sürümü kütüphane içerisinde gelmektedir. Animasyonlu sürümü olmayan arayüz bileşenlerinin de üzerinde kolayca animasyon efektleri uygulanabilir. Konuya incelemeden önce Flutter'da arayüz oluşturan bileşenleri ve yerleşimine hakim olmanızı tavsiye ederim.

8.1 Flutter'da Animasyonlara Giriş

Animasyonlar uygulamaların daha kullanıcı dostu ve ilgi çekici olmasını sağlarlar. Flutter çatısı altında animasyon özelliği bulunan bileşenler bulunmakta ve animasyon hazırlamak için gelişmiş seçenekler yer almaktadır. Flutter'in sunduğu bir çok hazır bileşen(özellikle Material bileşenleri) kendiliğinden hareket (motion) efektlerine sahiptir ama bunları da değiştirme şansımız bulunmaktadır.

Flutter'ın animasyonlarla ilgili dokümantasyonuna <https://flutter.dev/docs/development/ui/animations> Internet adresinden erişilebilirsiniz. Burada Flutter'da animasyon oluşturma çeşitleri dört başlık altında incelenmiştir.

1. Implicit Animations: Flutter çerçevesinde gelen animasyon efektleri uygulamayı sağlayan hazır widgetlar kullanılarak animasyon gerçekleştirmeye yöntemidir. En basit yaklaşımındır ve birçok ihtiyacımızı doğrudan karşılamak için yeterlidir.

2. Explicit Animations: Hazır Animasyonlu widgetlarda bulunmayan düzenlemelerin eklenmesi ile gerçekleştirilen yöntemdir. Bunun için Animasyon kütüphanesindeki; **Animated Builder**, **Transition** sınıfları ve **AnimatedWidget** sınıfı ihtiyaca göre kullanılmaktadır.

3. Low Level Animations: Bu kategori de Animasyon kütüphanesinin kullanımını ele almaktadır. Explicit Animations için tarif edilen yöntemlerin üzerine animasyondaki çizimlerin CustomPainter ile özel olarak tasarlamanızı ifade etmektedir.

4. Third-Party Animation Framework: Bu madde de Flutter'ın kendi sunduğu Animasyon kütüphanesinin dışında bir çerçeve kullanılarak animasyon hazırlanmasını ifade etmektedir. Flutter çok geniş bir çerçevede animasyon hazırlama özellikleri sunsa da üçüncü parti bir çerçeve kullanımı tercih edilebilir. Flutter'ın bu açıdan bir engeli bulunmamaktadır.

Biz bu yaklaşımı zorluk seviyelerine göre üç başlık altında inceleyelim.

8.2. Implicit Animations

Flutter çerçevesi içerisinde temel animasyon efektlerini yerine getirecek bir takım pre-build (hazır) animasyonlu görsel bileşen (animated-widget) ile gelmektedir. Bu sayede çoğu zaman ek bir efor sarf etmeden arayüze animasyon eklememiz mümkün olacaktır.

Flutter'daki pek çok temel Widget için animasyonlu bir sürümü de mevcuttur. Eğer widget'in adı "Foo" ise animasyonlu sürümünün adı "AnimatedFoo" olarak kullanılmaktadır. Örneğin Container widgeti için bir örnek üzerinden farklılığı inceleyelim.

```
//Burada yer alan Consumer ve CounterModel bir sonraki konuda ele alınacaktır.  
//Şimdilik "counter" değerinin her değişiminde arayüzü güncellediğini  
//bilmemiz yeterlidir.  
Consumer<CounterModel>(  
    builder: (_, counter, __) {  
        return Container(  
            height: counter,  
            width: counter,  
            color: Colors.lightBlue,  
            alignment: Alignment.center,  
        );  
    }  
)
```

Yukarıdaki kod blogunda temel Container bileşeni kullanılmıştır. Animasyon efekti olmaksızın bu kod counter değeri her değiştiğinde bir sonraki üitede işlenecek Consumer bileşeni aracılığı ile arayüzü yenilemektedir. Aynı kodu Container bileşeni yerine AnimatedContainer kullanarak aşağıdaki gibi yazmamız durumunda counter değerlerinin her değişiminde iki değer arasındaki geçiş görsel bir animasyona dönecektir.

```

Consumer<CounterModel>(
  builder: (_, counter, __) {
    return AnimatedContainer(
      height: counter,
      width: counter,
      color: Colors.lightBlue,
      alignment: Alignment.center,
      duration: const Duration(seconds: 2),
    );
  }
)

```

AnimatedContainer widgetin yapıcı fonksiyonu, Container widgetinin yapıcı fonksiyonu ile neredeyse aynıdır. Sadece animasyonun ne kadar süreceği ile ilgili olarak ek “duration:” parametresi almaktadır. Burada verilen 2 saniye değeri counter değerleri arasındaki geçiş süresinin 2 saniyede gerçekleşeceğini ifade etmektedir. Container’ın animasyonlu hali ile Container ile verdığımız temel özelliklerin hepsinde animasyon uygulayabilmekteyiz. Yukarıdaki örneğimizde color ve alignment özellikleri sabit tutulmuş bu nedenle bu özellikler üzerinden bir animasyon olmamıştır. Sadece height ve width değerleri counter değişkenine göre bir durum değişimi yaşamış ve bunun üzerinden animasyon uygulanmıştır. color ve alignment değerleri için aşağıdaki gibi tanımlama ekлense bunların da animasyona kavuþtuðunu göreceksiniz.

```

color: counter % 2 == 0 ? Colors.lightBlue : Colors.lime,
alignment: counter < 500 ? Alignment.center : Alignment.topCenter,
curve: Curves.easeInOut,

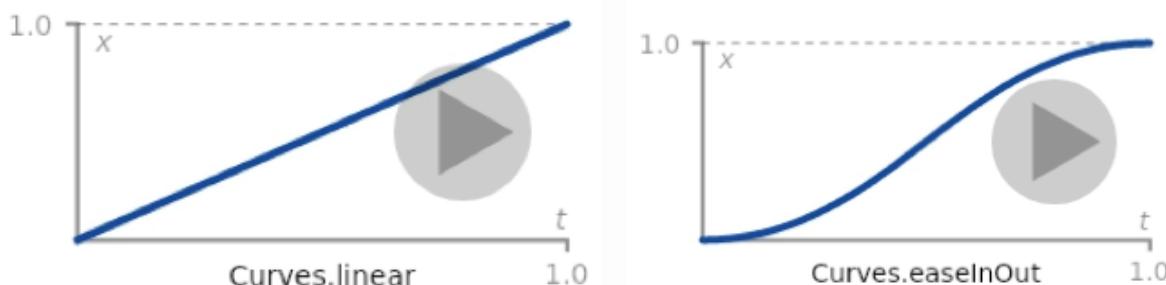
```

Burada animasyonların nasıl icra edileceği ile alakalı bir özellik (**curve**) daha tanımlanmıştır. “curve” parametresi animasyondaki geçişin nasıl olacağını ifade etmektedir. Örneğimizin ilk halinde “curve” parametresi verilmediğinde varsayılan olarak Curves.linear değerini almaktadır. Yani animasyondaki geçiş lineer bir şekilde olacaktır.

Curves sınıfı bu şekilde animasyon geçişleri için oluşturulmuş şablonları sunmaktadır. Farklı animasyon geçiş modelleri için Flutter API dokümantasyonundaki Curves sınıfının sayfası incelenebilir.

<https://api.flutter.dev/flutter/animation/Curves-class.html>

Curves.Linear ve Curves.easeInOut animasyon geçişlerinin değişim grafiği aşağıda verilmektedir. Her bir animasyon geçişinin API dökümanında geçiş grafiğini ve örnek uygulamasını inceleyebilirsiniz.



İlk kendi AnimatedContainer widget örneğinizi kodlamak için aşağıdaki Flutter Cookbook sayfasındaki yönergeleri takip edebilirsiniz.

<https://flutter.dev/docs/cookbook/animation/animated-container>

Az önce ilk implicit animation yaklaşımımız olan AnimatedContainer’ı incelemiş olduk. Şimdi sırasıyla kullanabileceğimiz Implicit Animation yaklaşımlarını yani hazır animasyonlu bileşenleri inceleyelim. Tüm

Implicit animation bileşeninde duration ve curve parametreleri ile animasyon süresini ve geçiş şeklini belirleyebilirsiniz.

- **AnimatedAlign:** Align widgetinin animasyonlu halidir. Align bileşeni altındaki widgetların ekrandaki yerleşimlerini düzenlemektedir. Altındaki bileşenlerin ekran üzerindeki konumlanma şeklini animasyonlu bir geçiş ile değiştirecektir.

```
// 'value' değerinin bir yererde 'int'  
// olarak tanımlandığı ve değiştiriliyor  
// olduğunu varsayıınız.  
AnimatedAlign(  
  alignment: value % 2 == 0 ? Alignment.center :  
  Alignment.topCenter,  
  duration: const Duration(seconds: 1),  
)
```

- **AnimatedDefaultTextStyle:** TextStyle widgetinin animasyonlu halidir. İçerisindeki metinlere uygulanacak sıtildeki değişimi animasyonlu bir hale getirir.

```
// 'fontsize' ve 'color' değerlerinin bir yererde  
// tanımlandığı ve değiştiriliyor  
// olduğunu varsayıınız.  
AnimatedDefaultTextStyle(  
  duration: const Duration(milliseconds: 500),  
  curve: Curves.elasticInOut,  
  style: TextStyle(  
    fontSize: fontSize,  
    color: color,  
    fontWeight: FontWeight.bold,  
,  
  child: const Text("Pasta"),  
)
```

- AnimatedScale
- AnimatedRotation
- AnimatedSlide
- AnimatedOpacity
- AnimatedPadding
- AnimatedPhysicalModel
- AnimatedPositioned
- AnimatedPositionedDirectional
- AnimatedTheme
- AnimatedCrossFade
- AnimatedSize

- AnimatedSwitcher

Örnek olarak verdigimiz “**Implicit Animation**” grubunun hepsi **ImplicitAnimatedWidget** sınıfından miras alınarak oluşturulmuştur. Tüm güncel listeye ve her biri için örnek uygulamalara aşağıdaki bağlantıdan erişebilirsiniz.

<https://api.flutter.dev/flutter/widgets/ImplicitlyAnimatedWidget-class.html>

Flutter dokümantasyonunda Implicit widget oluşturma ve kullanımıyla alakalı video serisi ve diğer makaleler de aşağıda sıralanmaktadır.

<https://flutter.dev/docs/development/ui/animations/implicit-animations>

<https://medium.com/flutter/flutter-animation-basics-with-implicit-animations-95db481c5916>

Aşağıdaki bağlantıda kendi kendinize bir Implicit Animation örneği gerçekleştireceğiniz laboratuvar çalışması bulunmaktadır.

<https://flutter.dev/docs/codelabs/implicit-animations>

8.3. Tween Animation

Flutter “Implicit Animation” kategorisinde sunduğu hazır (pre-build) animasyon modelleri ile temel animasyon işlevsellliğini çok kolay uygulamamıza yansımamızı sağlamaktadır. Peki istediğimiz arayüz bileşeni için bir “Implicit Animation” varyantı bulamazsa ne yapmalıyız? Daha sonra ele alacağımız daha karmaşık ve daha derinlemesine kodlama gerektiren yöntemlerden önce yönelikmemiz gereken ilk çözüm **TweenAnimationBuilder** kullanımıdır. Bu sınıf da **ImplicitAnimatedWidget** sınıfından miras alınarak oluşturulmuş bir Implicit Animation yöntemidir. Diğer Implicit yöntemlerinden farklı olarak **TweenAnimationBuilder’da** animasyonun gerçekleşeceği değer aralıkları **Tween<T>** sınıfı aracılığı ile tanımlanabilmektedir.

Tween sınıfı farklı veri türleri ile çalışabilmektedir. Bunları ColorTween, RectTween, Tween<double> gibi örnöklememiz mümkündür. Aşağıda TweenAnimationBuilder’ın bizim oluşturacağımız bir widgetin build metodu içerisinde kullanımına örnek verilmektedir.

```
Widget build(BuildContext context) {
  return TweenAnimationBuilder<double>(
    tween: Tween<double>(begin: 0, end: targetValue),
    duration: const Duration(seconds: 1),
    builder: (BuildContext context, double size, Widget? child) {
      return IconButton(
        iconSize: size,
        color: Colors.blue,
        icon: child!,
        onPressed: () {
          setState(() {
            targetValue = targetValue == 24.0 ? 48.0 : 24.0;
          });
        },
      );
    },
    child: const Icon(Icons.aspect_ratio),
  );
}
```

Örneğimizde TweenAnimationBuilder <double>, double veri tipi üzerinden bir animasyon geçisi yapılacağını belirtilerek çağrılmıştır. Bu çağrıya karşılık olarak yapıcı fonksiyonun aldığı tween parametresi

`Tween<double>` türünde bir sınıf almaktadır.

```
tween: Tween<double>(begin: 0, end: targetValue),
```

Yukarıdaki satır ile 0 ile `targetValue` arasında değişeceğin bir değerin animasyon için kullanımı sağlanmaktadır. `tween` parametresi animasyon süresi boyunca bu aralıkta animasyonun çizim yöntemine göre değerleri animasyona besleyecektir. `TweenAnimationBuilder` widgeti `tween` parametresinin yanında `duration` ve `builder` adında parametreler de almaktadır. `duration` parametresi diğer Implicit Animationlarda olduğu gibi animasyonun süresini belirtecektir. `builder` parametresi ise animasyonun çiziminde tetiklenecek kısmı almaktadır. (Bu parametre kullanımını ve çağrılan fonksiyon yapısını daha önce görmüştük.)

`ImplicitAnimatedWidget`'tan miras alarak oluşturulan tüm animasyon özelliği gösteren bileşenleri aslında gizliden kullandığı bir **AnimationController** sınıfı bulunmaktadır. Bir sonraki başlıkta bu sınıf ve animasyon kütüphanesinin diğer özelliklerinin kullanımını ele alınacaktır.

`TweenAnimationBuilder`'ın Flutter API dokümantasyonuna aşağıdaki bağlantıdan erişebilirsiniz.

<https://api.flutter.dev/flutter/widgets/TweenAnimationBuilder-class.html>

Aşağıdaki bağlantıda `TweenAnimationBuilder`'ın kullanımı ile ilgili Flutter ekibinin hazırlamış olduğu anlatım ve videoyu inceleyebilirsiniz.

<https://medium.com/flutter/custom-implicit-animations-in-flutter-with-tweenanimationbuilder-c76540b47185>

<https://youtu.be/6KiPEqzJIQ>

8.4. Animasyon Kütüphanesi

Flutter'ın hazır olarak sunduğu animasyonlu arayüz bileşenleri ihtiyacınızı görecek animasyon geçişleri üretmemiyorsa bu durumda animasyon kütüphanesindeki özel tanımlamaları kullanarak kendi animasyonunuza tasarlamanız (Explixit Animation) gerekmektedir. Bu noktada temel bazı trigonometri ifadelerine hâkim olmanız gerekmektedir.

Animasyonların oluşması için kullanılan yapı aşağıdaki bloklardan oluşmaktadır:

- Schedulers
- Tickers
- Simulations
- Animatables
 - o Tweens
 - Curves
 - Animations

Bir animasyonun oynatılması için bu yapı bloklarından bazıları aşağıda incelenecaktır.

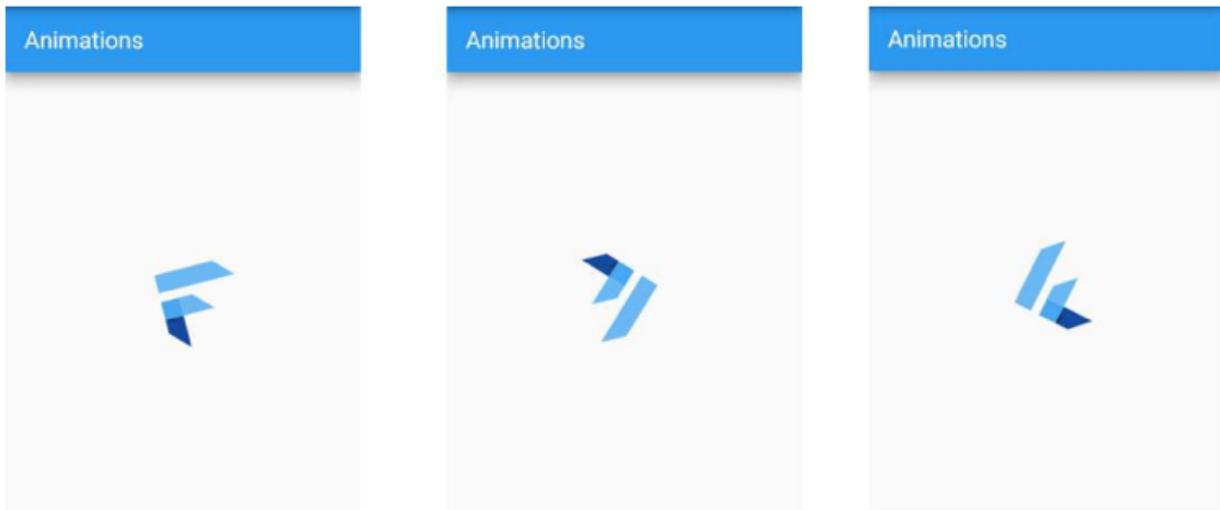
Animasyonun durumu hakkında Animasyon sınıfı içerisinde bilgi sahibi olabilmemiz için **AnimationStatus** enum değeri kullanılır. Bu enum tanımlamasındaki değerler:

- completed: Animasyonun bittiğini belirtir.
- dismissed: Animasyonun başlangıçta durdurulduğunu belirtir.

- forward: Animasyonun ileri yönlü çalıştığını belirtir.
- reverse: Animasyonun ters yönde çalıştığını belirtir.
- values: Enum içerisindeki tüm değerlerin oluşma sırasına göre olduğu sabit bir liste sunar.

8.4.1. AnimatedWidget

Bir animasyonun üretim süreci genel olarak iki adımdan oluşmaktadır. Bu adımları bir Flutter logosunun ekranın ortasında çevrildiği bir örnek üzerinden anlatacağız. Örnek uygulamanın farklı zaman aralıklarında yakalanana ekran görüntüleri aşağıdakine benzemektedir.



Bu örnekte FlutterLogo() adında bir widget Flutter logosunu göstermek için kullanılacaktır. Bu logo vektörel bir imaj kullanıldığındá farklı ekran boyutlarında aynı kalitede görüntülenecektir. Logonun döndürülmesi için gerekli animasyon kodlaması aşağıdaki gibi yapılacaktır:

1. Amacımız FlutterLogo adında bir sınıfı sonsuz bir döngüde(loop) döndürme(rotation) işlemine tabi tutmaktır. AnimatedWidget bu amaç için ilgili arayüz bileşenini alarak üzerinde istenilen animasyon efektlerinin gerçekleştirilmesi için kullanılır. Burada RotatingLogo adında oluşturduğumuz sınıf AnimatedWidget sınıfından miras almaktadır.

```
// 'AnimatedWidget' herhangi bir widgeta uygulanabilir.
class RotatingLogo extends AnimatedWidget {
  final AnimationController _controller;
  const RotatingLogo({required AnimationController controller})
      : _controller = controller,
        super(listenable: controller);
  static const _fullRotation = 2 * math.pi; // 360 derece
  @override
  Widget build(BuildContext context) {
    return Transform.rotate(
      angle: _controller.value * _fullRotation,
      child: const FlutterLogo(
        size: 80,
      ),
    );
  }
}
```

Örneğimizi incelemeye başlamadan önce özellikle animasyonlar konusunda daha önem arz ettiği için şu noktanın altını çizelim. Flutter'da arayüz bileşenlerinin çiziminde pek çok yerde yapıcı fonksiyonların çağrılarında **const** anahtar kelimesinin geçtiğini görmüşünüzdür. **const** yapılandırıcı çağrıları Flutter çerçevesine bu widget için her seferinde baştan çizim yapmamasını söylemektedir. Animasyonlarda her sahnede çizimin tekrarlanacağını düşünürseniz performans açısından const kullanımı önemlidir. Flutter çerçevesi 120fps'ye kadar yüksek bir yenileme hızı sunabilmektedir. Fakat arayüzdeki gereksiz tazeleme talepleri bu oranı azaltabilir.

```
angle: _controller.value * _fullRotation,
```

Eğer animasyonumuz kendi ekseni etrafında tam bir dönme yapacaksa angle parametresinin 0 ile 30 derece arasında değerler alması gereklidir. Flutter açı değerlerini radyan cinsinden almaktadır.

Transform.rotate() yapılandırıcısı bize döndürme işlemi uygulanmış bir FlutterLogo verecektir.

Son olarak dikkat çekenimiz nokta bir AnimationController nesnesinin kullanımı ve bunun dependency injection yöntemine uygun olarak **listenable**: parametresine besleniyor olmasıdır. Bu sayede **_controller.value**'daki değişimler doğrudan animasyona yansıyacaktır.

2. Şimdi yapmamız gereken animasyonu gerçekleyecek olan kısmın hazırlanması ve widget ağacımızın içerisinde yerleştirilmesidir. Bu amaçla **AnimationController** sınıfı bize belli bir frekans ve süreye göre çalışacak bir veri üretmektedir. Örneğin AnimatedController'da 0-1 arası değer ürtülmemesini istersek **AnimationController** ilgili **AnimatedWidget'a** 0 , ... , 0.1 , ... , 0.6 , 0.9 ,... 1 aralıklarında değişen bir değer uretecektir. Örneğimizde **_controller** değişkeni bir AnimationController objesini referans almakta ve **_controller.value** ile bu değişim gösteren degere erişilmektedir.

```
angle: _controller.value * _fullRotation,
```

Burada AnimationController'dan gelen değer **_fullRotation** ile (360 derecenin radyan karşılığı) ile çarpılarak yapılacak döndürme işleminin açısı belirlenmektedir. Şimdi ilgili AnimationController'in oluşturulması ve bağlanması kodunu ele alalım:

```

class FLSpinner extends StatefulWidget {
  const FLSpinner();
  @override
  _FLSpinnerState createState() => _FLSpinnerState();
}

class _FLSpinnerState extends State<FLSpinner> with TickerProviderStateMixin {
  late final AnimationController _controller;
  @override
  void initState() {
    super.initState();
    _controller = AnimationController(
      duration: const Duration(seconds: 3),
      vsync: this,
    )..repeat();
  }
  @override
  void dispose() {
    _controller.dispose();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    return RotatingLogo(controller: _controller);
  }
}

```

Bu aşamada Animasyon ekrandan çıktıgı zaman sistem kaynaklarını geri vermek adına dispose metodunu çağrılmak için StatefulWidget kullanılması gerekmektedir. Aksi halde sistem kaynakları boş tüketilmiş olacaktır. Animasyon için hazırlık ve temizleme aşamalarının initState ve dispose metodlarında yazımının unutulmaması gerekmektedir.

```
late final AnimationController _controller;
```

Cihaz hazır olduğunda sürekli değer üretimi için AnimationController kullanılacaktır. Aksi belirtilemediği sürece AnimationController 0 ile 12 arasında değerler üretmektedir. Normalde AnimationController bir saniyede 60 yeni değer üretmektedir. Eğer yoğun bir animasyon veya telefonun yoğun kullanımı söz konusu ise bu değer düşebilir. Görünmüyor olsa da birden fazla animasyonun eş zamanlı olarak koşturulması bir performans düşüklüğüne sebep olabilir.

repeat() metodunun kullanımı ilgili animasyonun sonsuza dek oynatılmasını talep etmektedir. Bu örneğimizde her bir animasyon döngüsü 3 sn sürecek ve sonsuza dek tekrar edecktir.

AnimationController animasyonun yönü için bize 3 farklı seçenek sunmaktadır;

- forward: ($0 > 1$) animasyondaki değişen değer 0'dan 1'e doğru verilecektir.
- reverse: ($1 > 0$) animasyondaki değişen değer 1'den 0'a doğru verilecektir.
- repeat: ($0 > 1, 0 > 1, 0 > 1, \dots$) animasyondaki değişen değer 0'dan 1'e döngüsel olarak sürekli verilecektir.

```
vsync: this,
```

vsync parametresi animasyonun ekrandan çıktıktan sonra çalışmasını engellemektedir. Böylece gereksiz sistem kaynağı tüketilmez. Burada this değerinin verilmesi animasyonu gerçekleştiren StatefulWidget üzerinden bu işlemin yürütülmesini sağlayacaktır.

```
.. with TickerProviderStateMixin { }
```

StatefulWidget oluşturularken vsync özelliğinin kullanılabilmesi için miras alma sürecine animasyonlar tarafında kullanılan TickerProvider sınıfının bir örneği olan TickerProviderStateMixin mixin'in eklenmesi gerekmektedir.

```
runApp( MaterialApp(
  home: Scaffold(
    appBar: AppBar(title: const Text("Animations")),
    body: const Center(
      child: FLSpinner(),
    ),
  ),
));
```

Yukarıdaki kod içerisinde oluşturduğumuz FLSpinner yapıcı fonksiyonu çağrılarak animasyonumuzu bir uygulama içerisinde kullanmış olduk. AnimatedWidget ve AnimationController kullanımı ile iki aşamalı olarak kendi özel animasyonumuzu nasıl yapabileceğimizi deneyimledik. Burada farklı transformasyon yöntemleri kullanarak değişik geçişli animasyonlar yapmak mümkündür.

Flutter ekibinin bu konuda hazırlamış olduğu makale ve videoyu aşağıdaki bağlantılardan inceleyebilirsiniz.

<https://medium.com/flutter/directional-animations-with-built-in-explicit-animations-3e7c5e6fbbd7>

<https://youtu.be/CunyH6unILQ>

8.4.2. AnimatedBuilder

AnimatedBuilder aslında AnimatedWidget sınıfının optimize edilmiş ve gereksiz yeniden inşaları engelleyen basitleştirilmiş formudur. Animasyonları kullanırken nasıl çalışıklarını anlamak uygulamanızın performansını düşürmemek adına önemli bir noktadır.

Bir önceki başlıkta Flutter'ın her animasyon karesinde alt elemanları baştan çizmemesi için const yapılandırıcı fonksiyonları çağrılmamız gerektiğini söylemiştim. Ama her Flutter widget'ı için const bir yapılandırıcı çağrıSİ bulunmaya bilir. Daha da kötüsü bu bileşenlerin iç içe geçmiş bir kümesi animasyona konu olabilir. Bu durumda tüm widget ağacının her animasyon karesinde yeniden çizimi gerekeceği için uygulama performansı ciddi manada kötü etkilenecektir. Bu gibi durumlarda sergilenebilecek iki yaklaşım söz konusudur. İlk animasyonda oynatılacak widget ağacının manuel olarak tamponlanması ve bunun animasyon çiziminde kullanımını sağlayacak kodlamanın yapılması ikincisi ise AnimatedBuilder kullanımızdır.

AnimatedBuilder bu aşamada elle yapılması gereken birçok işlemi (tamponlama gibi) bizim yerimize halledeceği için uygun yaklaşım olacaktır.

```

@Override
Widget build(BuildContext context) {
  return AnimatedBuilder(
    animation: someController,
    builder: (context, child) {
      return Transform.rotate(
        angle: _controller.value * _fullRotation,
        child: child,
      );
    }
    child: Container(...),
  );
}

```

Burada child parametresi widget ağacında animasyonun içerisinde kalacak ve tamponlanacak kısmı tutacaktır. Yani child parametresi içerisinde verilen widget ağacı sadece bir defaya mahsus oluşturulacaktır.

Flutter ekibinin AnimatedBuilder ve AnimatedWidget kullanımı ile ilgili makalesine ve videosuna aşağıdaki bağlantılarından erişebilirsiniz.

<https://medium.com/flutter/when-should-i-useanimatedbuilder-or-animatedwidget-57ecae0959e8>

https://youtu.be/fneC7t4R_B0

8.4.3 Curves

Her animasyon standart olarak lineer bir zaman döngüsüne sahiptir. Yani üretilen değerler belirtilen aralık içerisinde eş zaman aralıklarında değişirler. Animasyonlar konusunun başında bahsedildiği üzere Curves sınıfındaki tanımlı geçiş efektleri kullanılarak değer değişim zaman çizelgesi lineerden farklı yaklaşımlara da ayarlanabilir. Misal alınan değerlerin zıplama efekti ile değişmesi daha ilgi çekici bir animasyon oluşturacaktır.

```

late final AnimationController _controller;
late final CurvedAnimation _curved;
@Override
void initState() {
  super.initState();
  _controller = AnimationController(
    duration: const Duration(seconds: 2),
    vsync: this,
  )..repeat();
  _curved = CurvedAnimation(
    parent: _controller,
    curve: Curves.bounceIn,
  );
}

Widget build(BuildContext context) {
  return RotatingLogo(controller: _curved);
}

```

CurvedAnimations kullanarak Curves sınıfında tanımlı farklı geçiş efektlerini kullanabiliriz.

8.4.4. Tweens

AnimatonController'ın varsayılan olarak 0 -1 aralığında double değerler ürettiğini söylemişik. Bu aralığın farklı bir skalada veya farklı bir veri türü üzerinden yapılması için Tween<T> sınıfı kullanılır.

```
late final AnimationController _controller;
late final Animation<double> _tween;
@Override
void initState() {
  super.initState();
  _controller = AnimationController(
    duration: const Duration(seconds: 3),
    vsync: this,
  )..repeat();
  _tween = Tween<double>(
    begin: 0,
    end: 2 * math.pi,
  ).animate(_controller);
}
```

Buradaki tween objesi AnimationController'ın 0-1 aralığında değer üretmek yerine 0-6.2831.. aralığında değer üretmesini sağlayacaktır.

Farklı veri türleri için kullanılabilen özel Tween varyantları mevcuttur. Bunlardan bazıları;

- BorderRadiusTween
- IntTween
- DecorationTween
- ShapeBorderTween
- SizeTween

Şeklinde sıralanabilir. Tam bir liste için API dokümantasyonuna bakınız.

<https://api.flutter.dev/flutter/animation/Tween-class.html>

Eğer bir animasyonda birden fazla geçiş olan bir yaklaşım kullanmak istiyorsanız bunun için **TweenSequence** sınıfını kullanabilirsiniz. **TweenSequence** sınıfı her bir geçiş aralığını **TweenSequenceItem** olarak tanımlayacağınız bir liste ile bu geçişleri yürütecektir. Aşağıda bununla ilgili Flutter API'sinde verilen örnek sunulmaktadır.

```

final Animation<double> animation = TweenSequence<double>(
  <TweenSequenceItem<double>>[
    TweenSequenceItem<double>(
      tween: Tween<double>(begin: 5.0, end: 10.0)
        .chain(CurveTween(curve: Curves.ease)),
      weight: 40.0,
    ),
    TweenSequenceItem<double>(
      tween: ConstantTween<double>(10.0),
      weight: 20.0,
    ),
    TweenSequenceItem<double>(
      tween: Tween<double>(begin: 10.0, end: 5.0)
        .chain(CurveTween(curve: Curves.ease)),
      weight: 40.0,
    ),
  ],
).animate(myAnimationController);

```

Daha fazla ayrıntı için Flutter API dokümantasyonundaki TweenSequence sınıfının sayfası incelenebilir.

<https://api.flutter.dev/flutter/animation/TweenSequence-class.html>

8.5. Özel Animasyonlar

Flutter'da istenilen her tür animasyon efektini yapmak mümkündür. Bu aşamaya kadar animasyon için gerekli temel bileşenlerden bahsettik. Bunların dışında 3D dönüşümlerin uygulanması için farklı animasyon efektleri de kullanabiliriz. Bu efektlerin kullanımı için matrisler ve trigonometri kurallarına hakim olmak gerekebilir. Bu tarz özel bir animasyon hazırlamak diğer bir bakış açısıyla aşağıdaki üç sınıfın işleyişine hakim olma ile de gerçekleştirilebilir.

- **AnimationController:** Verilen frekansta animasyonun üretilmesini sağlayacak
- **Transform:** Ölçekleme, döndürme, kaydırma, büzme gibi işlevleri matrisler aracılığı ile yapacak
- **Stack:** Widget ağacında children olarak belirtilen görsel objeleri üst üste bindirecek



Yukarıdaki resimdeki gibi farklı zaman dilimlerinde döndürülme efekti uygulanan bir Container için aşağıdakine benzer bir kod yazılabılır. Burada Transform sınıfı Container üzerinde uygulanacak dönüşüm işlemlerini yapmaktadır.

```

@Override
Widget build(BuildContext context) {
  return AnimatedBuilder(
    animation: _controller,
    builder: (context, child) {
      return Transform(
        transform: Matrix4.skewY(0.1)
          ..rotateY(_controller.value * 2 * math.pi),
        child: child,
      );
    },
    child: Container(
      decoration: const BoxDecoration(...),
      child: ...
    ),
  );
}

```

Transform işlemleri karmaşık matematiksel hesaplamalar içereceği için burada daha fazla detaya inilmeyecektir. Tüm dönüşüm işlemleri ile ilgili detay API dökümanlarından incelenebilir.

<https://api.flutter.dev/flutter/widgets/Transform-class.html>

Flutter animasyonlarının nasıl çalıştığı ile ilgili olarak Flutter ekibinin hazırlamış olduğu makale ve videoya aşağıdaki bağlantılardan ulaşabilirsiniz.

<https://medium.com/flutter/animation-deep-dive-39d3ffea111f>

<https://youtu.be/PbcILiN8rbo>

Bir diğer animasyon üretme şekli olarak animasyon efektlerinin üst üste bindirilmesi ile kademeli bir animasyon elde edilebilir. Bununla ilgili örnek Flutter dokümantasyonunda aşağıdaki bağlantıda anlatılmıştır.

<https://flutter.dev/docs/development/ui/animations/staggered-animations>

8.6. Hero Animasyonlar

Flutter’ın sunduğu diğer bir animasyon modeli de master-detail şeklinde hazırlanmış iki sayfa arası geçişlerde kullanılan Hero Animasyonlarıdır. Bir alışveriş sitesindeki ürün katologunda seçilen bir ürünün detay sayfasının açılmasında ürün resminin genişleyerek ekrana gelmesi gibi efektlerin verilmesinde kullanılır. Sayfalar arası navigasyon konusu ile iç içe olan bir başlık olduğu için Flutter dokümantasyonunda sayfalar arası gezinme konu başlığında ele alınmaktadır.

Aşağıdaki bağlantıdan iki sayfa arası geçişinde bir Hero Animation hazırlanmasında gereken adımlar verilmektedir.

<https://flutter.dev/docs/cookbook/navigation/hero-animations>

```
import 'package:flutter/material.dart';

void main() => runApp(const HeroApp());

class HeroApp extends StatelessWidget {
  const HeroApp({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      title: 'Transition Demo',
      home: MainScreen(),
    );
  }
}

class MainScreen extends StatelessWidget {
  const MainScreen({Key? key}) : super(key: key);

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('Main Screen'),
      ),
      body: GestureDetector(

```

```

        onTap: () {
            Navigator.push(context, MaterialPageRoute(builder: (context) {
                return const DetailScreen();
            }));
        },
        child: Hero(
            tag: 'imageHero',
            child: Image.network(
                'https://picsum.photos/250?image=9',
            ),
        ),
    ),
),
);
}
}

class DetailScreen extends StatelessWidget {
const DetailScreen({Key? key}) : super(key: key);

@Override
Widget build(BuildContext context) {
    return Scaffold(
        body: GestureDetector(
            onTap: () {
                Navigator.pop(context);
            },
            child: Center(
                child: Hero(
                    tag: 'imageHero',
                    child: Image.network(
                        'https://picsum.photos/250?image=9',
                    ),
                ),
            ),
        ),
    );
}
}

```

Master-detay sayfaları arasındaki geçişler Navigator.push ve Navigator.pop metodları aracılığıyla yapılmaktadır. MainScreen ve DetailScreen sınıfları içerisinde Hero widgetinin kullanımına dikkat ederseniz, her iki tanımda da tag parametresinin aynı olduğunu göreceksiniz. Bu parametre farklı iki Hero objesi arasında geçiş animasyonu olacağını bildirecektir ve bu neden aynı değer olmalıdır. child parametresi Hero animasyonu içerisinde kalacak (değişecek) alanın ifade edildiği parametredir. Bu örnekte ana ve detay görünümde aynı resim kullanılmıştır.

Flutter dokümantasyonunda Hero Animasyonlarının nasıl hazırlanacağıyla ilgili iki farklı örnek verilmektedir. Bu örnekler ve Hero animasyon ile ilgili dokümantasyona aşağıdaki linkten erişebilirsiniz.

<https://flutter.dev/docs/development/ui/animations/hero-animations>

8.7. Gelişmiş Arayüz Özellikleri

Flutter çerçevesi mobil platformlardaki arayüzlerin oluşturulması için klasik form bileşenlerinin dışında gelişmiş özel bileşenler de içermektedir. Ayrıca masaüstü ve web desteği için klavye kısayolları gibi özellikler de çerçeveye dahil edilmektedir. Bu başlıkta bu konular için referanslar verilecektir. Flutter'a yeni eklenen Actions, Shortcuts, Keyboard Focus konuları ders materyaline eklenmemiştir. Bu konular ile ilgili Flutter dokümanlarına aşağıdaki bağlantılarından erişebilirsiniz.

https://flutter.dev/docs/development/ui/advanced/actions_and_shortcuts

<https://flutter.dev/docs/development/ui/advanced/focus>

8.7.1. Gestures

Kullanıcı etkileşimleri Flutter'da tıklama, sürükle bırak ve dokunmatik ve donanımsal klavye üzerinden olabilmektedir. Dokunmatik ekrandan yapılan girişleri Flutter çerçevesi Gesture denilen bir sistem ile işlemektedir. Gesture sistemi iki kademeli bir yapıda kullanıcı etkileşimlerini işlemektedir. Bu yapının ilk kademesi hareketleri ve konumları salt işaretçiler olarak tanımlayan Pointer katmanı ikincisi ise bir veya birden fazla işaretçi hareketini anlamlıdan gestures katmanıdır.

Flutter'da tanımlı olan gesture olayları aşağıda sıralanmıştır.

Tap işlemi:

onTapDown: Bir işaretçinin ekran üzerinde belli bir konuma tıklama işlevini başladığını ifade eder.

onTapUp: Bir işaretçinin ekran üzerinde belli bir konuma tıklama işlevini terk ettiğini ifade eder

onTap: Bir işaretçinin önce onTapDown ardından onTapUp sürecini tamamlayarak bir tıklama işlevini gerçeklediğini gösterir.

onTapCancel: onTapDown ile başlayan bir tıklama işleminin onTapUp ile bitirilmediği durumu ifade eder.

Double Tap işlemi:

onDoubleTap: Kullanıcının ekranda belli bir konuma hızlıca çift tıklama yapmasını ifade eder.

Long Press (Uzun Basma):

onLongPress: Bir işaretçinin ekrandaki belli bir konuma uzun süre basılı durmasında tetiklenir.

Vertical Drag (Dikey Sürükleme):

onVerticalDragStart: Ekranda bir işaretçi ile dikey sürüklemenin başlamasını ifade eder.

onVerticalDragUpdate: Ekrana temas eden bir işaretçinin dikey konumda değişim yaptığında tetiklenir.

onVerticalEnd: Ekrana temas eden bir işaretçinin dikey konumdaki yer değiştirme işlemini bırakması ve işaretçinin teması kesmesini ifade eder. Bu süreçte işaretçi ekranda dikeyde belli bir ivme ile gitmektedir.

Horizontal Drag (Yatay Sürükleme):

onHorizontalDragStart, onHorizontalDragUpdate ve onHorizontalDragEnd vertical için olan fonksiyonların yataydaki karşılıklarıdır.

Pan

onPanStart: Bir işaretçinin ekrana dokunduktan sonra dikeyde veya yatayda hareket etmeye başlaması durumunda tetiklenir. onVerticalDragStart veya onHorizontalDragStart uygulanırsa hata verir.

onPanUpdate: Bir işaretçinin yatayda veya dikeyde sürükleme işlemeye devam ettiğince tetiklenmektedir. onVerticalDragUpdate ve onHorizontalDragUpdate uygulanırsa hata verir.

onPanEnd: Bir işaretçinin yatayda veya dikeyde sürükleme işlevini bitirdiği (kestiğini) ifade etmektedir.

Ekrandan gestureların yakalanması için Flutter GestureDetector adlı widgetı kullanmaktadır.

8.7.2. Silvers

Silver bileşenleri arayüzde sürüklenebilir bileşenlerin daha efektif olmalarını sağlamaktadır. Silver bileşenler Material tasarımda uygulanmaktadır. Bir sayfa içerisinde **SilverAppBar**, **SilverList** ve **SilverGrid** bir arada kullanılarak ardışık olarak sürüklenebilen bir sayfa tasarımı oluşturulur. Silver kullanımının avantajı ekranı kaydırığınızda AppBar kısmının daralması ve gizlenmesi sürecini otomatik yapmasıdır. Aşağıda bu bileşenlerin tanıtımı yapılan videolar için linkler bulunmaktadır.

[SilverAppBar: https://youtu.be/R9C5KMJKluE](https://youtu.be/R9C5KMJKluE)

[SilverList ve SilverGrid: https://youtu.be/ORiTTaVY6mM](https://youtu.be/ORiTTaVY6mM)

Kullanım örnekleri için ayrıca Flutter API dokümanlarına bakınız.

[SliverAppBar: https://api.flutter.dev/flutter/material/SliverAppBar-class.html](https://api.flutter.dev/flutter/material/SliverAppBar-class.html)

[SliverGrid: https://api.flutter.dev/flutter/widgets/SliverGrid-class.html](https://api.flutter.dev/flutter/widgets/SliverGrid-class.html)

[SliverList: https://api.flutter.dev/flutter/widgets/SliverList-class.html](https://api.flutter.dev/flutter/widgets/SliverList-class.html)

Bölüm Özeti

Bu bölümde Flutter'da animasyon hazırlama ve diğer bazı arayüz özelliklerine degenilmiştir. Flutter basit animasyon hazırlama (**Implicit Animations**) ve daha detaylı animasyon hazırlama (**Explicit Animations**) yöntemlerini sunmaktadır. Ayrıca istenildiği takdire harici animasyon paketleri de kullanılabilir. Klavye ve Dokunmatik ekrandan kullanıcı girişleri için Flutter'in desteği bulunmaktadır. Dokunmatik ekrandan girişler için **Gestures** en yaygın kullandığımız Flutter bileşenidir. **Silver**lar ile liste ve ızgara görünümleri için kolay kaydırma ve ekranı efektif kullanma süreçlerini zahmetszizce yapabiliriz.

Kaynakça

<https://flutter.dev/docs/development/ui/animations>

<https://api.flutter.dev/flutter/animation/Curves-class.html>

<https://flutter.dev/docs/cookbook/animation/animated-container>

<https://api.flutter.dev/flutter/widgets/ImplicitlyAnimatedWidget-class.html>

<https://flutter.dev/docs/development/ui/animations/implicit-animations>

<https://medium.com/flutter/flutter-animation-basics-with-implicit-animations-95db481c5916>

<https://flutter.dev/docs/codelabs/implicit-animations>

<https://api.flutter.dev/flutter/widgets/TweenAnimationBuilder-class.html>

<https://medium.com/flutter/custom-implicit-animations-in-flutter-with-tweenanimationbuilder-c76540b47185>

<https://youtu.be/6KiPEqzJIKQ>

<https://medium.com/flutter/directional-animations-with-built-in-explicit-animations-3e7c5e6fbcd7>

<https://youtu.be/CunyH6unILQ>

<https://medium.com/flutter/when-should-i-use-animatedbuilder-or-animatedwidget-57ecae0959e8>

https://youtu.be/fneC7t4R_B0

<https://api.flutter.dev/flutter/animation/Tween-class.html>

<https://api.flutter.dev/flutter/animation/TweenSequence-class.html>

<https://api.flutter.dev/flutter/widgets/Transform-class.html>

<https://medium.com/flutter/animation-deep-dive-39d3ffea111f>

<https://youtu.be/PbcILiN8rbo>

<https://flutter.dev/docs/development/ui/animations/staggered-animations>

<https://flutter.dev/docs/cookbook/navigation/hero-animations>

<https://flutter.dev/docs/development/ui/animations/hero-animations>

https://flutter.dev/docs/development/ui/advanced/actions_and_shortcuts

<https://flutter.dev/docs/development/ui/advanced/focus>

<https://youtu.be/R9C5KMJKluE>

<https://youtu.be/ORiTtaVY6mM>

<https://api.flutter.dev/flutter/material/SliverAppBar-class.html>

<https://api.flutter.dev/flutter/widgets/SliverGrid-class.html>

<https://api.flutter.dev/flutter/widgets/SliverList-class.html>

9. BÖLÜM

Giriş

Flutter'da ekranın hazırlanması bildirimsel(declarative) bir bakış açısından yapılmaktadır. Eğer daha önce bu tarz çalışan bir çerçeve kullandığınız (React-native gibi) bu bölümde anlatılanlar tanıdık gelecektir. Eğer daha önce sadece C#, Java, Kotlin gibi dillerle klasik işletim sistemine yönelik uygulama geliştirdiyseniz bildirimsel yaklaşımalar alışla gelinenden çok farklı bir şekilde arayüzü hazırlamaktadır.

9.1. Flutter'da Durum Yönetimi

Önceki konularda StatelessWidget ve StatefulWidget kavramlarını gördük. Bunlar arasındaki temel farkın durum (state) takibi olduğunu ve Flutter'in arayüz güncellemesini tetiklemek için bir durum değişimini takip ettiğini söylemişlik. Bunun için StatefulWidget ile setState() metodunun kullanımına baktık. Ama StatefulWidget kullanımında şöyle bir kısıdımız bulunmaktadır. StatefulWidget arayüzde güncelleme yapılacak alanın üzerinde bir yerde widget ağacında konumlandırılmalıdır. Aksi halde arayüz güncellemesini yapılamayacaktır. Buna karşın StatefulWidget'in kök eleman olarak kullanılması da performans açısından kötü bir yaklaşımdır. Çünkü veri (durum) değişiminde StatefulWidget altındaki tüm widget ağacının baştan çizilmesi tetiklenecektir. Bu durum istenmemektedir. Bir çok arayüzde veri değişiminde etkilenen alanlar bağımsız konumlarda olabilmektedir. Hatta bir alışveriş uygulaması düşünürseniz sepete eklenen ürünler aslında başka bir sayfa açıldığında veri manipülasyonu yapmalıdır. Yani Navigator.push, .pop işlemlerinden sonra arayüzün güncel halinin getirilmesi gereklidir. Bu durumlar için StatefulWidget ihtiyacımızı karşılayamaz.

Daha önce React-native veya benzeri durum yönetimi olan bir yaklaşım ile kodlama yaptıysanız Flutter'daki durum yönetimine kolayca uyum sağlayabilirsiniz. Daha önce böyle bir kodlama yapmayan ve .Net veya Java gibi platformlarda zorunlu (imperative) kodlama yapanlar için öncelikle bildirimsel (declarative) yaklaşımı anlamaları gerekmektedir.

9.2. Bildirimsel (Declarative) Düşünme

Flutter arayüzün oluşturulmasında bildirimsel bir mantık kullanmaktadır. Bu Android ve iOS API'larının kullandığı zorunlu (imperative) yaklaşımından farklıdır.



Bildirimsel yaklaşımın arayüzü elde etmek için kullandığınız fonksiyon uygulamanın o anki durumunu alır ve buna göre arayüz çizimini gerçekleştirir. Uygulamanın durumunda bir değişiklik olduğunda arayüz yeniden çizilmesi gereklidir ve Flutter'da arayüzü çizecek build metotları çağrılarak arayüz baştan çizilir. Klasik masaüstü programcılığında olduğu gibi widget.setText tarzında bir metot tanımlaması Flutter'da bulunmaz. Eğer widget üzerindeki metin değişiyorsa bu widget'in build metodunun baştan çağrılarak yeniden ekranada çizilmesi ile yapılır.

Bildirimsel yaklaşım ile arayüz tasarlamanın birçok faydası vardır. Bunlardan en önemlisi arayüzün oluşmasında takip edilen tek bir kod akışının (widget ağacındaki build metotları) olmasıdır. Uygulama çalışması esnasında herhangi bir anda arayüzün nasıl görüneceğini sadece durumu değiştirerek görmeniz mümkündür. İlk bakışta bildirimsel yaklaşımı anlamak biraz zorlayıcı olabilmektedir. Bildirimsel yaklaşımı alıştıktan sonra aradaki fark ve kullanışlılığı daha iyi anlaşılacaktır.

9.3. Geçici Durum & Uygulama Durumu

Önceki bölümlerde StatefulWidget ile ekranın belli bir alanında durum yönetimi yaptık. Bu yaklaşım eğer durum takibi yapan widget kendi altındaki nesneleri ilgilendiren bir veri işlemesi yapıyorsa yerindedir ve biz bunu geçici (Ephemeral) durum değişikliği olarak adlandıracagız. Geçici denmesinin nedeni buradaki değişimin ilgili widget ağacının kullanımında olması durumunda takibinin gerekli olmasındandır. Ama bazı veriler ve bunlardaki değişim uygulamanın farklı noktalarında arayüze yansıtılmakta ve bu nedenle bunlardaki değişimi uygulama seviyesinde takip etmemiz gerekmektedir. Ne yazık ki bu iki durum takibi arasında kesin bir sınır bulunmamaktadır. Kodlama ihtiyacına göre yazılımcı hangisinin olması gerektigine kendi karar vermelidir.

Ama basitçe örneklemek gerekirse StatefulWidget ve setState ile geçici durum takibi;

- Bir PageView bileşeninde aktif sayfanın değişimi
- Bir animasyonda frame'in değişimi
- BottomNavigationBar'da aktif sekmenin değişimi

Gibi widget'in kendisini ilgilendiren arayüz güncellemlerinde kullanılmaktadır.

Uygulama seviyesi durum takibinin gerektiği zamanlara ise aşağıdakiler örnek verilebilir;

- Kullanıcı tercihlerinin değiştirilmesi
- Kullanıcı oturum açma işlemleri

- Bir sosyal ağ uygulamasındaki bildirim işlemleri
- Bir e-ticaret uygulamasındaki sepet işlemleri
- Bir haber uygulamasında makalelerin okunup okunmadı bilgisi takibi

Bu örnekleri vermemize rağmen tekrar belirtelim setState ile uygulama seviyesi durum takibinden hangisinin kullanımının doğru olduğuna dair kesin bir sınır bulunmamaktadır. Bunu ilgili verinin kullanım şekline göre siz kararlaştıracaksınız. Basit olarak şöyle düşünülebilir. Eğer işlenen veri sadece kendime ait widget ağacını ilgilendiriyorsa StatefulWidget ile geçici durum işlemi yapabilirim. Eğer işlenen veri birbirinden bağımsız ve birden fazla veya tüm uygulamadaki widgetların görsellliğini etkileyebilecekse o halde uygulama seviyesinde bir durum takibi yöntemi kullanmalıyım.

Uygulama seviyesinde durum takibi yapmak için birden çok yaklaşım ve üçüncü parti paketleri mevcuttur. Ama Flutter ekibi eğer daha önceden bu gibi bir durum takibi paketi kullanmadıysanız Flutter’ın kendi yaklaşımını tercih etmenizi önermektedir. Flutter’ın durum takibi yaklaşımı diğer paketlerdeki özelliklere göre basit olsa da her türlü ihtiyacı görmek için yeterli ve kolay uygulanabilirdir. Flutter’da uygulama seviyesi durumu takibi için provider kütüphanesi kullanılmaktadır.

9.4. Basit Uygulama Seviyesi Durum Yönetimi

Bu bölümde provider paketi kullanılarak Flutter’da nasıl durum yönetimini gerçekleştirebileceğimizi anlatacağız. Redux, Rx, hooks gibi üçüncü parti paketler daha gelişmiş özellikler sunuyor olsa da Flutter’da durum yönetimine ilk defa başlayacaklar için bu paketleri kullanmak gereksizdir. Projelerinizde ne kadar az üçüncü parti paketi olursa o kadar az bağımlılık oluşturursunuz ve bu sayede uygulama güncelleme ve bakım sürecinde kolaylık sağlanır. Bu prensibe Flutter’ın kendi durum takibi paketi provider kullanımını tercih etmeniz önerilmektedir.

Bu bölümde Flutter dokümantasyonundaki alışveriş sepeti örneği üzerinden sepet(card) verisinin uygulama içerisinde provider paketi ile nasıl taşındığı ve arayüz güncellemesinde kullanıldığı anlatılacaktır. Provider paketinde verinin oluşturulması takip edilmesi ve arayüz güncellemesinin tetiklenmesi için farklı alt bileşenler yer almaktadır. Bu adımları ve kullanılan bileşenleri sırasıyla ele alacağız. Provider paketini kullanmak için ilk adım olarak bunu projenize **pubspec.yaml** dosyası ile dahil etmeniz gerekmektedir. Pubspec.yaml dosyasında provider paketini aşağıdaki gibi **dependencies:** başlığı altına ekleyiniz.

```
name: my_name
description: Blah blah blah.

# ...

dependencies:
  flutter:
    sdk: flutter

  provider: ^6.0.0

dev_dependencies:
  # ...
```

Burada provider: dan sonra paketlerin sürüm yönetimi ile ilgili versiyon kullanım bilgisi gelmektedir. Bu konuya ileride ayrıntılı bir şekilde bakacağınız. Pubspec.yaml dosyasına yeni bir paket eklendiğinde proje klasöründe **flutter get** komutunun çalıştırılarak gerekli paketleri **pub.dev** sitesinden indirmeniz gerekmektedir. Visual Studio Code(vscode) ile proje geliştirmeyorsanız **pubspec.yaml** dosyasını kaydettiğiniz anda gereksinimler taranarak yeni eklenen paketler için **flutter get** komutu çalıştırılır. Bunu vscode ekranında sağ alta gelen bildirim ile görebilirsiniz.

Provider paketi projenize eklendikten sonra provider bileşenlerini kullanacağınız .dart kod dosyalarına;

```
import 'package:provider/provider.dart';
```

Satırını ekleyerek provider paketini kullanmaya başlayabilirsiniz.

Provider paketi içerisinde farklı değişim takiplerini yapacak özel tanımlamalar içermektedir. Ama biz şu an için sadece veri üzerindeki değişim ve bunun arayüze yansıtılması ile ilgileneceğiz. Bu nedenle Provider içerisindeki şu üç temel sınıfı dikkate almamız şimdilik yeterlidir.

- ChangeNotifier
- ChangeNotifierProvider
- Consumer

9.5. ChangeNotifier

ChangeNotifier kendi içerisindeki değişimi dinleyicilerine (**listeners**) aktaran Flutter SDK'sı içerisindeki basit bir sınıf tanımlamasıdır. Diğer bir değişle bir sınıf eğer **ChangeNotifier** olarak oluşturulmuşsa ondaki değişiklikleri takip edebilirsiniz. Bu yaklaşım Observable tasarım deseni kullananların aşina olduğu bir yaklaşımındır.

ChangeNotifier, provider paketi içerisinde verinizi tek yönlü olarak kapsüllersek durum takibini yapmaktadır. Eğer takip edilecek birden fazla veri varsa kod içerisinde birden çok **ChangeNotifier** tanımlı olacaktır.

Alışveriş uygulaması örneğinde alışveriş sepeti için bir veri modeli (CardModel) oluşturulmaktadır. Bu veri modelimizdeki değişiklikleri uygulama seviyesinde takip etmek istediğimiz için bu modelin **ChangeNotifier** sınıfından miras alarak oluşturulması gereklidir.

```
class CartModel extends ChangeNotifier {
    final List<Item> _items = [];
    UnmodifiableListView<Item> get items => UnmodifiableListView(_items);
    int get totalPrice => _items.length * 42;

    void add(Item item) {
        _items.add(item);
        notifyListeners();
    }

    void removeAll() {
        _items.clear();
        notifyListeners();
    }
}
```

Modelimizde alışveriş sepeti verilerini **_items** adlı iç değişkende saklayacağız. Bu liste üzerinde veri ekleme ve silme işlemleri **add** ve **removeAll** metotları ile gerçekleştirilecektir. Her iki metot da veri üzerinde değişiklik yapacağı, yani **durum(state)** değiştireceği için bunu arayüz güncellemesi yapmak isteyen tüm dinleyicilere bildirmemiz gerekmektedir. Bu bildirimi **ChangeNotifier**'ın sunduğu **notifyListeners()** metodunu çağırarak yapmaktayız.

9.6. ChangeNotifierProvider

ChangeNotifierProvider, provider paketi içerisinde gelen ve **ChangeNotifier** ile takip edilen veri modelini altındaki diğer widgetlara ulaştıran bir widgettir. Tahmin edeceğiniz üzere widget ağacında **ChangeNotifierProvider** verinin güncellenmesi sonucu ekranda değişmesi gereken bileşenlerin hepsini kapsayacak seviyede olması gereklidir. Çoğu örneğimizde kök seviyede olduğunu fark edeceksiniz. Alışveriş sepeti uygulamasında iki ekran **MyCatalog** ve **MyCard** widgetlarını sarmalayacak şekilde yerleştirilmektedir. Widget ağacında **ChangeNotifierProvider**'ın yerini seçerken yine de mümkün olan en alt seviyede bulunması sistemin karmaşasını azaltmak adına yerinde bir tercih olacaktır (Özellikle birden fazla provider olan bir yapımız oluşacaksa). Örneğimizde **MyCard** ve **MyCatalog** widgetları **MyApp** widgeti altında olduğu için **ChangeNotifierProvider**, **MyApp** widgeti üzerinde konuşlandırılır.

```
void main() {
  runApp(
    ChangeNotifierProvider(
      create: (context) => CartModel(),
      child: const MyApp(),
    ),
  );
}
```

ChangeNotifierProvider widgeti neredeyse tüm Flutter widgetlarının yapılandıracı fonksiyonunda bulunan **child** parametresi ile kendi alt widget ağacını belirtmektedir. Burada yeni parametremiz **create** parametresidir. Bu parametre **ChangeNotifierProvider**'ın takip edeceği veri modelini(**ChangeNotifier**'dan miras almış olan) oluşturan bir fonksiyon almaktadır. Bu fonksiyonun parametresi ise **BuildContext**'tir.

```
create: (context) => CartModel(),
```

Yukarıdaki satırda **CartModel** yapılandıracı fonksiyonu çağrılarak bir adet **CartMoel** objesi değişiklik takiplerine konu olacak şekilde bellekte oluşturulur. Create parametresinde bir builder fonksiyonu yaklaşımı kullanıldığına dikkat ediniz. **ChangeNotifierProvider**, **CartModel**'in gerekmedikçe birden fazla kopyasını bellekte oluşturmayacak şekilde akıllı bir yaşam döngüsüne sahiptir. Ayrıca işlem bittiğinde oluşturulan **CartModel** dispose edilir.

Provider'ın veri değişiminden başka işlemleri de takip edebilecek bir paket olduğunu söylemiştık. Eğer uygulamanız içerisinde birden fazla provider yaklaşımı kullanacaksanız bunların eş zamanlı olarak uygulanması için **MultiProvider** widget'ı kullanılabilir. Bu her bir provider yapısını bir listede alacak şekilde **providers** adında bir parametre barındırır.

```
void main() {
  runApp(
    MultiProvider(
      providers: [
        ChangeNotifierProvider(create: (context) => CartModel()),
        Provider(create: (context) => SomeOtherClass()),
      ],
      child: const MyApp(),
    );
}
```

MultiProvider kullanımında her bir provider için **child** parametresi verilmediğini bunun yerine widget ağacının **MultiProvider**'ın kendi child parametresi üzerinden oluşturulduğuna dikkat ediniz. Her provider yapısı kendi takip edeceği modeli **create:** parametresi ile oluşturmaktadır.

9.7. Consumer

Uygulamanın widget ağacında üst bir seviyede **ChangeNotifierProvider** ile **CartModeldeki** değişim takip edildiğine göre artık bu değişimin ekranı yansıtılması için gerekli kod bloklarını widget ağacımız üzerine yerlestirebiliriz. Bu işlem için **Consumer** widget’ı kullanılacaktır.

```
return Consumer<CartModel>(
  builder: (context, cart, child) {
    return Text("Total price: ${cart.totalPrice}");
  },
);
```

Consumer widgetini tanımlarken değişikliği izlemek istediğimiz veri modelini belirtmemiz gereklidir. Bu nedenle örneğimizdeki **Consumer<CartModel>** jenerik formda oluşturulur. Consumer eğer işleyeceği veri türünü jenerik olarak belirtmezse provider kütüphanesi nasıl çalışması gerektiğini bilemez. Provider kütüphanesi veri türleri üzerinden takip yapmaktadır.

Consumer widgetinin yapılandırıcı fonksiyonun tek zorunlu parametresi **builder’dır**. Bu parametre üç parametresi olan bir fonksiyon almaktadır. İlgili fonksiyonun parametreleri sırası ile BuildContext(context), işlem gören veri modeline referans (cart CartModel türünde) ve Widget ağacında yeniden çizilmesi gerekmeyen alt ağaç kısmı (child).

Consumer içerisinde cart üzerindeki veriyi doğrudan alan widgetlar bulunması ve bunların oluşturduğu widget ağacının return ile döndürülmesi builder fonksiyonun beklenen davranışıdır. Burada bir Text widgeti içerisinde gösterilecek metin cart.totalPrice değerini kullanmaktadır. Bu örnekte child parametresi tanımlanmamış (null) ve widget ağaç içeresine yerleştirilmemiştir.

Aşağıdaki örnekte **Consumer’ın builder** parametresine verilecek fonksiyon için kullanılacak **child** parametresinin tanımlanması ve **builder** içerisinde kullanımı örneklenmiştir. Burada **Consumer** altında çizilecek widget ağaçının **builder** parametresinden gelecektir. Bunun içerisinde ilk çizimden sonra yenilenmesi gerekmeyen kısımları ifade etmek için Consumer, **child** adında başka bir parametre tanımlamaktadır. Bu parametreye verilen widget ağaçının **builder’a** verilen fonksiyona beslenir ve böylece widget ağacının sadece gerektiği kadar kısmının yeniden çizimi **Consumer** tarafından tetiklenmiş olur.

```
return Consumer<CartModel>(
  builder: (context, cart, child) => Stack(
    children: [
      if (child != null) child,
      Text("Total price: ${cart.totalPrice}"),
    ],
  ),
  child: const SomeExpensiveWidget(),
);
```

Consumer widgetinin widget ağacında yerleşiminde performans gereksinimi olarak olabildiğince değişimin olacağı alana yakın yani aşağılarda olması istenir.

9.8. Provider.of

Bazen arayüzün güncellenmesini gerektirmeyecek bir noktada provider ile tanımlanmış veri modeline erişme ihtiyacı duyuyoruz. Örneği alışveriş sepeti uygulamasında sepeti boşaltmak için bir ekran ve bir buton tasarlanmış olsun. Butona tıkladığınızda alışveriş sepetini boşaltırsın ama bu işlemden ekranda bir güncelleme gerekmemektedir. Burada veri modeline erişmek için Consumer kullanmak ekranda hiçbir güncelleme yapmayacağımız ve Consumer gereksiz yere rebuild işlevini çağıracağı için performans kaybına neden olur.

Bu aşamada sepeti temizlemek için kullanacağımız botunun kodlamasında

```
Provider.of<CartModel>(context, listen: false).removeAll();
```

Listen parametresi false olarak ayarlanmış şekilde Provider.of metodu ile veri kaynağına erişilebilir. Listen:false seçeneği bu değişimin bir takibinin yapılmayacağını ifade etmektedir.

Burada anlatılan örneğin tam kodunu;

https://github.com/flutter/samples/tree/master/provider_shopper

adresinden inceleyebilirsiniz. Eğer daha basit bir örnek üzerinden provider paketinin kullanımını incelemek isterseniz;

https://github.com/flutter/samples/tree/master/provider_counter

örneği ile deneyim kazanabilirsiniz.

9.9. Diğer Durum Yönetim Yaklaşımları

Durum yönetimi en karmaşık konulardan biridir. Buradaki içerikte çok yalnız şekilde Flutter'daki Provider paketinin uygulama seviyesi durum yönetimi için nasıl kullanılacağı verilmiştir. Eğer buradaki içerik ihtiyacınıza uygun değilse veya daha farklı özellikler sunan bir yaklaşım arayışınız varsa farklı durum yönetim modellerine referansların bulunduğu aşağıdaki Flutter dokümantasyon sayfasını ziyaret edebilirsiniz.

<https://flutter.dev/docs/development/data-and-backend/state-mgmt/options>

Bölüm Özeti

Flutter'da arayüzün güncellenmesi bildirimsel bir formda yapılmaktadır. Bunun için uygulama içerisindeki durum değişimi takip edilerek gerektiğinde arayüzün çizimi için widget ağacı belli bir noktada baştan çizilir. Flutter bunun için provider paketi içerisinde **ChangeNotifier**, **ChangeNotifierProvider** ve **Consumer** sınıfları ile basit bir yaklaşım sunmaktadır. İstenilirse üçüncü parti paketlerle de durum yönetimi yapılabilir.

Kaynakça

<https://docs.flutter.dev/development/data-and-backend/state-mgmt/declarative>

<https://docs.flutter.dev/development/data-and-backend/state-mgmt/ephemeral-vs-app>

<https://docs.flutter.dev/development/data-and-backend/state-mgmt/simple>

https://github.com/flutter/samples/tree/master/provider_shopper

https://github.com/flutter/samples/tree/master/provider_counter

<https://flutter.dev/docs/development/data-and-backend/state-mgmt/options>

10. KALICI VERİ SAKLAMA

Giriş

Dosya sistemleri işletim sistemleri arasında farklılık göstermektedir. Şu an aktif iki mobil platform da (Android-iOS) Unix dosya sistemini kullanıyor olsada klasörleme yapıları ve API'ları farklıdır. Formatlı ve karmaşık verilerin saklanması veritabanı gibi uzman bir veri saklama yapısını kullanmayı gerektirir. Mobil cihazlarda kullanılan yegane veritabanı sistemi SQLite veritabanıdır. Bu bölümde kalıcı veri saklama ile ilgili mobil platformları ilgilendiren konular temel olarak ele alınmıştır.

10.1. Dosya İşlemleri

Flutter uygulamalarında tüm uygulamalarda olduğu gibi dosya sisteme erişim ihtiyacınız olabilir. Örneğin uygulamanın yüklenmesinde var olmayan ama İnternet üzerinden indirilerek cihazda bir dosya saklanmak istediğinizde dosya işlemleri yapmamız gereklidir.

Flutter'da dosya sistemleri işlemleri için **path_provider** plugin'ı ve **dart:io** kütüphaneleri kullanılmaktadır.

Akıllı telefonlarda dosya sistemi işlemleri yapacağımız zaman en dikkat edilmesi gereken dosyanın yazılacağı veya okunacağı yeri seçmek olacaktır. Dosya sistemleri ve klasör yapısı her işletim sisteminde farklılık gösterebilir. Benzer şekilde farklı telefon üreticilerinin kullandıkları Android sürümleri arasında da farklılıklar olabilir. Bu noktada ürettiğiniz uygulamanın her platformda düzgün çalışabilmesi için dosya sisteme erişimde doğrudan yol adı vermek yerine bazı yardımcı fonksiyonları kullanmamız gerekmektedir. **path_provider** plugin'ı bize bu açıdan yardımcı olmaktadır.

Akıllı telefonların dosya sistemini görme şekilleri de farklı olabilir. Android kalıcı hafızayı (persistent memory) iki bölümde ele almaktadır. Bunlar uygulamalara ait özel bölümü ifade eden **Internal Memory** ve son kullanıcı dosyalarının kaydedildiği **External Memory** şeklinde ifade edilmektedir. Buradaki Internal ve External kavramları mantıksal kavumlardır. Hafızanın cihaza yerleştirilmesinde bu değerler belirlenerek hafıza ikiye bölünmektedir. External Memory, SDcard'ı ifade etmemektedir. SDcard takıldığından bu da External Memory'nin bir parçası olarak görülmektedir. Android bu ayrimı son kullanıcı dosyaları (resim,müzik,belge,video,melodi,...) ile uygulama özelindeki dosyaları birbirinden ayırmak için kullanmaktadır. External Memory'de kaydedilen veriler bütün uygulamalar tarafından ve son kullanıcı tarafından görülebilir, okunup-yazılabilir. Internal Memory'deki dosyalar ise uygulamaya özgüdür. Her uygulama sadece kendi Internal Memory alanına erişim sağlayabilir. Bu yaklaşım sayesinde bir uygulamanın diğer bir uygulama dosyasına müdahale ederek bozması ve güvenlik zafiyeti oluşturulması engellenmiştir. Bu yaklaşımı SandBox (Kum Havuzu) yaklaşımı denmektedir. **Path_provider** plugin'ı kullanılarak Internal ve External Memory alanlarında ayrı ayrı özel klasörlere erişim sağlayabilmekteyiz.

Dosya sistemi işlemleri için yapacaklarını 4 başlıkta inceleyeceğiz:

1. Doğru yerel dosya yolunu bulma işlemi
2. Bu dosya yoluna bir referans oluşturma
3. Dosya yazma işlemi
4. Dosya okuma işlemi

10.1.1. Doğru Yerel Dosya Yolunu Bulma İşlemi

Dosyaların okunması veya yazılması için uygun klasörün tespit edilmesi için **path_provider** plugin'ı kullanılacaktır. Akıllı cihazların dosya sistemlerinde özelleştirilmiş klasörler vardır. Örneği geçici (**Temporry**) dosyalar ile kalıcı dosyalar farklı klasörler altında saklanmaktadır.

Geçici dosyaların bulunduğu klasör bilgisi iOS'da **NSCachesDirectory** ifadesiyle, Android'de ise **getCacheDir()** fonksiyonu ile elde edilmektedir.

Dokümanların kaydedildiği uygulamaya özgü klasöre iOS'da **NSDocumentDirectory** ifadesiyle, Android'de ise **AppData** ifadesiyle elde edilmektedir.

Uygulamanın dokümanlarının tutulduğu klasörüne **path_provider** aracılığı ile erişmek için **getApplicationDocumentsDirectory()** fonksiyonu aşağıdaki örnekteki gibi kullanılabilir.

```
Future<String> get _localPath async {
  final directory = await getApplicationDocumentsDirectory();

  return directory.path;
}
```

10.1.2. Bu Dosya Yoluna Bir Referans Oluşturma

İlgili dosya yoluna bir referans oluşturmak için **dart:io** kütüphanesi içerisinde **File** sınıfı kullanılır.

```
Future<File> get _localFile async {
  final path = await _localPath;
  return File('$path/counter.txt');
}
```

10.1.3. Dosya Yazma İşlemi

File sınıfı üzerinden dosyaya oluşturularak yazma işlemi yapılır. Aşağıdaki örnekte **counter** değişkeni dosyaya yazılmaktadır.

```
Future<File> writeCounter(int counter) async {
  final file = await _localFile;

  // Write the file
  return file.writeAsString('$counter');
}
```

10.1.4. Dosya Okuma İşlemi

File sınıfı üzerinden dosyaya erişerek okuma işlemi yapılır.

```
Future<int> readCounter() async {
  try {
    final file = await _localFile;

    // Read the file
    final contents = await file.readAsString();

    return int.parse(contents);
  } catch (e) {
    // If encountering an error, return 0
    return 0;
  }
}
```

Verilen örneklerin tam bir uygulama olarak yazıldığı haline aşağıdaki bağlantıdan erişebilirsiniz.

<https://flutter.dev/docs/cookbook/persistence/reading-writing-files>

10.2. Shared Preferences ile Key-Value Çiftleri Saklama

Dosya sistemine veri saklama ihtiyacımız bellekte kullandığımız Map veri tipi gibi sadece key-value çiftleri şeklinde ise kalıcı veri saklama işlemini kolaylaştırına **shared_preferences** plugin'ı kullanabilir. Bu plugin

iOS'da **NSUserDefaults**, Android'de **SharedPreferences** olarak ifade edilen key-value çiftlerini dosya sistemine kaydedip okuma yaklaşımını Flutter'da sağlamaktadır.

Kullanımı için yapılacak **pubspec.yaml** dosyasına plugin gereksinimi eklenmelidir.

```
dependencies:  
  flutter:  
    sdk: flutter  
  shared_preferences: "<son sürümü>"
```

Burada pub.dev sayfasından paketin mümkün olan son sürümünü kullanmaya özen gösteriniz.

SharedPreferences üzerinden veri yazmak için ilgili veri türüne göre `setInt`, `setBool`, `setString`, ... fonksiyonları kullanılır. Eğer SharedPreferences dosyasına özel bir isim verilmeden erişiliyorsa her uygulamanın kendine has bir adet varsayılan SharedPreferences dosyası mevcuttur. Bu dosya üzerinde okunma yazma sağlanır. Buradaki Shared ifadesi kafanızı karıştırmamalıdır. SharedPreferences dosyası uygulamaya özgü olarak SandBox ortamında saklanır. Diğer uygulamaların erişimi yoktur. Onlar ile paylaşılmaz. Buradaki paylaşım uygulama içi bileşenler arası manasındadır. `getInstance()` metodu ile uygulamaya ait varsayılan SharedPreferences dosyasının referansı elde edilir.

```
// obtain shared preferences  
final prefs = await SharedPreferences.getInstance();  
  
// set value  
prefs.setInt('counter', counter);
```

Yazma işlemine benzer şekilde SharedPreferences dosyasından veri okumak için `getInt`, `getBool`, `getString`, ... formunda fonksiyonlar kullanılır.

```
final prefs = await SharedPreferences.getInstance();  
  
// Try reading data from the counter key. If it doesn't exist, return 0.  
final counter = prefs.getInt('counter') ?? 0;
```

SharedPreference dosyasında eklenen bir kayıt siz silmediğiniz müddetçe sürekli saklanacaktır. Kaydı silmek için key değeri ile `remove()` fonksiyonu çağrılmalıdır.

```
final prefs = await SharedPreferences.getInstance();  
  
prefs.remove('counter');
```

SharedPreferences dosya yapısı sadece belli veri türleri ile çalışmaktadır. Bunlar; **int**, **double**, **bool**, **String** ve **StringList** veri türleri. Bu dosya türünün büyük boyutta veriler için tasarlanmadığını da unutmamak gereklidir. Eğer çok miktarda veri saklanacak ise SQLite veritabanı kullanımını tercih edilmelidir.

SharedPreferences kullanımı ile ilgili daha detay bilgi için Android işletim sisteminin dokümantasyonunu inceleyiniz.

<https://developer.android.com/guide/topics/data/data-storage#pref>

10.3. SQLite ile Veritabanı İşlemleri

Her uygulamanın veri saklama ihtiyacı bulunmaktadır. Eğer verileriniz sürekli sorgulanacak formatlı bir yapıdaysa, oransal olarak büyük ve birbiriyile ilişkili ise bizim için en iyi alternatif veritabanı kullanımızdır. Akıllı cihazlarda veritabanı olarak SQLite veritabanı desteklenmektedir.

SQLite, sürekli arka planda çalışan bir servise ihtiyaç duymaksızın çalışan ilişkisel bir veritabanıdır. MSSQL, MySQL, Oracle gibi klasik veritabanı yönetim sistemlerinden bazı farklılıklar olsa da onların tekli kurulum için verdikleri özelliklerin çoğunu desteklemektedir.

SQLite hakkında şu bilgileri özetlemekte fayda vardır:

- Veriler tablolarda **row**(satır) ve **column**(sütun) formunda bir Excel tablosu gibi saklanır.
- Her satır ve sütun kesişimine **Field** denir ve Field'lar aynı sütun içerisinde aynı veri türünü saklamak zorunda değildir.
- Field'larda veri, diğer field'lara referans veya başka tablolara referanslar olabilir.
- Her satır tekil bir ID değeri ile ifade edilirler.
- Bir tablo içerisindeki sütun adları tekil olmalıdır.
- SQLite **self-contained** (kendi başına çalışabilir) formdadır. Çalışması için sadece kütüphanesinin çağrılması yeterlidir.
- SQLite **serverless** (sunucusuz) çalışır. Yani arka planda işletim sistemi içerisinde sürekli çalışacak bir servise ihtiyaç duymaz.
- **Zero-configuration** (Ayar yapmadan) çalışma özelliği vardır. Bu açıdan uygulama geliştirme için çok elverişlidir.
- **Transactional** özellikleştir. Birden fazla işlemi atomik olarak çalıştırma imkanına sahiptir.

Flutter'da SQLite veritabanı işlemleri için **sqflite** plugin'i kullanılmaktadır. Bu plugin pub.dev sayfası altından incelenebilir.

<https://pub.dev/packages/sqflite>

sqflite plugin'i kullanılarak SQLite üzerinde Select, Insert, Create, Update, Delete gibi SQL işlemleri gerçekleştirilmektedir.

SQLite kullanımı hakkında detaylı bilgiye SQLite'in Tutorial İnternet sayfasındaki içerikten bakabilirsiniz.

<http://www.sqlitetutorial.net/>

Şimdi adım adım temel veritabanı işlemlerini sqflite plugin'i kullanarak gerçekleştireceğiz. Öncelikle gerekli paketleri **pubspec.yaml** dosyasına **dependencies**: altına ekleyerek başlıyoruz. **sqflite** yanında veritabanı yolunu girmek için **path** paketini de kullanacağız.

```
dependencies:  
  flutter:  
    sdk: flutter  
  sqflite:  
  path:
```

Sonrasında veritabanı işlemleri yapacağımız dosyaların başında bu paketlere erişmek için import kısımları eklenir. Veritabanı işlemleri I/O işlemleri olduğu için asenkron olarak yapılmalıdır gereklidir.

```
import 'dart:async';  
  
import 'package:path/path.dart';  
import 'package:sqflite/sqflite.dart';
```

Veritabanına kaydedilecek veriler için kod içerisinde bir veri modeli oluşturmamız gerekmektedir. Bu örnekte veri modeli olarak “**Dog**” sınıfı kullanılacaktır ve bu sınıfın **id**, **name** ve **age** adında üç tane üyesi olacaktır. Bunlar SQLite veritabanında tablonun sütunlarına eşlenecektir. **id** veritabanı için tekil ID alanı olarak kullanılacaktır.

```
class Dog {  
    final int id;  
    final String name;  
    final int age;  
  
    Dog({  
        required this.id,  
        required this.name,  
        required this.age,  
    });  
}
```

Veritabanı ile bağlantı sağlamak için önce veritabanının Internal Memory içerisinde veritabanları için işletim sistemlerinin varsayılan klasörünün yolu **getDatabasesPath()** ile edinilir. Sonra veritabanı açma-oluşturma fonksiyonu (**openDatabase()**) kullanılarak bu yol üzerinde bir veritabanı oluşturulur veya var olan veritabanına bağlanılır.

Veritabanı işlemleri daima asenkron yapılmak durumundadır. Bu nedenle aşağıda verilecek örnekte veritabanı bağlantısı doğrudan main fonksiyonu içerisinde çağrıldığı için main fonksiyonu **async** anahtar kelimesi ile asenkron çalışır hale getirilmiştir.

```
// Flutter upgrade işlevinin oluşturacağı hataları bertaraf eder.  
// 'package:flutter/widgets.dart'ın import edilmesini gerektirir.  
WidgetsFlutterBinding.ensureInitialized();  
// veritabanı açma ve referansını alma işlemi  
final database = openDatabase(  
    join(await getDatabasesPath(), 'doggie_database.db'),  
);
```

Yukarıdaki kod uygulamanın kendine özgü veritabanı klasöründe doggie_database.db adındaki veritabanı dosyasını oluşturur veya varsa açar. Burada kullanılan **join** işlevi veritabanı adı ile yolu bilgisini ilgili dosya sisteme uygun şekilde path plugin’ı vasıtasyyla düzgün birleştirmeye yarar.

Veritabanına bağlantı kurduğumuza göre bir sonraki aşama olarak veritabanında tablo oluşturma sürecini ele alalım. Veritabanı yokken ilk bağlantı kurulduğunda **Dog** yapısı için bir tablo oluşturmak gerekecektir. Bunun için **openDatabase** fonksiyonunun **onCreate** parametresine veritabanını hazır hale getirme fonksiyonu verilir. Bu fonksiyon iki parametre almaktadır. Bunlardan ilki veritabanını ifade eden bir referans ikincisi ise veritabanının sürümünü ifade eden bir değerdir.

Oluşturulacak tabloda Dart dilindeki veri türlerinin SQLite’daki karşılıkları eşleşmelidir. **int** için **INTEGER**, **String** için **TEXT** veri türleri SQLite’daki karşılıklardır. SQLite’da veri eşleştirmesi yaparken Dart için tanımlı her veri türünün SQLite’da bir karşılığı olmadığını unutmamanız gereklidir. Örneğin **DateTime** için SQLite’da bir veri türü karşılığı yoktur.

SQLite’ın sunduğu veri türlerine SQLite’ın kendi belgelendirme sayfasından bakabilirsiniz.

<https://www.sqlite.org/datatype3.html>

```

final database = openDatabase(
    join(await getDatabasesPath(), 'doggie_database.db'),
    // Veritabanı ilk oluşturulduğunda Dogs tablosunu oluşturur.
    onCreate: (db, version) {
        // veritabanı üzerinde CREATE TABLE sqlini çalıştırır.
        return db.execute(
            'CREATE TABLE dogs(id INTEGER PRIMARY KEY, name TEXT, age INTEGER)',
        );
    },
    // versiyon bilgisi onCreate'te kullanılır ve sürüm yükseltme düşürme kararı
    // versiyon bilgisine göre verilir.
    version: 1,
);

```

Dogs tablosuna yeni bir kayıt eklemek için `insert()` metodu aşağıdaki gibi çağrılır. Burada Dog sınıfının `toMap` fonksiyonu kullanılarak bir Dog objesi içerisindeki alanlar, key-value çiftleri haline getirilir. `insert()` metodundaki **conflictAlgorithm** parametresi eğer veritabanındaki iki kayıt tekil id (PrimaryKey) üzerinden çakışırsa nasıl davranışacağını belirtmektedir. `replace` değeri önceki kaydın üzerine yazılmasını söylemektedir.

```

class Dog {
    final int id;
    final String name;
    final int age;

    Dog({
        required this.id,
        required this.name,
        required this.age,
    });

    Map<String, dynamic> toMap() {
        return {
            'id': id,
            'name': name,
            'age': age,
        };
    }

    @override
    String toString() {
        return 'Dog{id: $id, name: $name, age: $age}';
    }
}

```

```

Future<void> insertDog(Dog dog) async {
    final db = await database;

    await db.insert(
        'dogs',
        dog.toMap(),
        conflictAlgorithm: ConflictAlgorithm.replace,
    );
}

```

```
// Bir Dog objesi oluşturup veritabanına kaydediyoruz.  
var fido = Dog(  
    id: 0,  
    name: 'Fido',  
    age: 35,  
);  
  
await insertDog(fido);
```

Veritabanından Select cümlecipleri ile veri okumak istiyorsak sqflite'in bize sunduğu iki sorgulama alternatifii vardır.

query: Eğer sonuçlar tek bir tablodan gelecekse, tablo sorgulaması query metodu ile parçalı şekilde (select, where , groupby, orderby ayrı ayrı alınacak şekilde) gerçekleştirilebilir. Sonucu bize bir **List<Map>** formunda dönmektedir.

rawquery: Eğer klasik SQL cümleciplerinin veritabanı üzerinde çalıştırılması isteniyorsa veya birden fazla tablonun birleştiği bir sorgu yazılacaksa **rawquery** metodu kullanılır.

Örneğimizde Dogs tablosundan query ile bir sorgulama yapılacak ve dönen List<Map> sonucu List<Dog> formuna çevrilecektir.

```
// dogs tablosundaki tüm kayıtları okur.  
Future<List<Dog>> dogs() async {  
    final db = await database;  
  
    // Veritabanını sorgulayan kısım  
    final List<Map<String, dynamic>> maps = await db.query('dogs');  
  
    // List<Map<String, dynamic> yapımı List<Dog>'a çeviriyor.  
    return List.generate(maps.length, (i) {  
        return Dog(  
            id: maps[i]['id'],  
            name: maps[i]['name'],  
            age: maps[i]['age'],  
        );  
    });  
}
```

```
// Yukarıda yazılan fonksiyon tüm dog kayıtlarını okumak için çağrılmıyor  
print(await dogs());
```

update() metodu ile veritabanındaki bir kayıt güncellenir.

```

Future<void> updateDog(Dog dog) async {
  final db = await database;

  // Belirtilen Dog kaydını günceller.
  await db.update(
    'dogs',
    dog.toMap(),
    // Dog'ın kaydı olduğu tekil id ile eşleştirilerek kontrol edilir.
    where: 'id = ?',
    // Where kısımındaki argümanları parametrik olarak SQL injection engellenir.
    whereArgs: [dog.id],
  );
}

```

```

// Fido's yaşı güncellenir ve db ye kaydedilir.
fido = Dog(
  id: fido.id,
  name: fido.name,
  age: fido.age + 7,
);
await updateDog(fido);

// Güncellenmiş sonucu yazar.
print(await dogs()); // Prints Fido with age 42.

```

Benzer şekilde verileri silmek için **delete()** metodunu kullanabilirsiniz. Veri silme işleminde tablodaki tekil id alanını **where** için parametrik olarak göndermeyi unutmamak gereklidir.

```

Future<void> deleteDog(int id) async {
  final db = await database;

  // Veritabanından bir Dog kaydı silinir.
  await db.delete(
    'dogs',
    where: 'id = ?',
    whereArgs: [id],
  );
}

```

Anlatılan tüm adımların uygulandığı örneği aşağıdaki sayfadan inceleyebilirsiniz.

<https://flutter.dev/docs/cookbook/persistence/sqlite#example>

sqflite plugin'ın sayfasında sqflite kullanımı ile ilgili ek örnekler verilmektedir.

<https://pub.dev/packages/sqflite>

Konunun yalnızca tutulması adına tüm örnekler burada sunulmamıştır. SQLite üzerinde veritabanı işlemlerini sqflite ile doğrudan yapabilir veya bir sonraki başlıklı gibi bir ORM yaklaşımı (Floor) kullanabilirsiniz. ORM yaklaşımı tercih etmek uygulama geliştirme ve hata bulma performansınızı artıracaktır.

10.4. Floor Kütüphanesi ile ORM Yaklaşımı

Veritabanı sorgulama işlemlerinde en büyük sıkıntısı bellekteki obje görünümü ile veritabanındaki kayıt görünümünün eşleşmesi için bir dönüşüm yapılması gereklidir. Eğer bunu programcı elle yaparsa

veritabanındaki bir şema değişimi kod tarafının elle güncellenmesini gerektirir. Bu işlem doğru takip edilmediğinde çalışma zamanı hataları oluşabilir. SQL sorguları String olarak yazıldığı için kod ile veritabanı şeması eşleşmeleri tam olarak güncellenemeyebilir.

Bu problemin çözümü için veritabanı bağlantı katmanları için **Nesne-İlişkisel Eşleşme (Object-Relational Mapping ORM)** yöntemi kullanılmaktadır. ORM sayesinde veritabanı işlemleri için kod içeresine SQL parçacıkları gömülümesi azaltılır ve programcı veritabanı işlemlerini bellekteki bir nesne ve nesne özelliklerine erişir şekilde kullanmış olur. .NET ve Java programlama ortamları için **EntityFramework** ve **Hibernate** en bilindik ORM yaklaşımlarıdır. Android işletim sistemi için **RoomPersistenceLibrary** adında bir ORM yaklaşımı kullanılmaktadır.

Room Persistence Library için açıklama sayfasına aşağıdan erişebilirsiniz.
<https://developer.android.com/topic/libraries/architecture/room>

Bu bölümde ele alacağımız **Floor** kütüphanesi Android'deki Room Persistence Library kılavuz alınarak hazırlanmış bir kütüphanedir. Room'un sağladığı aynı işlevleri Flutter çerçevesi içerisinde kullanıma izin vermektedir.

Floor kütüphanesinin açıklamasına pub.dev sayfası altından erişilebilir. <https://pub.dev/packages/floor>

Ayrıca Floor kütüphanesinin kendi internet adresinden de eğitim içeriğine erişilebilir. Bu sitede daha kapsamlı olarak kütüphane ile SQLite üzerinde yapılabilen tüm işlevler örnekler üzerinden verilmektedir. Biz burada sadece başlangıç seviyesinde kütüphane kullanımını tanıtacağız.

<https://floor.codes/getting-started/>

Floor veritabanı işlemleri için üç temel bileşen kullanmaktadır. Bunlar tabloları ifade eden **Entity**, Veritabanı işlemlerini organize eden **Data Access Object (DAO)** ve tüm mimariyi bir arada tutan ve süreçleri yöneten **Database** sınıflarıdır. Şimdi Floor kütüphanesini kullanarak bir SQLite veritabanına bağlanmak için gerekli adımları kısaca anlamaya çalışalım.

10.4.1. Gereksinimlerin Ayarlanması

Her ek paket kullanımında olduğu gibi Floor kullanılması için de **pubspec.yaml** dosyasına gerekli bağımlılıklar ifade edilecektir. Floor için iki farklı paket ekleyeceğiz. İlk **floor** ikincisi de **floor_generator**'dur. Floor bir ORM yaklaşımı olduğu aslında SQLite'a bağlantı kuracak gerçek kodları bizim adımıza üreticek ve projeye ekleyecektir. **floor_generator** paketi bu işlemleri yerine getirmek için gerekmektedir. Bir başka geliştirme aşaması gereksinimi de **build_runner** paketidir. Bunların **pubspec.yaml** dosyasında tanımlanması aşağıdaki gibi olmalıdır (sürüm numaraları pub.dev sitesinden kontrol edilerek güncellenmelidir). Bu paketlerin yanında başka paketler de kullanılıyorsa bunlar pubspec.yaml dosyasında yer almalıdır. Örneğin dosya yoluna erişim için kullandığımız **path** gibi. Burada sadelik açısından sadece konu ile ilgili başlıklar pubspec.yaml dosyasında gösterilmektedir.

```
dependencies:  
  flutter:  
    sdk: flutter  
  floor: ^1.2.0  
  
dev_dependencies:  
  floor_generator: ^1.2.0  
  build_runner: ^2.1.2
```

10.4.2. Bir Veri Modeli (Entity) Oluşturulması

ORM yaklaşımlarında kullanılan veri modellerine **Entity** denir. Entity veritabanındaki tabloların nasıl olacağını ifade ederler ve bir satırda bulunacak alanların Entity içerisinde tanımlamaları yapılır. Entity POCO denilen basit sınıf tanımlamalarıdır. Floor kütüphanesinin bu sınıf tanımlaması ile veritabanında

oluşacak birimleri eşleştirmesi için annotation dipnot- @ simbolü ile başlayan ek bilgi satırları) kullanılır. Bir sınıfın Entity olduğunu ifade etmek için **@entity** annotation’ı kullanılmaktadır.

```
// entity/person.dart
import 'package:floor/floor.dart';

@Entity
class Person {
    @primaryKey
    final int id;
    @ColumnInfo(name: 'custom_name')
    final String name;

    Person(this.id, this.name);
}
```

Yukarıdaki örnekte Person sınıfı **@entity** annotation’ı ile belirtilerek SQLite’da bir Person tablosu oluşması gerektiğini bildirmektedir. Person sınıfı içerisindeki üye değişkenler de bu tablo içerisindeki sütunları (Column) ifade ederler. Sütunların SQLite tarafından veri tipi dışında olması gereken özelliklerini vermek için yine annotationlar ile işaretlenmeleri gereklidir. Örneğimizde **id** değişkeni **@primaryKey** annotation’ı ile işaretlenerek tablodaki birincil anahtarın **id** sütunu olacağı belirtilmiştir. Yine bir sütun ile ek bilgi verilmesi gerekiyor ise (örneğin sütun için kod tarafında kullanılan değişken adı ile tabloda kullanılacak sütun adı farklı olması gerekiyorsa) **@ColumnInfo** annotation’ı **name:** parametresi ile kullanılarak bu sağlanır.

10.4.3. Bir Veri Erişim Nesnesi – Data Access Object -DAO Oluşturulması

Bu bileşen SQLite veritabanına erişimi yönetmektedir. Bu soyut (**abstract**) sınıf veritabanına erişimde kullanılacak fonksiyonların prototiplerini içerecektir. İlgili fonksiyonların gövdesi Floor tarafından oluşturulacaktır. Bu fonksiyonların dönüş değerleri **Future** veya **Stream** nesneleri olacaktır. Çünkü veritabanı işlemleri asenkron gerçekleştirilecektir. DAO için kullanılacak soyut sınıf **@dao** annotation’ı ile belirtilmektedir.

Veritabanı sorguları **@query** annotation’ı ile ifade edilir. **@insert**, **@update**, **@delete** annotationları da ilgili işlemler için kullanılmak istenilen soyut fonksiyon tanımlamalarının başına eklenmelidir. Örneğimizde iki farklı “Select” ifadesinin ve bir “Insert” işleminin gerçekleştirilmesi için gerekli olan fonksiyon tanımlamaları ve gerekli annotationlar verilmektedir.

```
// dao/person_dao.dart
import 'package:floor/floor.dart';

@dao
abstract class PersonDao {
    @Query('SELECT * FROM Person')
    Future<List<Person>> findAllPersons();

    @Query('SELECT * FROM Person WHERE id = :id')
    Stream<Person?> findPersonById(int id);

    @insert
    Future<void> insertPerson(Person person);
}
```

10.4.4. Bir Veritabanı Bileşeni (Database) Oluşturulması

Veritabanı bileşenimiz **FloorDatabase** sınıfından miras alan soyut (abstract) bir sınıf olmak durumundadır. Bu sınıf **@Database** annotation’ı ile işaretlenerek Floor’da veritabanı işlemlerini yönetecek sınıf olarak

bildirilir. Veritabanının oluşturulmasında kullanılacak tabloların **entities** parametresi ile **@Database annotation**'ı içerisinde belirtilmelidir. Ayrıca kodun otomatik oluşturacağı sınıfların üretilebilmesi için aşağıdaki **part** ifadesi import satırı altında eklenmelidir.

```
part 'database.g.dart';
```

Burdaki “database” kısmı bulunduğu database.dart adlı dosya adından gelmektedir. Eğer Floor, Database bileşeni başka bir dosya içerisinde kodlanacaksa part kısmında bu dosya adı “database” yerine gelmelidir.

```
// database.dart

// gerekli paketler ve diğer dosyaların importları.
import 'dart:async';
import 'package:floor/floor.dart';
import 'package:sqflite/sqflite.dart' as sqflite;
import 'dao/person_dao.dart';
import 'entity/person.dart';

part 'database.g.dart'; // the generated code will be there

@Database(version: 1, entities: [Person])
abstract class AppDatabase extends FloorDatabase {
    PersonDao get personDao;
}
```

10.4.5. Kod Üretecinin Çalıştırılması

Floor'un veritabanı erişim katmanını oluşturması için aşağıdaki komut çalıştırılır.

```
flutter packages pub run build_runner build
```

Eğer kod dosyalarındaki her değişiklikte kod üretiminin otomatik olarak yapılması isteniyorsa;

```
flutter packages pub run build_runner watch
```

komutu kullanılabilir.

10.4.6. Üretilmiş Kodun Kullanımı

Veritabanının bir örneğine erişmek için veritabanı üretiminden sorumlu üretilmiş olan **\$FloorAppDatabase** sınıfı kullanılır. Bu sınıfın adı **\$Floor** ifadesi ve üretim için kullanılan sınıfın adından oluşmaktadır.

Veritabanının hazırlanması için **.build()** metodu await anahtar kelimesi eklenerek çağrılır.

Veritabanı işlemleri için veritabanı bileşenindeki DAO'lar kullanılacaktır. Aşağıdaki örnekte **personalDao** için bir referans alınmıştır. Alınan personalDao referansı üzerinden veritabanında işlem yapacak fonksiyonlar aşağıdaki gibi çağrılmaktadır.

```
final database = await
$FloorAppDatabase.databaseBuilder('app_database.db').build();

final personDao = database.personDao;
final person = Person(1, 'Frank');

await personDao.insertPerson(person);
final result = await personDao.findPersonById(1);
```

Bu bölümde Flutter için bir ORM yaklaşımı olan Floor paketinin kısa bir tanıtımı yapılmıştır. Floor'un sunduğu özellikler çok daha fazladır. Bunların tamamı ve tam çalışır uygulama örnekleri için Floor'un kendi sayfasını ziyaret ediniz.

<https://floor.codes/>

Bölüm Özeti

Bu bölümde kalıcı belleğe erişim yöntemleri ele alınmıştır. Her tür (formatsız verinin) dosyanın kaydedilmesi için dosya sistemine **dart:io** kütüphanesi ile erişilmektedir. **path_provider** kütüphanesi ile dosya sisteminde doğru klasöre erişim sağlanmaktadır. Basit formatlı veri saklamak için (key-value) **SharedPreferences** kullanılabilir. Daha karmaşık formatlı veri saklama için **SQLLite** veritabanı kullanılmaktadır. **SQLLite**'a doğrudan erişip SQL sorguları çalışırmak için **sqflite** veya **ORM** yaklaşımı ile çalışmak için **floor** kütüphaneleri kullanılmaktadır.

Kaynakça

<https://flutter.dev/docs/cookbook/persistence/reading-writing-files>

<https://developer.android.com/guide/topics/data/data-storage#pref>

<https://pub.dev/packages/sqflite>

<http://www.sqlitetutorial.net/>

<https://www.sqlite.org/datatype3.html>

<https://flutter.dev/docs/cookbook/persistence/sqlite#example>

<https://developer.android.com/topic/libraries/architecture/room>

<https://pub.dev/packages/floor>

<https://floor.codes/getting-started/>

11. Ağ VE FIREBASE

Giriş

Bu konu soket programlama, web servislerinin kullanımı, NoSql veritabanı ile çalışma gibi bir çok farklı ders içeriğine atıfta bulunmaktadır. Uzak sunucuya bağlanacak sistemlerde genelde bu başlıkların birden fazlasına aynı anda hakim olmak gereklidir. Anlatılan içerik bu noktada sadece yol gösterici bir kılavuz mahiyetindedir. İlgili alanlarda verilen linklerden daha derinlemesine konunun incelenmesi gereklidir.

11.1. Dart Programlama Dilinde Ağ İşlemlerine Giriş

Günümüzde bir uygulamanın ağ üzerinden servislere bağlanmadan çalışması neredeyse düşünülemez. Büyük-küçük her ölçekte uygulama İnternete veya değişik uzak veri kaynaklarına bağlanma bilgi alma ve bilgi gönderme gereksinimi duymaktadır. Ağ üzerinden iletişim en alt seviyede Soket programlama ile gerçekleştirilmektedir. Ama günümüzde genelde uygulama katmanı ek protokol ve servis yapıları kullanarak soket programlamanın getirdiği karmaşıklık uygulama geliştiricilerinden gizlenmektedir. Bu bölümde önce Dart programlama dilinde web servisleri ve http protokolü ile haberleşmek için gerekli kütüphaneler tanıtılmaktır. Sonrasında Google firmasının web servisi yazma ve sunucu ayarlama kulfetini kaldırmak amacıyla sunduğu bulut hizmeti Firebase ile bağlanacak bir uygulama geliştirmek için gerekli adımlar ele alınacaktır. Bu bölümde anlatılacak içerik Flutter ile değil Dart ile ilgilidir. Flutter arayüz düzenleme çerçevesidir ve ağ ve veri iletişimi Flutter çerçevesinden bağımsız olarak Dart programlama dili ile gerçekleştirilecektir. Ağ işlemlerinin de I/O işlemleri gibi asenkron yapılması gerektiğini unutmayın.

Veri传递i gerçekleştirmek için kullanacağımız kütüphaneleri incelemeden önce iletilecek verinin formatını ele alalım. Bellekte nesneler olarak sakladığımız verilerimiz ağ üzerinden bir akış şeklinde metin formda serileştirilerek(serialization) iletilemek durumundadır. İletimdeki metnin belli bir formu olmalıdır. Aksi halde karşı taraf mesajın karşılığında aynı nesneyi kendi belleğinde üretmek için takip edeceği adımları bilemez. JSON formatı (JavaScript Object Notation), Restful Web Servisleri ve http üzerinden veri传递i için en yaygın kullanılan mesaj formatıdır. Öncelikle JSON ve Serileştirme işlemini nasıl yapacağımızı anlayarak başlayalım.

11.2. JSON Serileştirme (Serialization)

Flutter'da kullanabileceğiniz iki farklı JSON Serileştirme yöntemi vardır:

- Manuel Yöntem
- Otomatik Kod Üreteci Kullanma Yöntemi

Hangisini seçeceğiniz genelde proje boyutuna ve serileştirme ihtiyacına göre verilebilecek bir karardır. Küçük bir projede manuel yöntem yerinde bir çözüm olacakken proje büyündükçe bu gereksiz tekrarlar ve elle çok fazla kod yazılması manasına gelecektir. Buna karşın küçük bir projede otomatik kod üretimi yaklaşımı da gereksiz bir yük getirecektir.

11.2.1. Küçük Projelerde Manuel JSON Serileştirme Uygulama

Flutter'da basit JSON serileştirme işlemi için **dart:convert** kütüphanesi kullanılır. Bu kütüphane JSON için **kodlama(encode)** ve **çözme(decode)** işlemlerini içermektedir. Aşağıda basit bir veri modeli (user) JSON olarak ifade edilmiştir.

```
{  
  "name": "John Smith",  
  "email": "john@example.com"  
}
```

Bu JSON veri modelini **dart:convert** kütüphanesi ile iki farklı şekilde serialize edebiliriz.

1. Yöntem: dart:convert içerisindeki **jsonDecode()** fonksiyonunu kullanmaktadır. Bu fonksiyon JSON ifadeyi bir **String** olarak alıp **Map<String, dynamic>** formda key-value çiftleri şeklinde dönüştürür. Bu yöntemde üretilen Map'de value kısmında gelecek değer türü baştan bilinmemektedir. Hangi veri türü olacağı çalışma zamanında belirlenmektedir. Bu da kod geliştirme sürecinde hatalı kod yazma ihtimalini arttırmır ve bakımı zorlaştırır. Örnekteki name ve email alanlarının adı değişimse bu ancak kod çalıştığında bir hata oluşturacaktır.

```
Map<String, dynamic> user = jsonDecode(jsonString);  
  
print('Ahmet, ${user['name']}!');  
print('Sertifikanız E-Posta hesabınıza gönderildi ${user['email']}.'');
```

2. Yöntem: Model sınıfı içerisinde JSON serileştirme işlemini yapmaktadır. Bu sayede 1. Yöntemdeki çalışma zamanı veri türü eşleşmesinden doğan sıkıntı bir nebzə azaltılmış olur. Bunun için User adında bir sınıf tanımlaması aşağıda yapılmaktadır. User sınıfı JSON işlemleri için;

- User.fromJson() isimlendirilmiş yapılandırıcı fonksiyonu ile JSON metinden veri modeli üretilir.
- toJson() şeklinde bir fonksiyon tanımlanarak tersi şekilde veri modeli JSON metne döndürecek kodlama yapılır.

```
class User {  
    final String name;  
    final String email;  
  
    User(this.name, this.email);  
  
    User.fromJson(Map<String, dynamic> json)  
        : name = json['name'],  
          email = json['email'];  
  
    Map<String, dynamic> toJson() => {  
        'name': name,  
        'email': email,  
    };  
}
```

Bir önceki örnektenden farklı olarak JSON'u encode ve decode edecek işlemler veri modelinin içerisinde alınmıştır. Bu sayede bir önceki yöntemdeki kodlama aşağıdaki gibi yazılabılır.

```
Map<String, dynamic> userMap = jsonDecode(jsonString);  
var user = User.fromJson(userMap);  
  
print('Ahmet, ${user.name}!');  
print('Sertifikanız E-Posta hesabınıza gönderildi ${user.email}.');
```

Bir User nesnesini JSON metne dönüştürmek için jsonEncode() fonksiyonunu User nesnesi ile çağrırmak yeterlidir. toJson() fonksiyonunu belirtmek gerekmektedir. jsonEncode() tarafından otomatik olarak çağrılır.

```
String json = jsonEncode(user);
```

2. Yöntemle JSON serileştirme süreçleri User sınıfı içerisinde hapsedilmiştir. Bu sayede dışarıdaki kod bloklarında JSON serileştirme ile ilgili kod parçaları olmayacak hepsi User nesnesine referansta olacaktır. Ne yazık ki kodun çalışma zamanı oluşacak hataları tamamen yok edilmemiştir. Ayrıca bize JSON besleyen servislerin davranışları ve geleceği değerler her zaman yalnız veri modelleri değildir. Çok daha karmaşık işleyişleri olabilmektedir. Bu gibi durumlarda manuel bir veri işleme şekli çok da yerinde bir karar olmayacağıdır. Otomatik kod üretici yöntemler JSON ile haberleşen servislerden gelen verileri alıp işlemek için bize çok daha konforlu bir imkân sunmaktadır.

11.2.2. Kod Üretim Kütüphaneleri Aracılığıyla JSON Serileştirme

Pub.dev Internet adresinde birçok JSON kütüphanesi bulunmaktadır. Bu içerikte json_serializable kütüphanesi ile JSON serileştirme işleminin nasıl otomatik hale getirileceği gösterilecektir. İşlemleriniz için doğru kütüphanenin seçimi noktasında Flutter Favorite listesinden faydalansınız. Otomatik serileştirme işlemi sayesinde artık JSON serileştirme gerektiren kod bloklarına elle müdahale etmeyecek ve bu sayede çalışma zamanı görüşebilen serileştirme uyumsuzluklarından kaynaklı hataları almayacağız. Şimdi json_serializable kütüphanesinin kullanımını adım adım inceleyelim.

1. json_serializable Kütüphanesinin Projeye Eklenmesi:

json_serializable Kütüphanesinin kullanımı için pubspec.yaml dosyasına biri dependencies ve ikisi dev_dependencies bölümünde olmak üzere üç kayıt tanımlanacaktır. dev_dependencies kısmı projenin yayımlanmasında eklenmeyip sadece geliştirme aşamasında kullanılan paketlerin tanımlandığı kısımdır.

```
dependencies:  
  json_annotation: <son_surum>  
  
dev_dependencies:  
  build_runner: <son_surum>  
  json_serializable: <son_surum>
```

2. json_serializable ile Veri Modelinin Üretilmesi:

Bir önceki başlıkta kullandığımız User veri modelini json_serializable kütüphanesine uygun hale getiriyoruz.

```
import 'package:json_annotation/json_annotation.dart';  
part 'user.g.dart';  
  
@JsonSerializable()  
class User {  
  User(this.name, this.email);  
  String name;  
  String email;  
  factory User.fromJson(Map<String, dynamic> json) => _$UserFromJson(json);  
  Map<String, dynamic> toJson() => _$UserToJson(this);  
}
```

Json_serializable kütüphanesi annotation ve otomatik kod üretim tekniklerini kullanmaktadır. *part '*.g.dart'* kısmı ile otomatik üretilecek kod dosyası bildirilmektedir. Burada belirtilen dosya adı sınıf adı ile aynıdır.

User sınıfı **@JsonSerializable()** ile işaretlenmiştir. İçerdiği **fromJson()** ve **toJson()** fonksiyonları da otomatik kod üretici ile oluşturulacak **_UserFromJson()** ve **_UserToJson()** fonksiyonlarına yönlendirilir. Bu sayede JSON dönüştürme işleminin iç kodlaması programcının müdahalesinden soyutlanmış olur.

Ayrıca otomatik kod üretiminde özel isimlendirme **@JsonKey** veya benzer şekilde **@JsonSerializable(fieldRename:)** annotationları ile verilebilir. **@JsonKey** özelliği farklı parametrelerle varsayılan değer, zorunlu veya işlem dışı olma gibi özellikleri de her bir alana verilebilir.

```
@JsonKey(name: 'hesaplanan_alan')  
final int hesaplananAlan;  
  
@JsonKey(defaultValue: false)  
final bool isAdult;  
  
@JsonKey(required: true)  
final String id;  
  
@JsonKey(ignore: true)  
final String verificationCode;
```

3. Kod Üretecinin Çalışmasının Tetiklenmesi:

Otomatik kod üretimi için aşağıdaki komutları proje klasöründe terminalde çağırabiliriz. Kod üretim işleminin çağrılmamasından önce otomatik üretilmiş kod için var olan girdilerin hata vermesi normaldir.

Aşağıdaki komutla **user.g.dart** şeklinde yukarıdaki örnekte verilen dart dosyası üretilmiş olacak ve bu hata gidecektir.

Eğer sadece bir defaya mahsus otomatik kod üretimi tetiklenecekse:

```
flutter pub run build_runner build
```

Eğer kod üretim işleminin arka tarafta dosyaların güncellenmesini değişimleri takip ederek otomatik tetiklenmesini istiyorsak:

```
flutter pub run build_runner watch
```

4. **Json_serializable** ile Üretilen Modelin Kullanımı:

Otomatik kod üretici kullandığımızda artık model sınıf içerisinde kodlama gereksinimi olmayacağındır. JSON encode ve decode işlemleri için aşağıdaki kodları yazmamız yeterlidir.

```
Map<String, dynamic> userMap = jsonDecode(jsonString);
var user = User.fromJson(userMap);

String json = jsonEncode(user);
```

Otomatik kod üretimini iç içe kullanılan sınıf yapıları ve farklı modeller için de kullanmadan mümkün değildir. Konu hakkında daha fazla örnek ve detay için ilgili paketlerin referans sayfasını inceleyiniz. Aşağıda verilen son bağlantıda GitHub üzerinde **json_serializable** paketi için örnekler mevcuttur.

<https://api.dart.dev/stable/dart-convert>

<https://api.dart.dev/stable/dart-convert/JsonCodec-class.html>

https://pub.dev/packages/json_serializable

https://github.com/dart-lang/json_serializable/blob/master/example/lib/example.dart

11.3. Ağ İşlemleri

Flutter'da ağ üzerinden iletişim yapmak için daha önce de belirttiğimiz gibi çok sayıda kütüphane kullanılabilir. Farklı türde protokoller ile haberleşmek için ilgili protokole yönelik yazılmış paketin tercih edilmesi gerekmektedir. Buradaki içerikte internetteki hizmetlerin yaygın olarak Restful API ile yazılmış olması ve http protokolü ile konuşuyor olması sebebiyle http iletişimini Flutter'da gerçekleyebileceğimiz en sade kütüphane olan **http** kütüphanesi ile anlatım yapılacaktır. Günümüzde çift yönlü veri传递ası ihtiyacından dolayı servislerin **WebSocket** temelli hazırlandığını görmekteyiz. **WebSocket** haberleşmesi için de Flutter'da paketler aracılığıyla destek verilmektedir. Her iki kütüphaneye aşağıdaki bağlantılardan erişebilirsiniz. İlgili kütüphane sayfaları detaylı örnekler ve referanslar içermektedir.

<https://pub.dev/packages/http>

https://pub.dev/packages/web_socket_channel

Ağ iletişimini için Android Platformu özel izin alınmasını gerektirmektedir. Android işletim sisteminde yüklenen uygulamalar **AndroidManifest.xml** dosyası içerisinde uygulamanın telefon içerisinde standart erişim hakları dışındaki kaynaklara ulaşmak istediği alması gereken izinler tanımlanmalıdır. Eğer bu izinler tanımlanmazsa Android işletim sistemi ilgili uygulamaya asla izin vermeyecektir. İnternet erişimi de bu gerekli izinler arasında yer almaktadır. Aşağıdaki kod parçası **AndriodManifest.xml** dosyasına eklenmesi gereken kısmı göstermektedir.

```
<manifest xmlns:android...>
...
<uses-permission android:name="android.permission.INTERNET" />
<application ...
</manifest>
```

11.3.1. http Paketi ile JSON Veri İletimi

http Paketini kullanarak http protokolü üzerinden haberleşen Web Servislerini sorgulayabilir, http protokolünün POST, GET, DELETE, PUT, PATCH taleplerini sunucuya gönderen hazır fonksiyonları mevcuttur. Bunların kullanımın detaylarını paketin sayfasından ve basit kullanım örneklerini aşağıda verilen Flutter Cookbook sayfasından inceleyebilirsiniz.

<https://docs.flutter.dev/cookbook/networking>

Şimdi HTTP GET ile bir web sunucusundan bilgi alıp işleyen örnek için gereken adımları inceleyelim.

1. Tüm paketlerin kullanımında olduğu gibi http paketi için de pubspec.yaml dosyası içerisinde gereksinimlere eklenir.

```
dependencies:
  http: <latest_version>
```

2. Kütüphanenin kullanılacağı sayfada import ile http paketi belirtilir. Burada paket as http ile etiketlenmiştir. Bu sayede kod içerisinde http. Şeklinde kullanım ile içerdigi doğrudan çağrılar fonksiyonları görebiliriz.

```
import 'package:http/http.dart' as http;
```

3. HTTP GET için **http.get()** fonksiyonu kullanılır. Bu fonksiyon **Future<http.Response>** türünde bir cevap dönmektedir. Buradan anlaşılacağı üzere **http.get()** çağrısından donecek bilgi bize asenkron bir kodlama gerektirecektir. Örneklerde

<https://github.com/typicode/jsonplaceholder>

adresinde yayınlanan sahte bir Restful Web Servisi üzerinde sorgulama yapılmaktadır.

```
Future<http.Response> fetchAlbum() {
  return http.get(Uri.parse('https://jsonplaceholder.typicode.com/albums/1'));
}
```

4. **fetchAlbum**'ün bize formatlanmış bir JSON verisi döndürmesini beklemekteyiz fakat **http.get()** ile gelen veri salt bir http **Response** (Cevap) objesidir. Bunun sebebi sonucun normal web sayfalarında aldığımız http Response mesajları gibi olmasıdır. http paketi HTTP protokolünün varsayılan davranışını sunmaktadır. Taşınan veri JSON yerine düz metin XML, Dosya dahi olabilir. Burada gelecek olan mesajı içeriğinin JSON formatında olduğunu ve formatını bilerek bir kodlama yapmamız gerekmektedir. Restful Web Servislerinde genel olarak dönülen sonucun formatı bir doküman veya bilgilendirme servisi olarak verilmektedir. Önce servisin geleceği JSON yapısına uygun bir Album sınıfı oluşturulmalı sonrasında **http.Response** objesi içerisinde veri çekecek şekilde JSON decode işlemi yapılmalıdır. Aşağıda sırası ile bu kodlamalar verilmektedir.

```

class Album {
    final int userId;
    final int id;
    final String title;

    Album({
        required this.userId,
        required this.id,
        required this.title,
    });

    factory Album.fromJson(Map<String, dynamic> json) {
        return Album(
            userId: json['userId'],
            id: json['id'],
            title: json['title'],
        );
    }
}

```

```

Future<Album> fetchAlbum() async {
    final response = await http
        .get(Uri.parse('https://jsonplaceholder.typicode.com/albums/1'));

    if (response.statusCode == 200) {
        return Album.fromJson(jsonDecode(response.body));
    } else {
        throw Exception('Failed to load album');
    }
}

```

http.Response objesinin burada önce **statusCode** değeri kontrol edilir. http cevaplarında durum kodları sunucunun ilgili talebi yerine getirip getiremediğini belirtmektedir. 200 OK cevabı talebin düzgün tamamlandığını ifade eder. Bu teyitten sonra **response.body**'e erişerek bir **Album** objesi için oluşturulmuş JSON verisi okunması beklenir. Aksi halde body kısmında sunucunun döndüğü bir hata mesajı olması veya boş gelmesi Restful servisin çalışma biçimine bağlıdır. Cevap başarılı olsa dahi sonucun istenilen JSON formatında olmama ihtimali vardır. Bu nedenle dönen cevabin dönüşüm işleminin başarı ile tamamlanıp tamamlanmadığı da kontrol edilmelidir.

5. Kod içerisinde `fetchAlbum` fonksiyonu istenilen bir yerde çağrılr. Fonksiyonun dönüş değeri ise bir **late** değişkende saklanır. Yani bu fonksiyon çağrıldığında http request gönderilecek ama cevabin dönüşü beklenmeden sonraki kod satırları işletilecektir.

```

class _MyAppState extends State<MyApp> {
    late Future<Album> futureAlbum;

    @override
    void initState() {
        super.initState();
        futureAlbum = fetchAlbum();
    }
    // ...
}

```

Örnekte Stateful bir widget'ın **initState** metodunda **fetchAlbum()** çağrılmıştır. Yani widget belleğe ilk yükleniği anda http talebi gönderilecektir.

6. Burada talebin gerçekleştiğinde bunu ekrana yansıtın bir yaklaşımı ihtiyacımız vardır. Yani **futureAlbum** değişkeninin içeriği değiştiğinde (durumu değiştiğinde) ekranın güncellenmesi gerekmektedir. **FutureBuilder** adındaki özel bir Widget bu işlevi yerine getirmektedir.

<https://api.flutter.dev/flutter/widgets/FutureBuilder-class.html>

FutureBuilder yapılandırıcısı takip etmesi gereken Future objesini ve **future** parametresi olarak ekran çizimini yönetecek kısmı **builder** parametresi olarak almaktadır. **builder** parametresine verilecek fonksiyonda **snapshot**, Future objesinin **loading**(yükleniyor), **success**(başarılı) ve **error**(hata) şeklinde üç durumunu belirten bir kodlama imkanı sunar. Bu sayede Future objesinin durumuna göre ekran çizimi tetiklenmiş olur. Aşağıda bu kısmı içeren kodlar yer almaktadır.

```
FutureBuilder<Album>(
  future: futureAlbum,
  builder: (context, snapshot) {
    if (snapshot.hasData) {
      return Text(snapshot.data!.title);
    } else if (snapshot.hasError) {
      return Text('${snapshot.error}');
    }

    // Eğer success veya error durumu yoksa yükleniyor ifadesi getir.
    return const CircularProgressIndicator();
  },
)
```

Burada anlatılan örneğin bütün haline

<https://docs.flutter.dev/cookbook/networking/fetch-data>

adresinden ulaşabilirsiniz.

11.4. Firebase Kullanımı

Firebase Google firmasının sunduğu Backend-as-a-Servis (BaaS) bulut yaklaşımıdır. Çoğu web sayfası veya mobil uygulamanın geliştirmesinde uzak bir sunucuda bulunan veritabanı ve bu veritabanının önünde erişimi denetleyen bir web servisi yazılması gereklidir. Bu hızlı uygulama geliştirmek isteyen ekipler için ciddi bir külfettir. Çünkü enerjisini mobil uygula geliştirmek için harcayacak ekibin veritabanı düzenlemesi, sunucu kiralama ve bakımı, muhtemelen farklı bir dilde yazılmış bir web servisi uygulaması geliştirmesi ve en kritiklerinden biri olarak bu sistemin her aşaması için performans ve güvenlik sağlama gereksinimidir. Firebase bu açıdan Google firmasının sunduğu bir servisler hiyerarşisi olarak özellikle mobil uygulama geliştirme sürecini hızlandırmaya yönelik çözümler sunmaktadır. Günümüz için sunulan çözümleri; Kimlik doğrulama (Authentication), Veritabanı, Bulut Tabanlı Fonksiyonlar, Yapay Zekâ Motoru, Bulut Veri Depolama, Bulut Web Sayfası, Bulut Mesajlaşma, Uzak Konfigürasyon şeklinde sıralanabilir. Sunulan hizmetler ve kullanımı ile ilgili detaylı bilgi Firebase sayfasından edinilebilir.

<https://firebase.google.com/products-build>

Flutter ile Firebase alt yapısında kullanılan tüm özelliklere ulaşmak için farklı kütüphaneler kullanılmaktadır. Bu kütüphanelerin toplandığı ana çatının adı **FlutterFire**'dır. Aşağıdaki bağlantı ile FlutterFire sayfasını ziyaret edebilirsiniz.

<https://firebase.flutter.dev/>

FlutterFire sayfasında Firebase özellikleri için kullanılan kütüphanelerin ve geliştirme durumlarının bir listesi verilmektedir. Flutter sürekli iyileştirilen bir çatı olduğu için buradaki listede sürekli bir güncelleme olduğunu belirtelim. Biz Firebase bağlantısı için temel özelliklere ve bir NoSQL Firebase veritabanı yapısında veri kaydetme adımlarını inceleyeceğiz. Eğer Firebase'in farklı bir servisini kullanmak isterseniz FlutterFire ve Firebase'in sayfasındaki dokümantasyonlara bakabilirsiniz.

11.4.1. Firebase Kullanıma Hazırlık

Firebase servislerini projenizde kullanabilmek için öncelikler bir Firebase projesi oluşturmanız gerekmektedir. Bunla ilgili yönergelere aşağıdaki bağlantıdan bakabilirsiniz.

<https://console.firebaseio.google.com/>

Herhangi bir Firebase servisini kullanabilmek için öncelikli olarak `firebase_core` paketinin projeye eklenmesi gereklidir. `firebase_core` paketi Firebase sistemine bağlantıyla alakalı temel işlevselligi yerine getirmektedir. Kurulum için `pubspec.yaml` dosyasına gereksinim başlığı altında eklenmelidir.

Sonrasında uygulamanın çalışacağı platforma yönelik hazırlıkların yapılması gereklidir. Firebase bağlantıda bağlanacak uygulama için size özel bir API key üretmekte ve ilgili uygulamanın bu API key ile talep göndermesi gerekmektedir. Android, iOS, MacOS ve Web ortamından erişim için gerekli kurulum işlemlerini aşağıdaki bağlantılardan takip edebilirsiniz.

<https://firebase.flutter.dev/docs/installation/android>

<https://firebase.flutter.dev/docs/installation/ios>

<https://firebase.flutter.dev/docs/installation/macos>

<https://firebase.flutter.dev/docs/installation/web>

11.4.2. FlutterFire Başlatılması

Firebase servislerine bağlanmak için Flutter projesinde önce bağlantının kurulması gereklidir. Bu işlem için **Firebase** sınıfında tanımlanan `initializeApp()` fonksiyonunu çağrırmalıyız. Bu fonksiyonun işini bitirmesi herhangi bir Firebase çağrısından önce yapılmalıdır. Bu nedenle aşağıdaki gibi await anahtar kelimesi eklenerek yazılmalıdır.

```
await Firebase.initializeApp();
```

Firebase ile yapacağımız tüm çağrılar asenkron çalışacaktır. Bu nedenle donecek cevaplar Future objeleri şeklinde olacaktır. Sayfada bu **Future** objelerinin gösterimi bir önceki başlıkta bahsi geçen **FutureBuilder** Widgetinin kullanımı ile sağlanmalıdır.

Firebase servislerinin Flutter içerisinde kullanımı http paketi ile web servisi kullanıma oldukça benzerdir. Firebase servisleri ve her biri için verilecek örnekler başlı başına bir ders içeriği olacağı için burada ilgili dokümantasyona referans vererek konuyu kısa tutacağız.

<https://firebase.flutter.dev/docs/overview>

Firebase servislerine bağlantı ile ilgili örnek video ve çalışma örneklerine Flutter dokümanındaki Firebase referanslar sayfasından erişebilirsiniz.

<https://docs.flutter.dev/development/data-and-backend/firebase>

Bölüm Özeti

Dart programlama dili ağ üzerinden iletişim kurmak için temek soket iletişimini destekler. Buna ek olarak web servislerini sorgulama işlemlerini kolaylaştıracak ek kütüphaneler bulunmaktadır. **http** kütüphanesi Restful web servislerini sorgulamak için kullanılabilir. Diğer kütüphaneler için pub.dev sayfası incelenmelidir. Popüler yakışımalar **FlutterFavorite** programı altında yayınlanır.

Ağ iletişiminde serileştirme obje-metin dönüşümü gereklidir. **dart:convert** ile JSON serileştirme yapılabilir ve **json_serializable** ile bu süreç otomatikleştirilir.

Firebase, Google firmasının sunduğu BaaS yaklaşımıdır ve Flutter içerisinde erişim desteği sunulmaktadır.

Kaynakça

<https://docs.flutter.dev/development/data-and-backend/json>

<https://api.dart.dev/stable/dart-convert>

<https://api.dart.dev/stable/dart-convert/JsonCodec-class.html>

https://pub.dev/packages/json_serializable

https://github.com/dart-lang/json_serializable/blob/master/example/lib/example.dart

<https://pub.dev/packages/http>

https://pub.dev/packages/web_socket_channel

<https://docs.flutter.dev/cookbook/networking>

<https://github.com/typicode/jsonplaceholder>

<https://api.flutter.dev/flutter/widgets/FutureBuilder-class.html>
<https://docs.flutter.dev/cookbook/networking/fetch-data>

<https://firebase.google.com/products-build>

<https://firebase.flutter.dev/>

<https://console.firebaseio.google.com/>

<https://firebase.flutter.dev/docs/installation/android>

<https://firebase.flutter.dev/docs/installation/ios>

<https://firebase.flutter.dev/docs/installation/macos>

<https://firebase.flutter.dev/docs/installation/web>

<https://firebase.flutter.dev/docs/overview>

<https://docs.flutter.dev/development/data-and-backend/firebase>

12. PAKET KULLANIMI

Giriş

Pubspec.yaml dosyasını paket gereksinimleri için bundan önceki pekçok bölümde ele almıştık. Bu bölümde paket gereksinimleri, çakışmaları, versiyon uyuşmazlıklar gibi paket yönetimindeki daha detay konular ele alınacaktır.

12.1. Paket Kullanımı Temel Bilgiler

Başka kullanıcıların geliştirdikleri ve Flutter&Dart ekosistemine sundukları paketleri kendi projelerinizde kullanma imkânınız vardır. Benzer şekilde sizde geliştirmiş olduğunuz paketleri ekosisteme yükleyerek diğer geliştiricilerin kullanımına sunabilirsiniz. Bu sayede yeni bir uygulama geliştirileceği zaman her şey baştan yazılacak zorunda kalmayacaktır.

Paket kavramı iki farklı tanımlama içermektedir; Package (Paket), Plugin (Eklenti).

Package (Paket): Bir Dart paketi en yalın hali ile **pubspec.yaml** dosyasına sahip bir klasör olarak tanımlanabilir. Bir paket başka paketlere bağımlılık, resim, müzik, video ve benzeri veri dosyalarını barındırabilir. **Pub.dev** sayfası altında Google ekibinin geliştirdiği birçok paket mevcuttur.

Plugin (Eklenti): Eklentiler paketlerin özel bir formudur. Normal paketlere ek olarak içeriğinde Dart projelerinin alttaki platform (Android, iOS, Web, Masaüstü) ile iletişim kurmasını sağlarlar. Plugin normal paketlerin içerdiklerinin yanında ek olarak alttaki platform ile konuşmayı sağlayan kodlamalar (Android için Java veya Kotlin, iOS için Swift gibi) da içerirler.

Ekosistemdeki var olan paketler birçok temel ihtiyacı karşılamaktadır. Http tabanlı web servisleri ile iletişim için **http** paketi, özel bir navigasyon-yönlendirme altyapısı için **fluro** paketi, Cihaz ile etkileşime girme farklı uygulamalara veri gönderme için **url_launcher** ve **battery** paketi ve Firebase bağlantısı için **FlutterFire** paket koleksiyonu temel ihtiyaçları karşıyan paketler arasında sayılabilir.

12.2. İhtiyaca Yönelik Paket Bulma

Flutter&Dart ekosistemi paket paylaşımı için pub.dev sayfasını ana merkez olarak kullanmaktadır. Pub.dev sayfası altında paketin yayınıyor olması için belli kriterleri sağlaması gerekmektedir. Örneğin her paketin bir read.me sayfası, örnek kullanımı, ek dokümantasyonu, değişim kaydı (change log) gibi bileşenleri yayınıyor olması gerekmektedir.

<https://pub.dev/flutter/packages>

Yukarıdaki bağlantı sayesinde Flutter içerisinde kullanılmaya uyumlu tüm paketlerin bir listesini görebilirsiniz. Bu aşamada belki en zorlayıcı kısmı bu kadar çok seçenek arasından işinize yarayacak doğru paketi bulmaktır. Google ekibi bu noktada paketlere tam hakim olmayan geliştiricilerin başlangıç için en popüler ve kullanımı en uygun paketleri edinmesi için bunları bir liste haline getirerek sunarlar. Bu liste **Flutter Favorites** programı olarak adlandırılmaktadır ve aşağıdaki bağlantıdan incelenebilir.

<https://pub.dev/flutter/favorites>

12.3. Uygulamadaki Paket Gereksinimlerini Ayarlama

Daha önceki bölümlerde gereken paketleri kullanmak için bunların **pubspec.yaml** dosyasında **dependencies** başlığı altında olmaları gerektiğini vurgulamıştık. Bu ekleme yapıldıktan sonra paketleri pub.dev kaynağından çekilmesi için ya proje klasöründeyken terminalden **flutter pub get** komutu çalıştırılmalı ya da kullandığınız IDE içerisindeki kısayol ile bu komut tetiklenmelidir. Önerdiğimiz IDE olan VS Code ise bu işlemi biz **pubspec.yaml** dosyasını her kaydettiğimizde otomatik tetiklediğini söylemişlik. Eğer VS Code paketleri çekme komutunu otomatik tetiklemezse editörde **pubspec.yaml** dosyası açıkken sağ üst köşede çıkan **Get Packages** düğmesi kullanılarak da paketler indirilebilir.



VS Code'da **pubspec.yaml** dosyası düzenleme ekranı üzerinde **Get Packages** düğmesi

Bu aşamadan sonra kullanmak istediğimiz paketi ilgili Dart kodunda **import** anahtar kelimesi ile ekliyoruz.

Bu aşamaların her paket için nasıl gerçekleşeceği pub.dev sayfasında paketin **installing** sekmesi altında anlatılmaktadır.

12.4. Çakışmaları Giderme

Eğer kullandığını iki paket de aynı pakete bağlı ise ve farklı versiyonu ile çalışabiliyorlarsa bu durumda ortak kullanımda olan paketin hangi versyonunu istediğimizi doğrudan belirtmek yerine bir aralık belirtmeyi tercih edebiliriz.

```
dependencies:  
  url_launcher: ^5.4.0    # İyi, >= 5.4.0 ve < 6.0.0 olan versiyonları kullan  
  image_picker: '5.4.3'   # İyi değil, sadece 5.4.3 sürümünü kullan.
```

Bu işlem için yukarıdaki örnekte olduğu gibi versiyon numarasından önce (^) simbolü kullanılır. Eğer bu şekilde bir kullanım yaparsak pub paket yönetim sistemi bizim yerimize versiyon uyuşmazlıkların otomatik olarak çözecektir.

Eğer tam tersi bir durum ile karşılaşılır ve illaki url_launcher'ın 5.4.0 versiyonunun kullanılması gerekirse **dependency_overrides** başlığı altında ilgili sürüm ile paket belirtilir.

```
dependencies:  
  some_package:  
  another_package:  
  dependency_overrides:  
    url_launcher: '5.4.0'
```

12.5. Paket Bağımlılıkları ve Versiyonlarının Yönetimi

Bazı durumlarda paketlerin gereksinimlerinin veya kullanıldıkları özelliklerden dolayı projenizin çalıştırılması imkânsız hale gelebilir. Bu gibi durumları engellemenin en iyi yolu kullanılan paketler için belli bir versiyon aralığı tanımlamaktır. Tüm paketlerin aktif bir versiyon numarası vardır. Güncel versiyon numarası pub.dev sayfasında ilgili paketin adının yanında yer almaktadır. Eski versiyonlarına da pub.dev sayfasında pakete özel sayfasına girdiğinizde **Versions** sekmesinden görebilirsiniz. Örneğin url_launcher paketinin versiyon sayfası aşağıdaki gibi görülmektedir.

url_launcher 6.0.14

Published Nov 20, 2021 · flutter.dev Null safety

FLUTTER ANDROID IOS LINUX MACOS WEB WINDOWS

3.45K

Readme Changelog Example Installing Versions Scores

Stable versions of url_launcher

Version	Min Dart SDK	Uploaded		
6.0.14	Null safety	2.14	Nov 20, 2021	View Download
6.0.13	Null safety	2.14	Nov 15, 2021	View Download
6.0.12	Null safety	2.14	Sep 23, 2021	View Download
6.0.11	Null safety	2.14	Sep 18, 2021	View Download
6.0.10	Null safety	2.12	Sep 2, 2021	View Download

url_launcher'ın sadece belli versiyonları ile çalışacak bir bağımlılığı aşağıdaki gibi tanımlanabilir.

```
dependencies:  
  url_launcher: '>=5.4.0 <6.0.0'
```

Bir uygulamanın geliştirilmesinde ilk defa indirilen paket bilgileri **pubspec.lock** dosyasına kaydedilir ve farklı bir geliştirici paketleri indirmesi gerekirse bu dosyadaki önceden indirilmiş versiyon bilgisine bakılarak aynı versiyonun elde edilmesi sağlanır. Eğer kullanılan paketlerin daha güncel versiyonlara geçirilmesi isteniyorsa **flutter pub upgrade** komutu ile pubspec.yaml dosyasında izin verilen en güncel versiyona güncelleştirme yapılır.

Bazı durumlarda pub.dev altında yayınlanmayan paketlerin kullanımı gerekebilir. Örneğin bir firmanın kendine özgü paketlerini kullanması veya bir git reposunda paylaşılan bir paketin kullanımı gibi. Bu gibi durumları pubspec.yaml dosyasında paket tanılaması yaparken ek yol bilgisi ile gösterebiliriz.

Aşağıdaki örnekte plugin1 paketi proje klasörü içerisinde plugin1 adlı bir alt klasör olarak tanımlanmaktadır ve bu bilgi path başlığında verilmektedir.

```
dependencies:  
  plugin1:  
    path: ../../plugin1/
```

Aşağıdaki örnekte ise ilgili paket bir git reposu olarak paylaşılmaktadır.

```
dependencies:  
  plugin1:  
    git:  
      url: git://github.com/flutter/plugin1.git
```

Eğer paket bir git reposu içerisinde bir alt paket şeklinde tanımlı ise aşağıdaki gibi belirtilebilir.

```
dependencies:  
  package1:  
    git:  
      url: git://github.com/flutter/packages.git  
      path: packages/package1
```

İsterseniz kendi paketlerinizi oluşturabilirsiniz. Bunun için proje şablonu olarak paket oluşturma şablonu ile **flutter create** komutunu kullanabilirsiniz.

```
flutter create --template=package hello
```

Paket şablonu Flutter projesi şablonundan bazı farklılıklar göstermektedir. Bir paketin içermesi gereken temel dosyaların bir örneğini oluşturmaktadır. Paketlerin içerdiği her fonksiyonun tek tek test edilmesi önemlidir. Bu nedenle paketler oluşturulurken test klasörü altındaki unit test bloklarının yazılması ayrı bir öneme sahiptir. Bunu gerçekleştirmeyen paketler pub.dev altında kesinlikle yayınlanamazlar. Kendi paketlerinizi oluşturma ile ilgili daha detaylı adım listesi için aşağıdaki bağlantıyı ziyaret ediniz.

<https://docs.flutter.dev/development/packages-and-plugins/developing-packages>

Bölüm Özeti

Flutter&Dart için paket dağılımları **pub.dev** web sitesi üzerinden koordine edilmektedir.

FlutterFavorite listesi ile yeni başlayanların doğru paketleri bulması sağlanır.

pubspec.yaml ve **pubspec.lock** dosyaları içerisinde projedeki paketlerin yönetimi sağlanmaktadır.

Kendi paketlerinizi oluşturup yayinallyabilirsınız. Bunun için pub.dev in paket yayinallyama politikasına uygun olarak yayınlanan paketin kod kısmı dışında unit testleri ve dokümantasyonunun da tamamlanması gereklidir.

Kaynakça

<https://pub.dev/flutter/packages>

<https://pub.dev/flutter/favorites>

<https://docs.flutter.dev/development/packages-and-plugins/using-packages>

<https://docs.flutter.dev/development/packages-and-plugins/developing-packages>

<https://docs.flutter.dev/development/packages-and-plugins/favorites>

<https://docs.flutter.dev/development/packages-and-plugins/background-processes>

13. TEST VE HATA AYIKLAMA

Giriş

Bu bölümde önce hata ayıklama yaklaşımları sonra da test yöntemleri ele alınacaktır. Hata ayıklama işlemi diğer dillerle oldukça benzerdir. Burada Flutter'ın hata ayıklamada sunduğu ek araçlar tanıtılacaktır. Test

işlemleri de unit test ve entegrasyon testleri olarak ayrılır. Flutter sunduğu ek yapı sayesinde bize arayüzü test edecek bir yaklaşımda vermektedir.

13.1. Flutter Uygulamalarında Hata Ayıklama

Flutter uygulamalarında hata ayıklama işlemi için aşağıdaki araçlar kullanılmaktadır.

· **DartDevTools:** İnternet tarayıcısında çalışan performans ve profil çökarma işlemlerine yardımcı olan bir dizi araçtan oluşan bir araç setidir. VS Code, Android Studio/IntelliJ geliştirme ortamları içerisinde de çalıştırılabilir. Dart Dev Tools ile;

- o Bir Flutter uygulamasındaki arayüz düzeni ve uygulama durumu incelenebilir.
- o Arayüzde performansı kötü kullanan bölümlerin tanısı yapılabilir.
- o Flutter ve Dart uygulamalarının CPU kullanım profilleri çıkarılabilir.
- o Flutter uygulamalarının Ağ kullanım profilleri çıkarılabilir.
- o Flutter ve Dart uygulamalarında kaynak-kod seviyesinde hata ayıklama yapılabilir.
- o Flutter ve Dart uygulamalarında bellek temelli sorunlar tanılanır.
- o Bir Flutter veya Dart konsol uygulamasının genel çalıştırılmasındaki kayıtlar incelenebilir ve tanı bilgileri görüntülenebilir.
- o Kodun ve uygulamanın ebatları ölçülebilir.

DartDevTools, hata ayıklama işlemleri için Flutter&Dart çerçevesinde en önemli parçalardan biridir. Flutter dokümantasyonunda DartDevTools kullanımı hakkında detaylı bir doküman listesi bulunmaktadır. Aşağıdaki bağlantılardan bu doküman üzerinden daha fazla bilgiye ulaşılabilir.

<https://docs.flutter.dev/development/tools/devtools/overview>

· Android Studio/IntelliJ ve VS Code geliştirme ortamlarında Dart ve Flutter eklentilerini kurmanız halinde bütünsel olarak gelen hata ayıklama, kod analiz, durma noktası (**breakpoint**) kullanımı gibi özellikleri Flutter uygulama geliştirme sürecinde de kullanabiliriz.

· **Flutter Inspector:** DartDevTools içerisinde veya doğrudan IDE içerisinde açabileceğimiz bu bileşen bir Flutter uygulamasında widget ağacını, ekran düzenini görselleştirme ve incelemek için kullanılmaktadır. Var olan widget ağacını anlamamızı ve bir hata varsa bunu bulmamızı sağlamaktadır. Daha geniş bir anlatımı aşağıdaki bağlantılardan inceleyebilirsiniz.

<https://docs.flutter.dev/development/tools/devtools/inspector>

DartDevTools (veya kısaca **DevTools**) ile uygulamanızda hata ayıklama veya çalışma performansını izlemek istiyorsanız uygulamanın **debug mode** veya **profile mode** ile çalıştırılması gerekmektedir. **Release mode** ile çalıştırılan bir uygulama DevTools ile incelenemez. Eğer uygulamanızın gerçek bellek ve işlemci kullanım performansını incelemek isterseniz **profile mode** ile çalıştırılması gereklidir. **Debug mode**, hata ayıklama işlemi için standart dart kodu içerisine çok fazla ek adım ve log kaydı yüklemektedir.

Eğer Flutter ve Dart kodlamasına uygun bir IDE kullanıysanız **Dart Analyzer** yazdığınız kodu otomatik olarak bir hata olup olmasına karşın kontrol eder. Komut satırından da **flutter analyze** komutu ile çalıştırılabilir. **Dart Analyzer** kullanımı ile daha geniş bilgiyi aşağıdaki bağlantılardan incelenebilir.

<https://github.com/flutter/flutter/wiki/Using-the-Dart-analyzer>

Bir başka hata ayıklama yaklaşımı log kaydı almaktır. Log alma işlevi programlı bir şekilde başlatılıp sonuçlar DartDevTools'un Log görünümünden incelenebilir.

Flutter uygulamalarında en zor hata ayıklama animasyonlardaki hata ayıklama sürecidir. Flutter Inspector ekranında **SlowAnimation** butonu ile animasyon yavaşlatılabilir.

Uygulamanın başlangıç süresine hangi adımın ne kadar etki ettiğini görmek için **trace-startup** parametresi **profile mode** ile kullanılır.

```
flutter run --trace-startup -profile
```

Çalıştırma adımda her bir seviyede harcanan süreleri **start_up_info.json** adında bir dosyaya JSON formatında yazar. Örnek bir çıktı aşağıdaki gibidir.

```
{
  "engineEnterTimestampMicros": 96025565262,
  "timeToFirstFrameMicros": 2171978,
  "timeToFrameworkInitMicros": 514585,
  "timeAfterFrameworkInitMicros": 1657393
}
```

Dart kodundaki performansa etki eden kısımları net görmek için **DevTools** içerisindeki **Timeline** görünümü kullanılır.

Flutter Inspector ekranında **Performance Overlay** butonuna tıklanırsa uygulamanın çalışması esnasında harcadığı CPU bilgisi anlık olarak uygulama ekranında görüntülenir.

13.2. Flutter'da Programlanmış Şekilde Hata Ayıklama

13.2.1. Kayıt Tutma

Bir program içerisinde akışı takip etmek için efektif olmaya da en basit yöntem log kayıtları almaktır. Log kayıtlarını **DevTools** içerisindeki **Logging** görünümünden inceleyebilirsiniz. Log almak için basitçe **dart:io** kütüphanesindeki iki temel çıkış olan **stdout** veya **stderr** üzerine çıktı gönderen metotları çağırırız. Veya bu işlemi kolaylaştırın **print()** metodu da aynı işlevi yerine getirmektedir.

```
stderr.writeln('print me');
```

Eğer **print()** metodu ile çok fazla çıktı üretiyorsanız Android işletim sistemi bunlardan bazılarını performans açısından görmezden gelebilir. Bu gibi durumlarda çıktıının olmasını garantilemek için **debugPrint()** kullanmayı tercih ediniz.

Diğer bir kayıt alma alternatifisi ise **dart:developer** paketi içerisindeki **log()** fonksiyonunun kullanılmasıdır. Bu fonksiyon çıktıyı alt parametrelerle ayırarak daha parçalı vermeyi sağlamaktadır.

```
import 'dart:developer' as developer;

void main() {
  developer.log('kayıt 1, name: 'my.app.cat1');
  developer.log('kayıt 2', name: 'my.app.cat2');
  developer.log('kayıt 3', name: 'my.app.cat2');
}
```

Uygulama içerisinde üretilmiş bir obje de JSON olarak hata kayıtlarına gönderilebilir. Bunun için **log()** fonksiyonunun **error:** parametresi kullanılır. Bu şekilde iletlenen bir hata objesi **DevTools Logging**

görünümünde detaylı bir şekilde çözümlenerek gösterilir.

```
import 'dart:convert';
import 'dart:developer' as developer;

void main() {
  var myCustomObject = ...;

  developer.log(
    'kayıt 1',
    name: 'my.app.cat1',
    error: jsonEncode(myCustomObject),
  );
}
```

13.2.2. Durma Noktaları (Breakpoint) Kullanımı

Hata ayıklamak için kullanılan diğer bir yaklaşım kodun içerisinde durma noktaları (**breakpoint**) eklemektir. Tüm IDEler **Breakpoint** kullanımını desteklerler ve ilgili **Debugger** (Hata Ayıklayıcı) için **breakpoint** tanımlaması yapabilirler. Flutter projelerinde de IDE tabanlı **breakpoint** ekleyerek hata ayıklayıcı uygulamayı inceleme yapmak için anlık olarak kod akışını kesebiliriz. Bu gerçekleştirebileceğimiz diğer bir alternatif ise **dart:developer** paketi aracılığı ile **debugger()** ifadesi eklemektir.

```
import 'dart:developer';

void someFunction(double offset) {
  debugger(when: offset > 30.0);
  // ...
}
```

debugger() ifadesi aracılığı ile sadece belli bir koşul gerçekleştiğinde kod akışını hata ayıklama için durdurabilir ve **DevTools** üzerinden kodu inceleyebilirsiniz. Bunun için **when:** parametresi ile koşul belirtilir. Yukarıdaki örnekte **offset** değeri **30**'dan büyük ise kod bu bölümde hata ayıklama için duracaktır.

Flutter'da görsel objelerin oluşturulma, render edilme, widget ağacındaki yerleşimlerini görme adımları için de hata ayıklama işlemleri gerçekleştirilebilmektedir. Arayüz çizme performansı ve animasyonların oynatılmasında harcanan işlemci gücü de ayrıntılı şekilde incelenebilmektedir. Her bir hata ayıklama yönteminin alt detayları için Flutter dokümantasyonunda aşağıdaki referans sayfasından ilgili konu başlığına bakabilirsiniz.

<https://docs.flutter.dev/testing/code-debugging>

13.3. Flutter Derleme Modelleri

Bir Flutter projesinin derlerken (build) 3 farklı modelde derleme işlemi gerçekleştirilebilir. Bunlar;

- **Debug:** Geliştirme yaparken genel olarak kullanılan hata ayıklama işlevlerinin açık olduğu ve **HotReload** (Uygulama çalışırken arayüz düzenlemeleri yapılabilmesi) özelliğinin kullanımında olduğu modeldir. **Assert** işlemleri yapılabilir ve programın çalışması esnasında ek servis hizmetlerinin çalışmasını sağlar. Kodun boyutu daha büyük olur ve olması gerektiğinden daha yavaş çalışır. Geliştirme aşamasında uygulama kodunun emülatör veya cihaz üzerinde çalıştırılarak test edilmesini sağlar. Kod optimize edilmediği için

uygulama gerçek performansını göstermeden çalışır. Sadece geliştirme süreci için tercih edilmelidir. Varsayılan olarak **flutter run** komutu Debug modelinde derleme işlemi yapmaktadır.

- **Release:** Uygulamanın dağıtım için hazırlanacak sürümünün derlendiği modeldir. Uygulamanın kullanıma alınacağı zaman ilgili kurulum paketlerinin bu model kullanarak derlenmesi gereklidir. Bu sayede içerisinde geliştirme aşaması için kullanılan servis eklentileri ve log kayıt girdileri gibi birçok unsur derleme dışı bırakılır ve uygulama performansı en iyi hale getirilir. Ayrıca bu uygulama boyutunun olabilecek en küçük seviyeye getirilmesini sağlar.

```
flutter run --release
```

komutu ile uygulama dağıtım modelinde derlenebilir. Ayrıca

```
flutter build <target>
```

komutunda <target> kısmına uygulamanın dağıtımları için kullanılacak platform bilgisi verilerek de dağıtım platformu için derleme işlemi yapılabilir. Geliştirme ortamları bünyelerinde bu model ile derlemeyi desteklerler ve üst Run menüsü altında erişilebilir.

- **Profile:** Uygulamanın performansını incelemek istediğinizde derleneceği modeldir. Sadece performans izlemesi için kullanılan bazı servis eklentileri açıktır. Bunun dışındaki hata ayıklama için eklenen kod parçaları derleme işlemine katılmaz. Bu sayede uygulama release modeldekine yakın bir performans ile çalışabilir. Profile modelinde derleme işlemi için

```
flutter run --profile
```

komutu kullanılır.

Flutter derleme modelleri ile ilgili daha kapsamlı bir wiki sayfası aşağıdaki GitHub reposunda paylaşılmaktadır.

<https://github.com/flutter/flutter/wiki/Flutter%27s-modes>

13.4. Flutter Uygulamalarının Test Edilmesi

Bir uygulamanın boyutları büyükçe el ile test edilmesi zorlaşır. Bu durumda otomatik test süreçlerinin kullanımı hem iş yükünü azaltır hem de daha az kodlama hatası (bug) içeren uygulamanın üretimesini sağlar. Otomatik test süreçleri üç ana başlıkta incelenmektedir.

- **Unit Test:** Fonksiyon, sınıf tabanlı en alt seviye test sürecidir.
- **Widget Test:** Bir arayüz bileşeninin (Widget) test sürecidir.
- **Integration Test:** Entegrasyon testi uygulamanın bütününe ya da belli bir bölümünün topluca test edilmesidir.

13.4.1. Unit Test

Unit test kavramı özellikle her bir fonksiyon veya sınıfın kendi içerisinde tutarlı davranışını ortaya koymak için iyi bir yaklaşımındır. Özellikle kod karmaşası arattıktan sonra bir fonksiyonda yapılan değişiklik birden fazla bölümün yanlış çalışmasına neden olabilir. Bu gibi durumların önüne geçilmesi için her bir fonksiyon veya sınıfın belirlenen senaryolarla düzenli test edilmesi gereklidir. Eğer ilgili fonksiyon/sınıfa yeni bir

işlevsellik eklenecekse bunun önceki test senaryolarına uyması ve gerektiriyorsa yeni senaryoların eklenmesi gereklidir. Böylece değişiklik yapıldığında uygulamanın farklı alanlarında oluşabilecek hataların en başta önüne geçilmiş olunur.

Flutter'da unit test gerçekleştirmek için **test** paketi kullanılır. **flutter_test** paketi de widgetlara yönelik test özellikleri için kullanılmaktadır.

Şimdi Flutter Cookbook sayfasında yer alan adım adım unit test oluşturma senaryosunu ele alacağız. Anlatımın geniş versiyonunu ve tam örneği

<https://docs.flutter.dev/cookbook/testing/unit/introduction>

sayfasından inceleyebilirsiniz. Unit test konusunda daha detaylı bilgi için aşağıdaki bağlantıdan test paketinin dokümantasyon sayfasına bakınız.

<https://pub.dev/packages/test>

Unit test uygulanması için öncelikle ilgili paketlerin **test** ve **flutter_test**, pubspec.yaml dosyasına geliştirme gereksinimi (dev_dependency) olarak eklenmesi gereklidir.

```
dev_dependencies:  
  test: <son_surum>
```

Sonrasında test işlemleri için bir **.dart** dosyası oluşturulur. Test işlemleri ile ilgili **.dart** dosyaları projenin **Test** klasörü altında bulunurlar. Normal kod dosyalarının **Lib** klasöründe saklandığını anımsayalım. **Test** klasöründeki **.dart** dosyaları genel olarak **Lib** klasöründeki ilgili olduğu kod dosyasının adının sonuna **_test** son eki eklenecek oluşturulur. Örnek test dosyası aşağıdakine benzer bir hiyerarşide oluşturulur.

```
counter_app/  
  lib/  
    counter.dart  
  test/  
    counter_test.dart
```

Aşağıda unit test işleminin üzerine uygulanacağı örnek bir sınıf (**Counter / counter.dart**) oluşturulmaktadır. Bu sınıfın iki adet fonksiyonu bulunmaktadır.

```
class Counter {  
  int value = 0;  
  
  void increment() => value++;  
  
  void decrement() => value--;  
}
```

Şimdi Test klasörü altında bu sınıftaki fonksiyonların testi için **counter_test.dart** dosyası hazırlanır. Test paketinde **unit test** için **expect** fonksiyonu kullanılır. Bu fonksiyon ilgili fonksiyonun beklenen sonucu üretip üretmediğini kontrol ederek test senaryosunun başarılı veya başarısız olduğunu raporlar. Buradaki örnek senaryoda yeni bir **Counter** objesi oluşturulduktan sonra **increment()** fonksiyonu bir kere çağrılırsa **value**'da saklanan değerin 1 olması gereği **expect** ile sorgulanmaktadır. Eğer ilgili satırda **counter.value** değeri 1'den farklı ise test başarısız olur.

```
import 'package:test/test.dart';
import 'package:counter_app/counter.dart';

void main() {
  test('Counter value should be incremented', () {
    final counter = Counter();

    counter.increment();

    expect(counter.value, 1);
  });
}
```

Eğer birden fazla test senaryosunun test sonuçları ekranında bir başlık altında toplanmasını istiyorsanız **test** paketiyle gelen **group** fonksiyonunu kullanabilirsiniz.

```
import 'package:test/test.dart';
import 'package:counter_app/counter.dart';

void main() {
  group('Counter', () {
    test('value should start at 0', () {
      expect(Counter().value, 0);
    });

    test('value should be incremented', () {
      final counter = Counter();
      counter.increment();
      expect(counter.value, 1);
    });

    test('value should be decremented', () {
      final counter = Counter();
      counter.decrement();
      expect(counter.value, -1);
    });
  });
}
```

Yukarıdaki örnekte 3 adet test senaryosu Counter başlığı altında gruplanmıştır. Bu test sonuçlarının daha derli toplu şekilde incelenmesine yardımcı olur.

Test senaryolarının çalıştırılması için counter_test.dart dosyası açıkken IDE'nin Run Menüsü altından dosya çalıştırılarak test senaryoları işletilebilir. Konsoldan aşağıdaki komutun çalıştırılması da aynı neticeyi verecektir.

```
flutter test test/counter_test.dart
```

13.4.2. Mockito Paketi ile Veri Bağımlı Testleri Gerçekleme

Bazen test etmemiz gereken sınıflar uzak web servislerinden veya veritabanından gelen ve nasıl olacağını öngöremeyeceğimiz şekilde olmak zorunda olabilir. Bu şekilde bir veri kaynağı kullanmak uygun olmayacaktır. Çünkü test işlemi için bu ortamlar çok yavaştır, normalde başarılı bir test veritabanı veya web servisi beklenmedik cevaplar dönerse başarısız olur ve bir veritabanı veya uzak servis ile tüm olası başarılı ve başarısız senaryoları test etmek imkansızdır. Bu sebeple web servisi veya veritabanı gibi canlı bir veri kaynağına bağımlı kalmaktansa bunların bir taklidi “**mock**” ile işlem yapmayı tercih etmeliyiz. Taklit üzerinden işlem yaparak veri değişimlerine olan bağımlılığı ortadan kaldırılmış oluruz.

Aslında yaptığımız bağımlılığı oluşturan sınıfın alternatif bir gerçeklemesini kullanırız. Bunu manuel olarak yapabileceğimiz gibi **Mockito** paketini kullanarak otomatik olarak da gerçekleyebiliriz. Şimdi Mockito paketi ile bir web servisi veri kaynağının taklit edilmesi ve bunun için unit test prosedürünün oluşturulmasını ele alalım.

İlk olarak **Mockito** paketi ile unit test uygulaması için **pubspec.yaml** dosyasındaki gerekli düzenlemeler yapılır. Aşağıdaki gibi **dependencies** ve **dev_dependencies** kısımlarına gerekli paketler tanımlanır.

```
dependencies:  
  http: <son_surum>  
dev_dependencies:  
  flutter_test:  
    sdk: flutter  
  mockito: <son_surum>  
  build_runner: <son_surum>
```

Şimdi test edilecek bir web servisi çağrı yapan fonksiyon kodlaması ele alalım. Bunun için daha önce web servisi bağlantıları için kullanılan **fetchAlbum** fonksiyonu ele alınabilir.

```
Future<Album> fetchAlbum(http.Client client) async {  
  final response = await client  
    .get(Uri.parse('https://jsonplaceholder.typicode.com/albums/1'));  
  
  if (response.statusCode == 200) {  
    return Album.fromJson(jsonDecode(response.body));  
  } else {  
    throw Exception('Failed to load album');  
  }  
}
```

Yukarıdaki kodda **fetchAlbum** fonksiyonu bir **http.Client** objesini parametre olarak almaktadır. Bu şekilde kullanımı ile gerektiğinde farklı kaynaktan üretilmiş **http.Client** objeleri fonksiyona beslenebilecektir.

Bir önceki bölümde anlatılan unit test dosyası oluşturma adımlarına uygun olarak Test klasörü altında **fetch_album_test.dart** dosyası oluşturularak içeriği aşağıdaki gibi düzeltilir.

Test senaryoları çalıştırılmadan önce **http.Client** için **Mockito** paketinin taklit üretmesi için main fonksiyonun önüne **@GenerateMocks([http.Client])** dipnotu eklenir. Böylece taklit bir **http.Client** objesi **MockClient** üretilir. Oluşturulan bu taklit (**MockClient**) **fetch_album_test.mocks.dart** dosyasında saklanır. Bu dosyanın üretilmesi için

```
flutter pub run build_runner build
```

komutu çalıştırılır. Bu aşamadan sonra test dosyasındaki test fonksiyonları yazılır. **fetchAlbum** fonksiyonunun test senaryolarını iki ana grupta inceleyebiliriz.

1. Başarılı olma durumunda bir **Album** objesi döner.
2. http çağrısının başarısız olması durumunda bir **Exception** fırlatır.

Bu iki durumda fonksiyonun davranışını test etmek için oluşturduğumuz **MockClient**'da nasıl davranışacağını **Mockito** paketindeki **when()** fonksiyonunda belirterek test senaryoları yazılır.

```
import 'package:flutter_test/flutter_test.dart';
import 'package:http/http.dart' as http;
import 'package:mockito/annotations.dart';
import 'package:mockito/mockito.dart';
import 'package:mocking/main.dart';
import 'fetch_album_test.mocks.dart';

@GenerateMocks([http.Client])
void main() {
  group('fetchAlbum', () {
    test('http çağrıı başarılıysa Album objesi dönüyor', () async {
      final client = MockClient();
      when(client
          .get(Uri.parse('https://jsonplaceholder.typicode.com/albums/1')))
          .thenAnswer((_) async =>
      http.Response('{ "userId": 1, "id": 2, "title": "mock"}', 200));
      expect(await fetchAlbum(client), isA<Album>());
    });

    test('http çağrıı başarısız ise bir exception fırlatılır', () {
      final client = MockClient();
      when(client
          .get(Uri.parse('https://jsonplaceholder.typicode.com/albums/1')))
          .thenAnswer((_) async => http.Response('Not Found', 404));
      expect(fetchAlbum(client), throwsException);
    });
  });
}
```

Burada **when()** fonksiyonu içerisinde **MockClient**'ın anlık olarak davranışını belirlenip daha sonra **except()** fonksiyonunda bu taklit sınıf ile fonksiyon testi gerçekleştirilmiş olmaktadır. Örneğin tamamı Flutter dokümantasyonunda aşağıdaki bağlantıdan incelenebilir.

<https://docs.flutter.dev/cookbook/testing/unit/mock>

13.4.3. Widget Test

Unit test uygulamaları bellek üzerindeki değişken içerikleri ve fonksiyon çağrıları için uygundur. Fakat grafik arayüzüne sahip bir uygulamada grafik arayüzünün tutarlığını nasıl kontrol edebiliriz? Özellikle Flutter gibi arayüzü dinamik çizen bir yaklaşımında bu daha kritik öneme sahiptir. **flutter_test** paketi Widget ağacının oluşumunun test edebilmek için bazı ek araçlar sunmaktadır.

- **WidgetTester** sınıfı test ortamındaki Widgetlar ile iletişim kurmayı sağlar.
- **testWidget()** fonksiyonu widget testi için **test()** fonksiyonu yerine widgetlara erişimde kullanılacak yeni bir **WidgetTester** sınıfı oluşturmaktadır.
- **Finder** sınıfı ile de test ortamında ilgili widgetin olup olmadığı aranmaktadır.

· **Matcher** sabiti kullanılarak **Finder** ile bulunan widgetların doğruluğu teyit edilir.

Bu yaklaşımın nasıl kullanıldığını Flutter Cookbookta yer alan örnek üzerinden inceleyeceğiz.

<https://docs.flutter.dev/cookbook/testing/widget/introduction>

Öncelikle **flutter_test** paketi için gereksinimler düzenlenir.

```
dev_dependencies:  
  flutter_test:  
    sdk: flutter
```

title ve **message** adında iki değişkende saklanan verileri ekrana getiren bir widget (**MyWidget**) olsun.

```
class MyWidget extends StatelessWidget {  
  const MyWidget({  
    Key? key,  
    required this.title,  
    required this.message,  
  }) : super(key: key);  
  
  final String title;  
  final String message;  
  
  @override  
  Widget build(BuildContext context) {  
    return MaterialApp(  
      title: 'Flutter Demo',  
      home: Scaffold(  
        appBar: AppBar(  
          title: Text(title),  
        ),  
        body: Center(  
          child: Text(message),  
        ),  
      ),  
    );  
  }  
}
```

Şimdi arayüzde kullanılacak **MyWidget**'ı test edecek senaryoların **flutter_test** paketindeki **testWidgets()** fonksiyonu ile test edilmesi için test kodlamasının şablonunu oluşturacak kod yazılmalıdır. **testWidgets()** fonksiyonu bize arayüz bileşenleri test süreçleri için kullanacağımız **WidgetTester** sınıfını verecektir. Yazılacak test senaryosunda bir **MyWidget** arayüz bileşeninin istenilen title ve message bilgisini ekranda gösterecek şekilde oluşturulduğu test edilecektir.

```
void main() {
  testWidgets('MyWidget title ve message içerir',
    (WidgetTester tester) async {
      // Test kodlaması burada yazılacaktır.
    });
}
```

Test ortamında **MyWidget**'ın oluşturulması için **WidgetTester** sınıfının **pumpWidget()** fonksiyonu kullanılır. **pumpWidget()** fonksiyonu test ortamında **MyWidget** arayüz bileşeninin render edilmesini sağlar. Örnekte **MyWidget**'ın title için 'T' message için 'M' değerlerini yazdırın bir test senaryosu eklenmektedir.

```
void main() {
  testWidgets('MyWidget title ve message içerir',
    (WidgetTester tester) async {
      // tester aracılığı ile MyWidget oluşturulur.
      await tester.pumpWidget(const MyWidget(title: 'T', message: 'M'));
    });
}
```

pumpWidget() çağrılarından sonra **WidgetTester** aynı arayüz bileşeninin oluşturulması için ek yaklaşımlar sunmaktadır. Bu kullanım şekli **StatefulWidget** ve **animasyonların** test edilmesinde faydalıdır. Örneğin bir butonun tıklama işlevinde **setState()** kullanıyorsanız Flutter arayüzü günceller ama test ortamında bu otomatik tetikleme gerçekleşmez. Bunun için aşağıdaki fonksiyon kullanılabilir.

```
tester.pump(Duration duration)
```

Bu fonksiyon çağrısı yeni bir ekran çerçevesi çizimini çizelgelemekte (ekrana çizmek için sıraya almakta) ve Widget'in yeniden çizimini tetiklemektedir. Burada **Duration** parametresi verilirse yeniden çizimin çizelgelemesinde bu değere göre davranışır.

```
tester.pumpAndSettle()
```

pumpAndSettle() fonksiyonu ise ekranda çizilecek yeni bir ekran çerçevesi kalmayana kadar **pump()** fonksiyonu çağrılarını tekrarlamaktadır. Animasyonlardaki sahnelerin çizimi için uygun bir kullanım sunar.

Finder ile Widget ağacı içerisinde belirlenen kriterlere uygun bir widgetin olup olmadığı test edilebilir. Örnekteki title ve message değerlerinin widget ağacında bulunduğu bir widget var ise ekrandaki çizim teyit edilmiş olur. **flutter_test** paketi içerisindeki **Finder** sınıfı bize widget ağacı içerisindeki aradığımız widgeti bulmak için **find()** metodunu sunmaktadır. Eğer Text bir widget aranıysa **find.text()** metodu kullanılır. **Finder** kullanımı ile daha geniş bir örnek aşağıdaki bağlantıda verilen Flutter Cookbook sayfasından incelenebilir.

<https://docs.flutter.dev/cookbook/testing/widget/finders>

```
void main() {
  testWidgets('MyWidget title ve message içerir', (WidgetTester tester) async
{
  await tester.pumpWidget(const MyWidget(title: 'T', message: 'M'));

  // Finderlar oluşturuluyor
  final titleFinder = find.text('T');
  final messageFinder = find.text('M');
});
}
```

Son aşama **flutter_test** paketi içerisindeki **Matcher** sınıfının sunduğu yaklaşımları kullanarak **Finder** ile widget ağaç üzerinde nasıl bir eşleştirme yaparak test uygulanacağını söylemektedir. Örneğin belirtilen özelliklerde sadece bir tane widgetin widget ağacında olması istenildiğinde **findsOneWidget Matcherı** kullanılacaktır.

```
void main() {
  testWidgets('MyWidget title ve message içerir ', (WidgetTester tester) async
{
  await tester.pumpWidget(const MyWidget(title: 'T', message: 'M'));
  final titleFinder = find.text('T');
  final messageFinder = find.text('M');

  expect(titleFinder, findsOneWidget);
  expect(messageFinder, findsOneWidget);
});
}
```

Flutter_test paketinde genel arama şekilleri için sunulan diğer **Matcher** türleri;

- **findsNothing:** Hiçbir widgetin bulunmadığını teyit için
- **findsWidgets:** Bir veya birden fazla widgetin varlığını teyit için
- **findsNWidgets:** N adet widgetin varlığını teyit için
- **matchesGoldenFile:** widgetin render işlemi sonucu çıktısının belirtilen bitmap dosyası ile eşleşmesini teyit eder.

13.4.4. Integration Test

Unit Test ve Widget Test yaklaşımları ayrı ayrı bileşenlerin test edilmesi için güzel yaklaşımlardır. Ama bileşenlerin bir arada birbiri ile etkileşimleri ile genelde ilgilenmezler. Benzer şekilde uygulamanın performansı ile ilgili bir kaygı yürütülmeden yani gerçek bir ortamda davranışını incelemeden yapılan testlerdir. Uygulama bileşenlerinin bir bütün olarak test edilmesi için **integration_test** paketi ile entegrasyon testi uygulanmaktadır.

integration_test paketi kullanımı için temel işlevsellik aşağıdaki bağlantıda verilen Fltter Cookbook sayfasından incelenebilir. Bu örnekte Unit Test ve Widget Test işlemlerinde kullanılan yaklaşımların bir senaryo üzerinden uygulama bütününe uygulanması için bir test senaryosu yazımı görülmektedir.

<https://docs.flutter.dev/cookbook/testing/integration/introduction>

Entegrasyon testleri ile ilgili diğer bir sıkıntı ise aynı uygulamanın farklı cihazlarda farklı tepkiler verebilmeye durumunu görme gereksinimidir. Bu gibi durumlarda geliştiricinin elle uyguladığı test senaryolarını her bir cihaz konfigürasyonu için tekrarlaması yorucu bir süreç olacaktır. Daha iyi bir yaklaşım olarak bu işlemin otomatik hale getirilmesi ve bir servis tarafından düzenli olarak yapılmasıdır. Bu gibi test hizmetleri veren çok sayıda oluşum bulunmaktadır. Bunlar genel olarak bulut tabanlı test sunucusu ve bu sunucu ile sürekli geliştirme/ sürekli entegrasyon mantığında çalışma olanağı vermektedirler.

Google Firebase servislerinde uygulama entegrasyon testleri için de bir çözüm eklemiş durumdadır. **Firebase TestLab** sayesinde farklı cihaz için test işlemlerini otomatik olarak yürütebilirsiniz. Aşağıdaki bağlantı ile **Firebase TestLab** dokümantasyonuna erişebilirsiniz.

<https://firebase.google.com/docs/test-lab>

Flutter dokümantasyonunda entegrasyon testleri için kullanılan teknikler ve yönlendirmeler için aşağıdaki bağlantıyı ziyaret ediniz.

<https://docs.flutter.dev/testing/integration-tests>

Bölüm Özeti

Flutter'da yazılan kodların ve arayüzün test edilmesi ve hataların giderilmesi için birçok araçlar sunulmaktadır. Hata ayıklama sürecinde kullanılan araçlar DartDevTools altında toplanmıştır. Flutter Test-Driven Development (Test-Oaklı Geliştirme) yaklaşımını desteklemektedir. Unit-Test, Arayüz Bileşen testi ve entegrasyon testi için hazır paketler bulunmaktadır. CI/CD prensibi ile çevrimiçi bir hizmet ile flutter projeleri de sürekli olarak entegrasyon testlerine tabi tutulabilir. Firebase TestLab bunun için bir alternatifidir.

Kaynakça

<https://docs.flutter.dev/testing/debugging>

<https://docs.flutter.dev/testing/build-modes>

<https://docs.flutter.dev/development/tools/devtools/overview>

<https://docs.flutter.dev/development/tools/devtools/inspector>

<https://github.com/flutter/flutter/wiki/Using-the-Dart-analyzer>

<https://docs.flutter.dev/testing/code-debugging>

<https://github.com/flutter/flutter/wiki/Flutter%27s-modes>

<https://docs.flutter.dev/cookbook/testing/unit/introduction>

<https://pub.dev/packages/test>

<https://docs.flutter.dev/cookbook/testing/unit/mockling>

<https://docs.flutter.dev/cookbook/testing/widget/introduction>

<https://docs.flutter.dev/cookbook/testing/widget/finders>

<https://firebase.google.com/docs/test-lab>

<https://docs.flutter.dev/testing/integration-tests>

14. PERFORMANS, OPTİMİZASYON VE DAĞITIM

Giriş

Performans iyileştirmek ve uygulama dağıtımları oluşturmak yazılım geliştirme sürecinin en son aşamalarıdır. Bu aşamaya gelen uygulama aslında kullanıcı deneyimine sunulmaya uygundur fakat kötü izlenim, kodun çalınması gibi unsurlar daima uygulama canlıya alınmadan önce düşünülmelidir. Bu bölümde uygulamanın canlıya alınmasına hazır hale getirilmesi için iyileştirme uygulamalarını ele alacağız ve uygulamamız dağıtıma hazır hale getirilecektir.

14.1. Performans Temelleri

Flutter'da performans iyileştirmesi için anlatıma başlamadan önce performansın ne olduğu nasıl ölçüleceği konularını göz önüne almak gerekiyor. Mobil cihazların ihtiyaçları düşünüldüğünde Flutter'da performans metrikleri; hız, bellek kullanımı, uygulama boyutu ve enerji kullanımı şeklinde gruplanmaktadır.

14.2. Uygulama Boyutunun Ölçülmesi

Flutter uygulamalarını Android ve iOS sürümleri için uygulama paketleri haline getireceğimiz zaman paket içerisinde uygulamanın kodlarının yanında Flutter'ın çalışması için gerekli sarıcı kodlar, resim ve müzik gibi varlıklar(assets) da uygulama paketinin içerisine eklenecektir. Bu durumda olacak APK, app bundle, IPA paket yapılarının boyutu dikkat edilmesi gereken bir metriğe dönüşmektedir. Zira büyük paket boyutları uygulamanın daha uzun sürede kurulması daha geç yüklenmesi ve cihaz belleğinde daha fazla yer kaplamasına sebebiyet verir.

Uygulama boyutu ölçmek için doğru ölçme yaklaşımı kullanmamız gereklidir. Bu açıdan yapılacak ilk yanlış hata ayıklama (**Debug**) kipinde elde edilen verilerin kullanımıdır. Terminalde **flutter run** komutu girdiğinizde veya IDE'den uygulamayı doğrudan çalıştırığınızda uygulama Debug kipinde çalışacaktır. Bu kipte hata ayıklama için ek bilgiler ile derleneceğinden uygulama boyutu normalde olması gerekenden kat be kat fazla çıkacaktır. Debug kipinde derlenen bir uygulama dağıtım için kullanılmamalıdır.

Uygulamanın varsayılan dağıtımlarını **flutter build apk** ve **flutter build ios** komutları ile Google Play Store ve App Store için elde edilebilir. Buradan mağazalar için yükleme paket boyutları elde edilir. Bu mağazalar uygulama paketini farklı cihaz modellerine göre parçalara ayırarak küçültürler. Varlıkların arasından ilgili ekran boyutlarına göre resimlerin seçilmesi, CPU türüne göre yerel (native) kütüphane sürümlerinin seçilmesi v.b. düzenlemeler mağazalar tarafından yapılmaktadır.

Google Play Markette olacak uygulamanın asıl boyutlarını görmek için aşağıdaki bağlantıda yer almaktadır. Bu yönergeler uygulanabilir.

<https://support.google.com/googleplay/android-developer/answer/9302563?hl=en>

Apple Markette olacak uygulama boyutu ile ilgili yine detay bilgi edinilmesi için aşağıdaki bağlantıda yer almaktadır. Bu yönergeler uygulanabilir.

<https://developer.apple.com/documentation/xcode/reducing-your-app-s-size#3458589>

14.3. Uygulama Boyutunun Azaltılması

DevTools içerisinde uygulama dağıtım boyutunun tespit edilmesi için bir analiz aracı ile gelmektedir. Bu analiz aracını dağıtım sürümünü oluştururken **--analyze-size** parametresi verilerek aktifleştirilir. Aşağıda farklı platformlar için ilgili komutun kullanım örnekleri verilmektedir.

- flutter build apk --analyze-size
- flutter build appbundle --analyze-size
- flutter build ios --analyze-size
- flutter build linux --analyze-size
- flutter build macos --analyze-size
- flutter build windows --analyze-size

Analiz aracı terminalde uygulama paketi içeriğinin boyut dağılımını özet bir gösterimde vermektedir ayrıca ürettiği ***-code-size-analysis_*.json** dosyası içerisinde boyut dağılımı ile ilgili daha detaylı bilgi sunmaktadır. Bu json dosyasının incelenmesi ile ilgili bilgiler aşağıdaki bağlantıda anlatılmaktadır.

<https://docs.flutter.dev/development/tools/devtools/app-size>

Terminal ekranında alınan özet dökümü uygulamanın boyutunun şişiren bölümün genel hatları ile görüntülenmesini kolaylaştırmaktadır. Kullanılan paketleri ve varlıklarını burada boyutları ile görebilir ve beklediğinizden daha büyük bir değer olması durumunda detaylı inceleme için json dosyasının incelenmesi adımına bakabilirsiniz. Flutter projeleri yerel dillerle (native) projelere göre daha büyük olacaktır. Bu Flutter’ın kullandığı sarmal (wrapper) kodlarından kaynaklanmaktadır. Aşağıda örnek bir terminal görüntüsü verilmektedir.

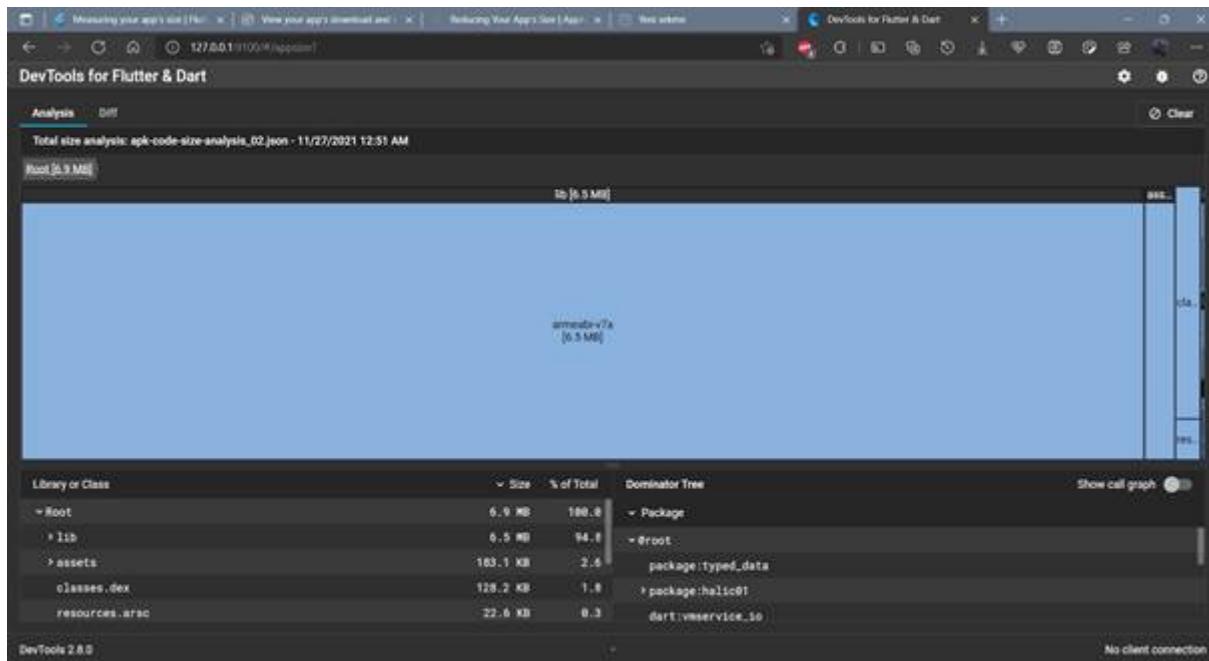
```
PowerShell 7 (x64)
$ Built build\app\outputs\flutter-apk\app-release.apk (5.0M).

app-release.apk (total compressed) 5 MB

res/
    mipmap-xxxhdpi-v4 1 KB
META-INF/
    MANIFEST.MF 2 KB
    CERT.SF 3 KB
    kotlin-stdlib.kotlin_module 1 KB
    CERT.RSA 1011 B
assets/
    flutter_assets 183 KB
kotlin/
    reflect 2 KB
    collections 1 KB
    kotlin.kotlin_builtins 4 KB
resources.arsc 23 KB
lib/
    armeabi-v7a 5 MB
    Dart AOT symbols accounted decompressed size 3 MB
        package:flutter 2 MB
        dart:core 323 KB
        dart:typed_data 240 KB
        dart:ui 195 KB
        dart:async 102 KB
        dart:collection 94 KB
        dart:convert 65 KB
        dart:io 41 KB
        dart:isolate 36 KB
        package:vector_math 29 KB
        dart:developer 9 KB
        dart:ffi 7 KB
        package:typed_data/
            src/
                typed_buffer.dart 6 KB
        package:collection/
            src/
                priority_queue.dart 6 KB
        package:halic01/
            main.dart 2 KB
        dart:math 2 KB
        dart:vmservice_io 2 KB
        dart:mirrors 604 B
        dart:nativewrappers 251 B
        Never 52 B
    classes.dex 128 KB

A summary of your APK analysis can be found at: C:\Users\crane\.flutter-devtools\apk-code-size-analysis_02.json
```

Oluşturulan json dosyası DevTools aracılığı ile daha detaylı incelenmek için DartDevTools'u açın. (VSCode içerisinde sağ alt köşedeki durum çubuğu üzerinden çalıştırabildiğinizi anımsayın.) Sevdığınız internet tarayıcısı içerisinde DartDevTools açıldıktan sonra ana ekranında “App Size Tooling” başlığı altındaki “Open app size tool” butonu aracılığı ile kaydedilen json dosyasını göstererek aşağıdaki gibi her bir paketin ve kod dosyanızın dağıtım paketindeki kapladığı alanı görebilirsiniz.



Örnek projemde sadece Flutter’ın verdiği şablon projenin incelemesini görmekteyiz. Bu projenin analizinde 6.9MB toplam uygulama boyutundan 6.5MB’ı derleme için kullanılan android-arm platformu için kullanılan temek kütüphanelerdir. Bu durum azaltamayacağımız minimum Flutter uygulama boyutunu göstermektedir.

Uygulamanın dağıtım sürümünün boyutunu azaltmak için uygulanabilecek işlemler;

1. Dağıtım sürümü üretilirken --split-debug-info etiketinin kullanımı; Nasıl kullanılacağı ile ilgili anlatımı aşağıdaki bağlantıda bulabilirsiniz.

<https://docs.flutter.dev/deployment/obfuscate>

2. Kullanılmayan kaynakların projeden çıkarılmasını sağlamak.

3. Kütüphanelerden yüklenen kaynakların en aza indirilmesini sağlamak.

4. PNG ve JPEG dosyalarını sıkıştırarak kullanmak.

14.4. Uygulamanın Render Performansının İyileştirilmesi

Flutter, Skia grafik motoru kullanması sayesinde arayüz bileşenlerinin hazırlanıp ekrana getirilmesinde oldukça iyi bir performans sergilemektedir. Normal kullanımda herhangi bir iyileştirmeye ihtiyaç duyulmayacak şekilde Flutter arayüz bileşenleri optimize edilmektedir. Sadece bazı hatalı yaklaşımlardan sakınmak gerekmektedir. Arayüz hazırlama (render) performansından bahsedince en kritik konu başlığını animasyonlar almaktadır. DartDevTools içerisinde render performansını incelemek için araçlar mevcuttur. Şimdi bu konu başlıklarına kısaca değineceğiz ve daha ileri düzey işlemler için ilgili referansları vereceğiz.

Animasyonların düzgün olmadığını veya sıçramalar yaptığını görüporsanız bunu dert edinmeden önce uygulamanızın release veya profile modelinde derlendiğinden emin olunuz. Çünkü debug modelin derlenen bir uygulama gerçek render performansını göstermeyecektir.

İster **StatelessWidget**, ister **StatefulWidget** kullanın arayüz bileşeninizin ekrana çizimi için **build()** metodu üzerinden geçilmektedir. **build()** metodu üzerinden geçme işlemi genellikle üst seviye bir widget'in kendi **build()** metodunu çağrımasıyla olmaktadır. Yani **build()** metodları iç içe bir zincir şeklinde

çağrılmaktadır. Arayüzün çizilmesi ancak widget ağacındaki tüm **build()** metodlarının çağrısının tamamlanması ile mümkündür. Bu nedenle **build()** metodu içerisinde arayüzün gelmesini geciktirecek bir kod olmamalıdır.

Geniş ve tek bir widget kullanımından kaçınmak gerekir bunun yerine uygun hiyerarşide ekranı bölen daha az genişlikte widgetlar tercih edilmelidir. Özellikle `setState()` metodunun kullanıldığı yerin widget ağacı içerisinde hiyerarşide olabilecek en alt seviyede olması gerekir. Bir diğer hususta `setState()` ile ekranda bir güncelleme yapılacağıda eğer widget ağacının daha alt kısımları değişimmeyecekse bunların **builder** yaklaşımı ile var olan çizimi kullanması sağlanmalıdır. Animasyonlar konusunda buna kısmen degenmişti. Animasyonlarda bu yaklaşımın kullanımı hakkında ek bilgiyi aşağıdaki bağlantılardan inceleyebilirsiniz.

<https://api.flutter.dev/flutter/widgets/TransitionBuilder.html>

<https://github.com/flutter/flutter/blob/master/packages/flutter/lib/src/widgets/transitions.dart>

Ayrıca **StatefulWidget** kullanımında performansın dikkate alınması ile ilgili Flutter dokümanını da aşağıdaki bağlantından inceleyebilirsiniz.

<https://api.flutter.dev/flutter/widgets/StatefulWidget-class.html#performance-considerations>

Arayüz hazırlanması ile ilgili diğer bazı başlıklarları şöyle sıralayabiliriz;

- Sadece gerektiğinde özel efekt kullanılmalıdır. Mesela bir resmin opaklılığı değiştirilecekse **Opacity** widget yerine görüntü işleme yaklaşımları daha performanslı sonuç sergilemektedir. Buradaki temel neden bu tarz efektlerini `saveLayer()` fonksiyonunu arkada tetiklemesi ve ekranları görüntü olmadan önce hazır etmesidir. Bu da ciddi bir işlemci kullanımı gerektirir. Buna benzer `saveLayer()` fonksiyonunu kullanan diğer bazı widgetlar; `ShaderMask`, `ColorFilter`, `Chip`, `Text` şeklindedir.
- Grid ve List arayüz bileşenlerinde gelen verinin **Lazily** (sadece gerektiğinde) hazırlanmasını sağlamak (**Lazy Loading**) ekranda görüntülenmeyecek bileşenlerin hazırlanmasını engeller ve ekran kaydırıldıkça ekran dışı kalan bileşenler tekrardan kullanılır. Bu özellikle veri seti büyük bir liste kullanıldığından tüm listenin render edilmesi gibi yorucu bir işlemin bir seferde değil de kaydırma işlevi olduğu sürece parçalı olarak yapılması ile anlık render performansını iyileştirir. Bu uygulama şekli ile ilgili olarak aşağıdaki dokümanlar incelenebilir.

<https://docs.flutter.dev/cookbook/lists/long-lists>

<https://medium.com/saugo360/flutter-creating-a-listview-that-loads-one-page-at-a-time-c5c91b6fabd3>

<https://api.flutter.dev/flutter/widgets/ListView/ListView.builder.html>

- Her bir **build()** metodunun 16ms altında cevap döndüğünden emin olunuz. 60hz ekran tazelemesi sağlamak için 16ms'de bir kare gösterilmesi gerekmektedir. Bunun için widget ağacında bulunan `build()` metodlarının 16ms altında işlemini tamamlaması gereklidir. En kritik olan widget ağacının üst seviyelerindeki `build()` metodlarıdır. Flutter'in Skia motorunu kullanan arayüz yapısı 120hz çalışma performansına sahiptir. Yani `build()` metodlarında bir gecikme olmadığından arayüzün vaktinde render edileceğini varsayılabilsiniz. Bu incelemeyi yapmak için uygulamanızı profile kipinde çalıştırarak “**performance overlay**” bölümünden anlık ekran tazeleme gecikmelerini inceleyebilirsiniz.

14.5. DevTools Performance View Kullanımı

Performance View, uygulamanızın içerisindeki zamanlama ve performansı ile ilgili bilgileri sunan bir araçtır. **DartDevTools**'un bir parçasıdır ve sadece mobil platformlar için performans incelemesine yardımcı olur. Eğer web platformu için bir Flutter uygulamasının performansı incelenerekse Chrome DevTools kullanılabilir.

Üç temel bileşenden oluşmaktadır.

· Flutter Frames Chart: Bir Flutter uygulamasında ekrana gelen her bir çerçeveyin ne kadar süre ile hazırlandığını bir bar grafik olarak sunmaktadır. Bu grafikte Frame için UI hazırlanma süresi ve Render işlem süresi ayrı ayrı verilmektedir. Ekranda donukluğa neden olacak çerçeveler için farklı bir renk kullanılarak bunlar işaretlenir. Grafik üzerinden bir bar seçildiğinde bu çerçevenin hazırlanmasında yapılan işlemlerin alt detayları görüntülenir.

· Timeline Events Chart: Uygulama içerisinde tetiklenen tüm olayların zamanlamasını yansıtır. Bunlar arayüz çizimi ile ilgili olanların yanında örneğin bir http sorgusunun sonuçlanması da bu çizelgeden bakılabilir.

· CPU Profiler: Belirleyeceğiniz bir başlama ve bitiş zamanı arasında işlemcide yapılan işlemleri kaydederek grafiksel gösterimini almanıza yardımcı olur. Profile işlemi için (**CallTree**, **Bottom Up** ve **FlameChart**) şeklinde gösterimleri mevcuttur.

DartDevTools içerisindeki Performance View, uygulamanın CPU tüketimi, darboğazlarının tespiti için yardımcı olmaktadır. Daha detaylı kullanım bilgisine DevTools dokümantasyonu üzerinden erişilebilir.

<https://docs.flutter.dev/development/tools/devtools/performance>

14.6. Arka Planda Kod Çalıştırma

Performans konusunda çokça dejindiğimiz bir nokta arayüzün çiziminin bekletilemeyeceğidir. Burada 16ms kritik bir sınır değeridir. Peki bir web servisinden gelecek cevap, dosya sisteminde yüklenecek bir resim veya veritabanına atacağınız bir sorgu 16ms'den daha kısa bir sürede tamamlanabilir mi? Muhtemelen cevabımız hayır olacaktır. Bu sıraladıklarımızdan bazıları saniyeler mertebesinde uzayabilmektedir. O zaman biz Flutter'da kod yazarken bu veri kaynaklarına erişimi nasıl modellemeliyiz? Bu bahsi daha önceki konularda kısmen ele almıştık. Flutter uzun sürecek I/O işlemlerinin arayüzü bekletmemesi için çeşitli mekanizmalar sunmaktadır. Bunlardan en basit olanı **Future** ve **Stream** objeleri ile çalışmaktadır. Bu yaklaşım aslında ilgili I/O çağrısını yine UI (Arayüz) thread'i içerisinde gerçekleştirmektedir fakat işlem asenkron şekilde I/O cevabı beklenmemeksinin sonraki komutların içrası ile gerçekleştirilmektedir. Flutter bunun dışında da asenkron çalışma modelleri sunmaktadır. Arka planda ayrı çalışacak threadler oluşturmak için **Isolate** kullanılır. Isolate arka planda bir iş parçacığı oluşturur ve klasik uygulamalardaki threadlerden farklı olarak ortak paylaşımı bellek kullanmazlar. Her Isolate kendi bellek alanına sahiptir ve mesajlaşma ile diğer Isolateler ile haberleşirler. Aşağıdaki 5 videodan oluşan video serisinde Flutter ve Dart için arkaplan iş yapma modellerinin çalışma prensipleri anlatılmaktadır.

Third-party media, can we show it?

This media is from an external source. It might use cookies and has its own privacy policy. Your IP may be exposed to that party if you allow.

Always for YouTube

Only this media

Settings

 youtube.com

https://youtu.be/vl_AaCgudcY

14.7. Uygulamanın Dağıtımları

Flutter ile cross-platform bir uygulama geliştirdiğimiz için uygulamanın farklı platformlar için dağıtımının da sağlanması gerekmektedir. Özellikle Android ve iOS platformlarına dağıtım yapabilmek için Google ve Apple firmasının sunduğu altyapıya yönelik bazı düzenlemelerin uygulama paketlerinin oluşturulmasında eklenmesi gereklidir (API Key gibi). Bunun yanında uygulama için masaüstü ikonu seçilmesi, açılış ekranı hazırlanması gibi yine platforma yönelik düzenlemeler yapılması gereklidir.

Flutter dağıtımının farklı platformlar için hazırlanmasına ait yönergeler;

Android: <https://docs.flutter.dev/deployment/android>

iOS: <https://docs.flutter.dev/deployment/ios>

macOS: <https://docs.flutter.dev/deployment/macos>

Linux: <https://docs.flutter.dev/deployment/linux>

Web: <https://docs.flutter.dev/deployment/web>

Bu yönergelerde Flutter uygulamasını ilgili platform marketlerinde dağıtımlı için gerekli hazırlık süreçleri adım adım verilmektedir. Bazı noktalarda Google ve Apple firmasının kendi market sayfalarına yönlendirilecek ve burada oturum açmanız istenecektir. Marketler üzerinden uygulama dağıtımlı yapmanız için harici test süreçleri ve geliştirici hesap üyeliği almanız da gerekmektedir.

Yazdığınız uygulamaların dağıtımını yapmadan önce dikkat etmeniz gereken bir husus ise tersine mühendislik ile yazdığınız kodların çalınmasının engellenmesidir. Bunun için kodların karıştırılması süreci (**Obfuscating**) uygulanır. Dart kodları için obfuscating işlemi Android, iOS ve macOS'da desteklenmektedir. Linux ve Windows dağıtımları için obfuscating henüz desteklenmemektedir. Web için ise kodların küçültülmesi süreci obfuscating işlemine benzer bir netice doğurmaktadır.

Dart kodlarının obfuscating işlemine tabi tutulması ile ilgili dokümantasyon

<https://github.com/flutter/flutter/wiki/Obfuscating-Dart-Code>

adresinde verilmektedir. Obfuscating işlemi sadece release kipinde kullanılabilir.

Bir release derlemesinin obfuscating işlemine tabi olması için derlemenin **--obfuscate** etiketinin **--split-debug-info** etiketi ile yapılması gerekmektedir. Örnek bir kullanım aşağıdaki gibi bir terminal çağrısidır.

```
flutter build apk --obfuscate --split-debug-info=<project-name>/<directory>
```

Bölüm Özeti

DartDevTools içerisinde Flutter uygulamalarının performansını ölçmek ve iyileştirmek için araçlar bulunmaktadır. Uygulama boyutunu hesaplamak ve küçütmek için boyut analizi genel ve farklı platformlar için ayrı ayrı yapılır. Arayüz çizim işlemlerinde 16ms kritik bir kısıttır. Bunun sağlanması **Performance Overlay** ile anlık olarak görülebilir. Uygulama dağıtımlı için platformların kendine özgü işlem adımları yapılmalıdır. Bunların çoğu Flutter'dan bağımsız işlemlerdir. Dağıtımda dikkat edilmesi gereken bir unsur kodlarınızın güvenliği için **obfuscating** işleminin uygulanmasıdır.

Kaynakça

<https://docs.flutter.dev/perf/app-size>

<https://support.google.com/googleplay/android-developer/answer/9302563?hl=en>

<https://developer.apple.com/documentation/xcode/reducing-your-app-s-size#3458589>

<https://docs.flutter.dev/development/tools/devtools/app-size>

<https://docs.flutter.dev/deployment/obfuscate>

<https://api.flutter.dev/flutter/widgets/TransitionBuilder.html>

<https://github.com/flutter/flutter/blob/master/packages/flutter/lib/src/widgets/transitions.dart>

<https://api.flutter.dev/flutter/widgets/StatefulWidget-class.html#performance-considerations>

<https://docs.flutter.dev/cookbook/lists/long-lists>

<https://medium.com/saugo360/flutter-creating-a-listview-that-loads-one-page-at-a-time-c5c91b6fabd3>

<https://api.flutter.dev/flutter/widgets/ListView/ListView.builder.html>

<https://docs.flutter.dev/development/tools/devtools/performance>

https://youtu.be/vl_AaCgudcY

<https://docs.flutter.dev/deployment/android>

<https://docs.flutter.dev/deployment/ios>

<https://docs.flutter.dev/deployment/macOS>

<https://docs.flutter.dev/deployment/linux>

<https://docs.flutter.dev/deployment/web>

<https://github.com/flutter/flutter/wiki/Obfuscating-Dart-Code>