**BÖLÜM1**

**Docker Nedir? :** Docker yazılım geliştiriciler ve sistemciler için açık kaynaklı yeni nesil bir sanallaştırma platformudur. Dockersız ortamda sanal bir işletim sistemine ihtiyacımız vardır docker ile artık buna gerek yok.

**DockerFile Nedir?:** İlk adımımız bizim projemizin route klasörü içerisinde oluşturacağımız bir DockerFile dosyasıdır. Bu dosya içerisinde biz alt alta satır satır talimatlar yazarız. Bu talimatlar bizim uygulamamızın nasıl bir ortamda çalışacağıyla ilgili talimatlardır. Bir işletim sistemi, bu işletim sistemindeki ilgili .net uygulamasını çalıştıracak dll’e ihtiyaç var yani kütüphaneler gerekiyor en son ise .net publish edilmiş datasına ihtiyacımız var. Productionda bir .net uygulamasının çalışabilmesi için bu 3 bileşene ihtiyaç var. İşte bu bileşenleri tanımladığımız yer burası. Bu dosyada bu bahsettiğimiz sadece 3 yapı değil ayrıca bu projenin bağımlu olduğu başka dependencyler varsa onlarıda ekleyeceğimiz yerdir ayrıca bu projenin kullanmış olduğu bir env dosyası varsa yine bu dockerfile dosyası içerisinde talimat olarak belirtebiliyoruz. Bu dosya uzantısızdır.

**DockerImage Nedir?:** Biz docker build komutunu kullandığımız zaman bir image oluşur. Bu imageımız dockerfile dosyasında belirtmiş olduğumuz talimatlara göre oluşur. DockerImage bir layer yani katmanlar topluluğudur. Dockerfile dosyası içeriisnde belirtmiş olduğumuz her bir satıra karşılık bizim dockerimage içeriisnde bir layer oluşur. diyelim 4 satırlı bir dockerfile dosyamız varsa 4 katmanlı bir dockerimage oluşur. Imagein içerisinde bizim dockerfile dosyası içerisiinde belirtmiş olduğumuz talimatlara göre bir işletim sistemi bu işletim sisteminin üzerinde sizin .net uygulamamızın çalışacağı ortam envermiont bu ortamın üzerinde .net uygulamasının publish edilmiş datası bulunur. Dockerimage dosyası statik bir dosyadır ve sadece readonlydir. Çalıştırılabilir bir dosya değildir.

**DockerContainer Nedir?:** Dockerimage dosyası dockerfile dosyası içerisinde belirtmiş olduğumuz talimatlara göre oluştuktan sonra uygulamamızı çalıştırabilmek için docker run komutuyla beraber bir container ayağa kaldırırız. Container dediğimiz dockerimage’ların çalışabilen instancelarıdır. Containerları durdurabilir silebilir veya tekrar başlatabiliriz. Ancak dockerimagelar statik olduğu için böyle işlemleri gerçekleştiremiyoruz. Dockerimage üzerinden istediğiniz kadar container ayağa kaldırabiliriz. Dockerimagei programın içindeki bir fabrika gibi düşünebiliriz. İmageiniz hazır çalışabilen bir yapı içeride var biz bu imagedan istediğimiz kadar çalışabilen instancelarını üretebiliriz yani container. Uygulamayı dockerize edebilmek için bu saydığımız 3 adımı uygulamamız gerekiyor. **Container içeriisnde local bir şey bulunamaz containeri canlı bir ortam gibi düşünmeliyiz.**

**DockerCLI Nedir?:** DockerCLI docker ile haberleşmemizi sağlayan bir komut satırıdır. Biz dockerCLI a bir komut yazdığımız zaman bu komut docker deamon’a bir rest api isteği gerçekleştirir. Bu rest  api isteği üzerinden docker deamon yapacağı işlemi gerçekleştirir.

**Docker Registry Nedir?:** Bizim imagelarımzı kaydedebileceğimiz bir depo olarak düşünebiliriz. Diyelin .net uygulaması yaptık ve bunu dockerize etmek istiyoruz. route klasöründe bir dockerfile dosyası olışturduk bu dosyada gerekli komutları yazarak bir image oluşturacaksınız yalnız şöyle bir şeye ihtiyacınız var .net uygulamasının çalışabilmesi için mini bir işletim sistemine ve bu işletim sistemi içerisinde kurulu bir .net uygulamasını çalıştıracak bir envorminet a ihtiyacınız var işte bunu kendiniz olıuşturmak yerine direkt çeşitli firmaların yayınlaşmış olduğu imagelardan direkt çekip kullanabiliyoruz. Yani kendim gidip tek tek ortamları hazırlamak yerine zaten cloudda registery hizmeti veren firmalar var bu firmalar üzerinden hazır imagelar kullanıyorum. Ancak bu hazır imagelar içerisinde herhangi bir program yüklü değil bu imagei çektikten sonra üzerine çalışacak olan programı biz koyuyoruz.

**SDK RUNTİME**: Biz sdk içerisinde uygulama inşa ederiz sdk içeriisnde aynı zamanda runtime vardır. Runtime da ise sadece uygulamanızı yayınlarsınız. Runtine uygulamanızı çalıştırmak için bir ortam sağlarken sdk hem uygulama çalıştırmak hemde geliştirmek için bir ortam sağlıyor.

**LINUX WINDOWS IMAGE**: Windows imagelar içerisinde mini bir windows işletim sistemi linuxta ise yine mini bir linux işletim sistemi var ama linux windowsa göre çok daha küçük bu yüzden çok daha hızlı dolayısıyla container işlemleri vs çok daha hızlı olur tavsiye edilen linuxtur. Ancak bu öneri crossplatform frameworkler uygulamalar için geçerli örneğin .net core değil eğer bir .net uygulamamız olsaydı bu framework crossplatform olmadığı için winddows image kullanmak zorunda kalırdık.

**DOCKER HUB**: Docker hub üzerinden hangi uygulamayı çalıştıracaksak onun uygun versiyon tagli imagesini docker hub üzerinden aratıp pull edip yani bilgisyarımıza inidirip bu hazır olan image üzerine kendi uygulamamızı koyarak yeni bir image üretiyoruz. Bu sayede docker sayesinde yaptığımız bir uygulamayı dockerize ettiğimizde image içerisinde mini işletim sistemi ve runtime olacağı için versiyon uyumsuzlukları veya kişide uygulama olmamas sorunu engellenir mesela .net uygulamasını bir javacıya göstericez javacıda .net sdk veya vs olmadığı için projemizi rar olarak açamaz illa canlıya alıp öyle göstermemiz lazım .netciye gösterirsek bu sefer .netcinin versiyonu eski olabilir yine rar olarak uygulamayı çalıştıramaz dolayısıyla biz bunu dockerize edip image oluşturursak ve bu kişilerde bu imagei pull edip kullanırsa image içerisinde runtime olacağı için uyumsuzluk veya eksiklik sorunu olmadan uygulamayı kolayca çalıştırabilecektir.

**.NET UYGULAMASINI PUBLİSH ETTİĞİMİZDE** artık bir debug modda build etmek var birde release modda build etmek var. Uygulamanın bin klasöründe bir debug klaösrü olur birde release klasörü olur. Debug modda build daha çok hata ayıklama yapar ve içerisinde aynı zamanda uygulamanın çalıştığı zamanda çıktılar verecek bir yapıda exe dosyası oluşur ama release modda build edildiği zaman uygulamnın production ortam için daha optimize edilmiş ve debug kodlardan arındılımış bir şekilde bu işlem gerçekleşir. Canlı ortama uygulamayı atarken configuration olarak release modda build işlemi gerçekleştirilir data publish edilir ve o şekilde alınır. Docker için ise publish edilmiş .net datasına ihtiyacımız var. Debug mod hata ayıklamaya yönelik bir exe oluştururken release mode daha çok production ortama yönelik oalrak optimize edilmiş özellikle projenin boyutunun daha azaltıldığı bir dosyadır. Dockerfile dosyası içeriisnde bir image oluşturabilmem için uygulamanın publish edilmiş datasını vermem gerekiyor.

**FROM KOMUTU** bizim dockerhubdan bir image almamızı sağlar. Eğer ki imageimiza bir base image yüklemek istiyorsak from komutunu kullanıyoruz. Hep hazır bir image kullanırız bu image üstüne uygulamamızı yerleştiririz.

**EXPOSE KOMUTU** containerların birbiri ile haberleşmesi için kullanılan bir komuttur. Containerların birbiriyle haberleşmesi gerektiğinde bu containerlar kendi portlarını dışarı açarlar. Mesele Expose 80 yazıyorsa başka bir container bu container ile 80 portu üzerinden haberleşebilir anlamına gelmektedir. Bu port numarasını uygulamanın çalışacağı port değil bununla karıştırmayalım sadece containerların birbiriyle haberleşmesi için tanımlanan port numaralarıdır. Mikroservis yapıda sıkça bu komut kullanılmaktadır.

**WORKDIR KOMUTU** From komutu ile imagedaki ilk katmanı yazmış olduk şimdi workdır kullanarak o katmanın üstüne ikinci katmanı koyuyoruz bu komutun anlamı çalışma klasörü yani bir klasör oluşturayım kendi projemi bu klasör içerisine atayım. Buraya mesela WORKDIR /app yazarsak image içerisinde app adında bir çalışma klasörü oluaşcaktır. Ardından bu klasör içerisine benim uygulamamın publish edilmiş datasını atmam gerekiyor.

**COPY KOMUTU** ile beraber benim bilgisayarımdaki uygulamamdan imagenin içerisine bir dosya göndericem, COPY bin/Release/net6.0/publish /app/  copyden sonraki yazdığımız dosya yolu bu dosyaları al boşluktan sonra ise imagein içinde oluşturduğumuz /app/ klasörüne bu dosyaları kopyala diyoruz.

**ENTRYPOINT KOMUTU** ise bir imageden bir container ayağa kalktığı zaman çalıaşcak olan kodu belirler. Conteinerin çalışabilmesi için benim ona bir dll vermem lazım. ENTRYPOINT ["dotnet", "UdemyConsoleDocker"] şeklinde bunu yazabiliriz ikinci verdiğimiz şey publish klasörü içindeki dll, eğer bir proje içeriisnde uygulamamızı çalıştırmak istiyrosak dll’i kullanammız lazım dockerda kendi içerisinde bu uygulamayı çalıştaracağı için dll veriyoruz. Exe ise bir program olmadan uygulamayı çalıştırmak istiyorsak kullanılır. Toplamda 4 satırlık bir kod yazdık dolayısıyla image tarafıdna 4 tane layer oluşur biz bu imagedan bir container ayağa kaldırdığımız zaman docker bu 4 layer üzerinde birde yazılabilen bir layer ekler ve 5. layer ile beraber containerimiz ayakta olacak.

**İMAGE OLUŞTURMA** dockerfile dosyam tamam artık DockerCLI kullanarak bir image oluşturabilirim. Proje dizininin terminalinde docker build -t UdemyConsoleApp . burddaki . nın anlamı dockerfile dosyası buradaki terminalde belirtilen dizinin içerisinde anlamına geliyor bu komut ile bir image oluşturabilriim. -t ise image’e isim vermemi sağlıyor.

**IMAGENİN OLUŞUP OLUŞMADIĞINI TEST ETMEK** burada yine bir dockercli kodu kullanıcaz docker images yazdığım anda bilgisayarımda kurulu olan imagelar listelenecektir.

**CONTAİNER OLUŞTURMA** image oluşturduktan sonra bu imagenin çalışan bir instancesi yani container oluşturmak için proje dizinininde terminal ekranına docker create –- name containerismi hangiimageiçinolduğunubelirtmekiçinimageismi dedikten sonra containerimiz oluşur ve terminalde bize bu containerin id si verilir. Aynı imageden istediğimiz kadar farklı isimde container oluşturabiliriz.

**CONTAİNERLARI GÖRÜNTÜLEME** docker ps -a ile tüm mevcut containerlarımı görebiliyorum ama eğer -a yazmazsam sadece çalışan containerları görebilriim.

**CONTAİNER ÇALIŞTIRMAK** docker start containerismi veya idsi yazarsak container çalışır duruma geçer. Idnin sadece ilk 3 karakterini belirtmemde yeterli. Docker ps deki statusta up yazıyorsa bu container bu kadar süredir ayakta demiş oluyor. Eğer burası Exited ise bu kadar saniye önce durmuş anlamına geliyor.

**CONTAİNER DURDURMA** docker stop container ismi veya idsi ile containeri durduruyoruz. Idinin sadece ilk 3 karakterini belirtmemde yeterli oluyor burada.

**CONTAİNER SİLME** önce containerimi durdurmam gerekiyor. Sonrasında silme yapabilirim. docker rm containerisimi veya idsi (ilk 3 harfi yeterli).

**DOCKER CONTAİNER İÇERİSİNDEKİ ÇIKTIYI GÖRMEK** docker ps -a ile tüm containerlarımı bir göreyim. Bir containerimi çalışır duruma geçireyim. docker attach hangicontainerismi veya idsi yazarak ve yine burada idnin ilk 3 karakterini yazmam yeterli bu kodu diyerek bu containerin çıktısına ulaşabiliyoruz. İlgili containera bağlanmak yani.

**DOCKER CLI KOMUTLARI**

**RUN KOMUTU** bizim container oluşturmak için kullandığımız docker create ve containeri başlatmak çalıştırmak için kullandığımız docker strart ve attach işleminin komutlarını, run komutu bunların her üçünü birden aynı anda gerçekleştiriyor. Tek bir komutla üç komutun işini gerçekleştirmiş oluyoruz. Önce container oluşuyor sonra container çalıştırılıyor sonra attach işlemi gerçekleştiriliyor. docker run –-name oluşacakcontainerismi hangiimageıkullanacak isim veya imageid (ilk 3 harf yeterli). –-rm parametresini kullanırsak eğer container ne zaman durursa o container durduğu gibi silinecektir kullanımı ise docker run –-rm –-name myapp imageismiveyaidsi

**RUN KOMUTUNDA PORT PARAMETRESİ** run komutuyla beraber bir port numarası belirtmem gerekiyor. Benim uygulamam hangi portta ayağa kalkacak. Bir container ayağa kalktığı zaman o container içerisinde çalışan uygulamanın bir portu var birde benim kendi işletim sistemi yani Windowsun bir portu var. Eğer burada port belirtmezsem o container içerisinde çalışan uygulamaya ben localhost üzerinden erişemem. docker run - -name containerismi -p 5000:80 0bc92847d9d4. Bu kod üzerinde diyorum ki bu imageid ye bağlı container dışarıdan 5000 portu gitsin container içerisindeki 80 portuna bağlansın demiş oluyoruz. Eğer burada 80 yani hiçbir şey belirtmezsek otomatik default olarak container içerisindeki çalışan uygulama 80 portundan çalışacaktır. Yani bu 80 portundan çalışan uygulama şuanda container içerisinde çalışıyor benim buna dışarıdan erişebilmek için 5000 portu gidecek container içeriisndeki çalışan 80 portundaki uygulamaya bağlanacak. Sonuçta containerda aslında kendi içerisinde mini bir işletim sistemi bu mini işletim sistemi üzerinde aspnetcore uygulamasının çalışabilmesi için gerekli ortam ve uygulamanın publish edilmiş dataları vardı yani dolayısıyla bu containmerda kendi içerisinde bir dünya işte bu ayrı bağımsız olan containera bağlanmak için buradaki 80 portu gibi bir port üzerinden çalışıyor. ASp net core uygulamalarmızın imageini bu kod ile containerını ayağa kaldırırız ve 5000 portu ile bu çalışan containerdaki uygulamaya erişebiliriz. Yani özetle buradaki 80 portu container içinde çalışan uygulamanın portu oluyor bu porta ulaşmak için dışarıya açtığımız port ise başa yazdığımız 5000 portu oluyor.

**RUN KOMUTUNDA -d PARAMETRESİ** eğer docker run -d - -name containerismi -p 5001:80 imageid:v1 gibi bir kod yazarsak buradaki -d şu işe yarıyor normalde -d kullanmazsak çalışıtrılan containera otomatik olarak attach olunur ve containera bağlamış olur ama -d kullanırsak dettach demiş oluyourz yani container yoksa oluşturulacak çalıştırılacak ama o containera bağlanmayacak.

**RMİ KOMUTU** bu komutu bir imagei silmek için kullanıyoruz. Önce bu image’e bağlı olan containerlari silmemiz lazım sonrasında docker rmi imageismi veya imageid (ilk 3 harf yeterli) containeri sadece durdurmuş olmak yetmiyor eğer sadece durdumuş olmanın yetmesini istiyorsak force patametresini kullanmamız gerekli.

**IMAGELERE ETKİKET ATAMAK** yani imagelarımı versiyonlayabilirim bu sayede. docker build -t imageismi:v1 . Daha sonrasında bu imagenin containerini ayağa kaldırmak için docker run - -name containerismi imageismi:v1 şeklinde versiyon seçebilir hale geliyor. Bu şekilde uygulamamın imagelerini versiyonlayabiliyorum.

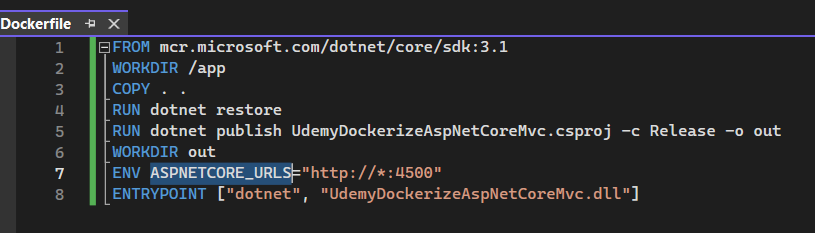
**DOCKER RMİ/RM FORCE PARAMETRESİ** container için konuşursak rm komutuyla silme işlemi yapabilmek için önce o containeri durdurmamız gerekiyordu. docker rm container-id/name - -force komutuyla birlikte bir container çalışsa bile hiçbir uyarı almadan silme işlemini gerçekleştirebiliyoruz. Image için ise durum eğer o imageye bağlı bir container aktif bir şekilde çalışıyorsa ilgili imageyi force parametresini kullansak dahi silemiyoruz. İmagenin bağlı olduğu containeri durdurmam lazım diyelim containeri durdurduk ama force parametresini kullanmadan o imageyi yine silemiyoruz container durmuş olsa bile sonuç olarak hala o image o containera bağlı o yüzden silemiyoruz bu durumda ise docker rmi image-id/name - -force komutunu kullnırsak ancak silme işlemini yapabiliyoruz.

**DOCKER PULL** bu komut ile docker hub sitesinden istediğimiz bir imageyi kendi local dockerimiza çekebiliyoruz. docker pull mcr.microsoft.com/dotnet/sdk:6.0 burada versiyon yani tag belirtmessek burası lastest olacak yani son versiyon gelecektir dikkat.

**DOCKER PUSH** bu komut pull komutu gibi direkt olarak docker hub ile haberleşir. Docker hub içerisindeki respositorylere private ya da public olarak imagelarımızı gönderebiliyoruz. Repository dediğimiz şey docker tarafında imagelarınızın tutulduğu yerdir. Docker CLI direkt olarak docker hb ile haberleşiyor ancak Azure CLI kullansaydık bu sefer azure tarafında imagelar azure container registery içerisinde kaydedilirdi. Yani kullandığımız CLI ‘a göre bizim ortamımız değişecektir. Biz Docker CLI üzerinden ileliyoruz dolayısıyla direkt olarak Docker Hub ile haberleşiyoruz. Docker hub üzerinden kendi repositoryimizi oluşturabiliyoruz. Bu reposityoryi public ya da private yapabiliriz. Önce docker images ile mevcut imagelarımızı bir görelim sonrasında repositorye göndermek istediğim imageye docker tag imageismi/imageid doganciftcir6/udemyrepository:v1 tagını istediğim imageye veriyorum. Burada yeni oluşan image eski imageyi referans alıyor dolayısıyla iki imagenin id kısmı aynı oluyor performans kaybı yaşamamak için sonrasında bu noktada dockera login olmuş olmamız önemli docker login diyerek login durumunu kontrol edebiliriz eğer bunu yapmazsak bizden kullanıcı adı şifre isteyebilir sonrasında docker push imagename/imageid:v1 diyerek bu imageyi docker hubdaki repositoryme gönderebilirim.

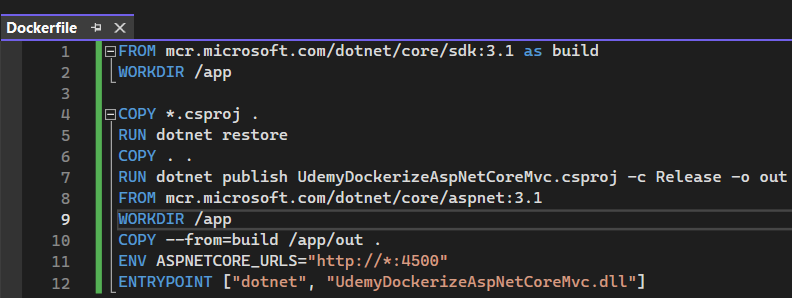
**.NET CORE CLI NEDİR** biz pcmize .net sdk kurduğumuz zaman default olarak gelen bir komut satır arayüzüdür. Buradan core uygulamaları oluşturabilir var olan uygulamalarımızı build işlemi gerçekleştirebilir veya publish işlemi gerçekleştirebiliriz bir çok komutu var. .net core uygulamalırımızı geliştirmeyi sağlayan bir komut satırı arayüzüdür aynı docker cliı gibi docker clida da docker komutlarını kullnarak docker ile haberleşmemizi sağlıyordu. Bu dotnet cli sayesinde dockerfile dosyamızın içeriisnde bu komutları çalıştırabilirim. Bu sayede COPY ile elle publish edilmiş dosyanın yolunu yazmak yerine ben burada publish edilme işlemini gerçekleştirebiliyoruz. Eğer böyle yapmazsak bir image oluşturduk diyelim biraz uygulamda geliştirme yaptık sonra tekrar image oluşturmak için bu sefer tekrar elle publish alıp o dosyayı image’e vermem lazım işte bizi bu manuel olarak publish alma işleminden kurtarıyor. Ancak burada önemli nokta runtime içerisinde cli gelmiyor sdk içerisinde geliyor. O yüzden base image olarak runtime değil sdk almamız gerekiyor.

**DockerFile DOSYASINDA .NET CORE CLI KOMUTU YAZMAK** bu işlem için baseimage olarak runtime değil sdk imagesine ihtiyacımız var runtime imagesinin içinde .net cli bulunmuyor. COPY kısmında İlk nokta benim Dockerfile dosyamın konumunda bulunmuş olan tüm dosyaları ifade ediyor ikinci nokta ise image içeriisndeki o anki çalışma dosyasını ifade ediyor yani /app Bu sayede benim projemdeki tüm dosyalar image içerisindeki /app içerisine gitmiş oluyor. Ardından RUN komutu ile beraber ben artık donet cli kodlarımı çalıştırabiliyorum. Restore komutu publish öncesinde eğer güncellenecek bir library varsa veya bu library eklenmiştir ama nuget package managerdan çekilmemiştir işte restore bu çekme işlemini aynı zamanda da bir güncelleme işlemi varsa library veya bağımlılılarda bunların güncellenmesini gerçekleştriyor. Yani restore ile beraber projemi güncel hale getiriyorum. Ardından RUN komutuyla beraber dotnet publish yazıyorum ve hangi uygulamayı publish edeceğimi .csproj uzantılı dosyayı veriyorum -c Release ile release modda publish yapacaksın diyorum ardından -o out diyerek out klasörünü publish işlemini gerçekleştir diyoruz yani bu /app içerisinde bir out klasörü oluşturup buraya publish edilmiş dosyaları atacak. Daha sonra WOKRIR out ile /app içeriisndeki out klasörüne geçmiş oluyoruz. ENV kodu ile default olarak 5000 portunda çalışan sdk içerisinde bulunan bu ASPNETCORE\_URLS envini değiştireceğiz. Diyorum ki htpp ile başla ama adresi bilmiyorum bu adress localhostta çalışmamalı herhangi bir adreste çalışmalı o yüzden \* yani adresin ne olursa olsun ama mutlaka 4500 portunda çalış. Yani otomatik publish alan dockerfile kodum bu şekilde gözükecek, bu noktoada container içindeki uygulama artık 4500 portunda çalışacak o yüzden container portunda 4500 diye belirtmeyi unutmayalım bu kısımlar aynı olmak zorunda dikkat bu noktayı launchSettings.json ile karıştırmayalım bu dosya sadece geliştirme aşamasında var geçerli uygualamyı publish ettiğim anda bu dosya gider sdk içindeki default env geçerli olur o env ise default 5000 portunda çalışır ancak runtime üzerinden çalıştıtırsak bu env orada yok ve dafault olarka 80 portundan çalışır uygulama çünkü runtimeda uygulama localde çalışmıyordu eğer sdk varsa uygulama localhostta çalışır o yüzden sdk image kullanyorsam env olarak portu mutlaka ayarlamam lazım;

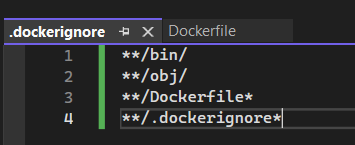




**DOCKER MULTİSTAGE BUİLD** dockerfile içerisinde uygulamanızı inşa ederken bir tane base imageden değilde birden fazla base imagedan faydalanıyorsanız buna multistage deniyor. Daha önce sdk imagesi kullanmak zorunda kalmıştık oto publish için ama bu image çok büyük boyutlara sahip. Benim burda bir optimizasyon yapmam gerekiyor. İşte bu boyut küçültme işlemi için multistage dediğimiz birden fazla imageden faydalanıcaz. Burada yaptığmız mantık şöyle sdk üzerinden otomatik publish alıyoruz dotnet cli komutlarını kullanarak sonrasında 2 adet copy kullanarakc catcheden değişiklikleri çekmesini sağlıyoruz dockerin performans için sonra sdk boyutu çok büyük olduğu için bu sefer runtime imagesini alıyoruz ve daha önce sdk üzerinden publish edilmiş projemizin datalarını alıp 3. Copyde önceki imagede belirttiğimiz yoldaki dataları alıp /app/out yani burdaki dataları yeni imagenin /app klasörüne . ile kopyala diyoruz ve bu sayede runtime içinde otomatik olarak publish alınmış projemiz oluyor. Bu sayede imagemizin boyutu 3 kat kadar azalıyor sdk üzerinden image oluşturmamış oluyoruz. Sdk imagesini sadece dotnet cli komutlarını çalıştırmak için kullanmış oluyoruz bu noktada.



**DOCKER İGNORE DOSYASI OLUŞTURMAK** dockerfile içerisinde kopyalama komutları yazdığımız zaman yani bizim projemizden image içerisine bir kopyalama işlemi gerçekleşirken dockera şu klasörleri veya dosyaları kesinlikle image içerisine kopyalama diyebiliyoruz. Bu işlemi iste .dockerignore dosyası ile yapıyoruz. Bu dosya içerisinde taşınmasını istemeyeceğimiz dosyaları belirtebiliyoruz. \*\* demek benim projemdeki tüm klasörleri ara. İsimden sonra gelen \* ise bu isimden sonra ne olursa olsun yani Dockefile3 buradaki 3 önemsiz olsun diye yapıyoruz. Bin, obj, dockerfile, .dockerignore dosyalarını image içerisine aktarmama hiç gerek yok.



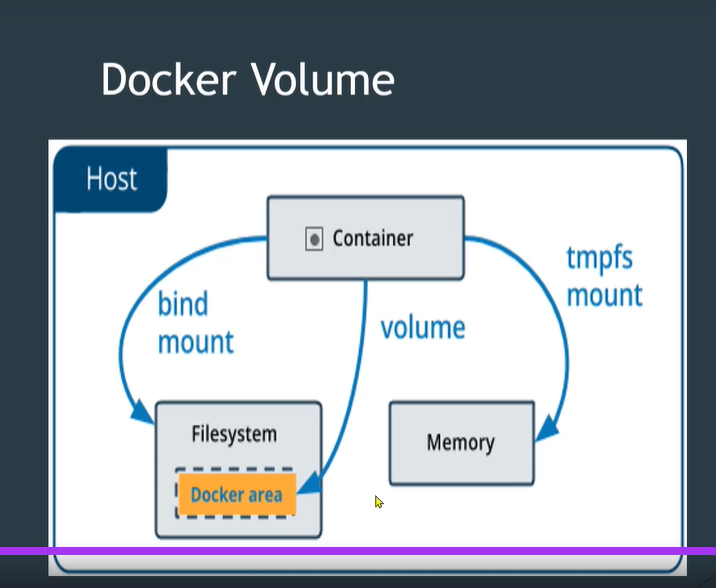
**DOCKER VOLUME**

**NEDİR** docker containerlar tarafından üretilen dataları kalıcı kılmamızı sağlayan bir yöntemdir. Diyelim .net uygulamasından resimler alıyoruz ancak docker container durdurulduğu anda bu resimler gider veya diyelim 2 tane container ayağa kaldırıyoruz bu containerların aldığı resimler kendilerine has yani bir containera yüklenmiş olan görseli veya dosyayı diğer containerda göremeyiz. Birbirinden bağımsız 2 ayrı uygulama oldukları için. Bu problemi çözmek adına containerlar tarafından üretilen dataların yani resimlerin vs kalıcı kılmanın bize yöntemini sunuyor aslında volümeler. Burada 3 tane datayı kalıcı kılmamızı sağlayan yöntem var;

**-Bind Mount=** Bizim işletim sistemi üzerindeki bir dosyaya containerlar tarafından üretilen dataların kaydedilme işlemidir.

**-Volume=** Docker cli tarafından yönetilen ve datalarımızı kalıcı kılmak için işletim sistemi üzerine değil dockerin kendi içerisindeki bir özel alanı oluşturulmasına deniyor. Volumeleri docker cli komutlarıyla yönetebiliyoruz yönetimi tamamen dockera ait. Bind mountta her şeyi manuel yapmamız lazım ama volumelar docker içerisindeki özel alanlarda olduğu için yönetimi tamamen dockera ait oluyor. Volumelari sadece kendi işletim sistemimiz üzerindeki docker üzerindeki değil uzak Cloud ortamdaki volumelerede gönderebiliyoruz dataları. Volumeler hem Windows hem linux containerlarda kullanılabiliyor. Çok daha performanslı bir yöntem.

**-tmpfs mount:** Containerlar tarafından üretilen datalar işletim sistem üzerindeki memoryde kaydedilir bu memoryden daha sonra containerlar dataları okuyabilir.



**Biz ikinci yolu tercih edelim** daha sağlıklı volume yani ama ilk yolda kurs içeirsinde var bilgisayara bir klasör olışturup container ayağa kaldırırken o klasörü containera bağlıyor dolayısıyla dosylar wwwroottan değil o klasörden çekiliyor. Biz bunu kullanmayalım ikinci yolu anlatalım. Önce docker içerisinde bir volume oluşturmalıyız. docker volume create images. İmages isminde bir volume oluşturduk. Artık elimde bir volume olduğuna göre ben containerimi ayağa kaldırırken bu volumeu bağlama işlemini gerçekleştirebilirim. Kodum bu şekilde olacak docker run -d -p 5000:4500 --name mycon -v images:/app/wwwroot/images 0e61f45f9a9b bu sayede docker image içindeki /app klasörü içinde wwwroot içinde images klaösründe bulunan dosyalar daha önce docker içinde oluşturduğumuz volume’E kopyalanır ve resim bilgileri wwwroot yerine bu volumedan çekilerek gösterilir. Bundan sonra eklenen ve silinen resimler wwwroot değil docker içerisindeki images volumeuna kaydedilir.

**VOLUME SİLMEK** önce volume’a bağlı olan containerları çalışan varsa durdurmalıyız. Sonrasında volume silmek için docker volume rm volumeismi komutunu yazarak silme işlemini gerçekleştirebiliriz.

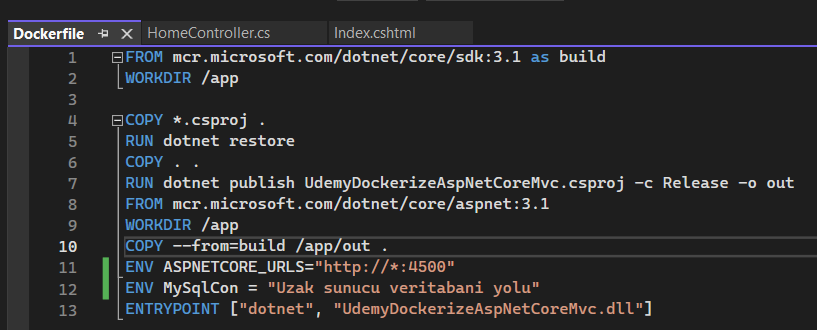
**NO-CACHE KOMJTU** image oluştuturken docker build --no-cache -t udemyaspcoremvc:v4 . şeklinde bir kullanım yaparsam tüm layerları 0 dan oluştrur catche den hiç kullanmaz. Bazen yeni bir image oluşturup container ayağa kaldırdığımızda değşikliklerimiz gözükmüyorsa bu komutu kullanabiliriz bu sayede imagenin tüm layerları 0’dan oluşturulur.

**DOCKER ENV**

**Asp.Net Core tarafında bir uygulama oluşturduğunuz zaman default olarak 3 tane env gelir.** Production, Development, Staging Evresi. Bunlardan herhangi bir tanesini seçerek uygulamayı ayağa kaldırabiliriz. Eğer herhangi bir env belirtmezsek uygulama production ortamda ayağa kalkar. Uygulamayı localhostta ayağa kaldırıyorsak default olarak development modde ayağa kalkar. Yani uygulamayı bir exeden çalıştırırsak yani sunucuda çalıştırırsak ortam default olarak production oluyor, localde çalıştırdığımız zaman ortam default olarak development oluyor. Eğer ortamımız production ise bir hata aldığımız zaman hatanın detayını göremeyiz. Eğer ortamımız development ise o zaman hatanın detayını görebiliriz. Bu noktada container içinde default olarak uygulama production modda çalışacağı için hata alırsak ancak hatmaızı loglama yaptıysak loglarda görebiliriz. Eğer loglama sistemimiz yoksa bazen canlıdaki ortamımızı hataları anlamak için development yapmamız gerekebilir. Docker container ayağa kaldırırken env belirtebiliyoruz bu container için. docker run -p 5000:4500 --env ASPNETCORE\_ENVIRONMENT=DEVELOPMENT --name mycon2 668b43c4b35d

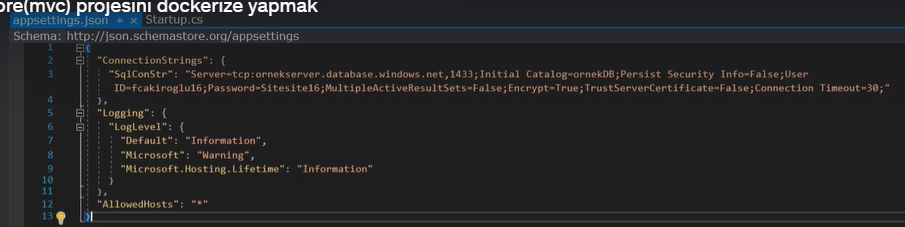
**<none> IMAGE NEDİR?** Her yeni versiyon image build alma işlemi gerçekleştirdiğimde herhangi bir tagı olmayan <none> isminde image oluşabilir. Docker bu none imageleri cachleme mekanizması oluşturmak için kullanıyor. İstersek bunları silebiliriz silmek faydalıdır. None imageleri listelemek için -f filterin kısaltılmışıdır. docker images -f "dangling=true" bunları toplu bir şekilde silmek için docker rmi $(docker images -f “dangling=true” -q), -q bu imagelerin idlerini al demiş oluyor.

**ENVLERİ SADECE ORTAM belirlemek için değil** **appsettings.json** içerisinde belirlemiş olduğumuz değişkenleri ezmek içinde kullanıyoruz. Diyelim ki appsetting.json içerisinde MySqlCon adında bir envim var. Bu development ortamda localde bulunan db connection stringi alacakken Production ortamda ise uzak sunucu veritabanı yolunu alması gerekecektir. Ama bunu production ortamda appsettingste direkt olarak yazmamı riskli bir durum bunun yerine secret keyleri kullanabiliriz. Konumuza dönelim production ortamında bu değerleri appsettingste belirlemek yerine uygulamamı container olarak ayağa kaldırırken env olarak belirlersem uygulamamın güvenliğini arttırmış olurum. docker run -p 5000:4500 --env MySqlCon="uzak sunucu veritabani yolu" --name mycon3 c33ad2ee0934 veya böyle tanımlamak yerine Dockerfile dosyamızın içeriinse envlerimizi tanımlayabiliyoruz. Kritik datalarımızı product ortamında appsettings.json içerisinde tutmak yerine bu yolları kullanmalıyız.

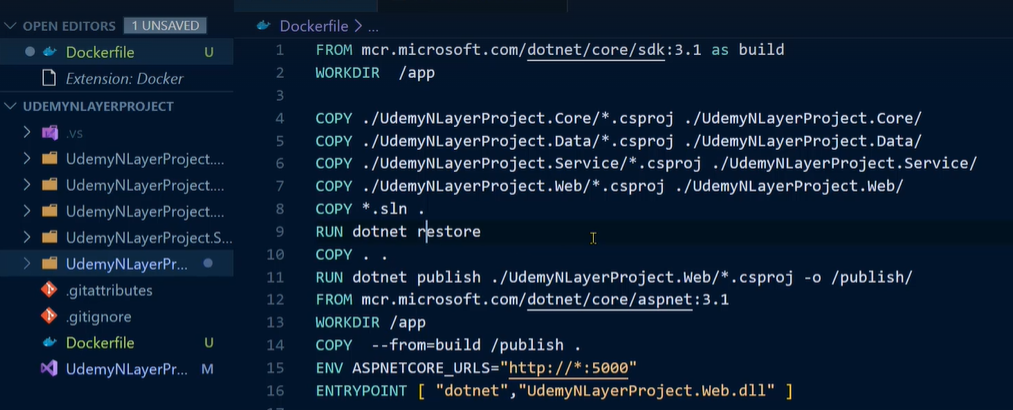


**ÇOK KATMANLI .NET CORE PROJESİNİ DOCKERİZE YAPMAK**

**Uygulamamızı** docker container içerisine attığımız zaman orada kendi bir yaşam alanı olur bizim local ortamımızdan tamamen bağımsız olur. Bu yüzden projemizi eğer dockerize yapıyorsak mutlaka uzak bir sunucudaki veritabanı yolunu kullanmalıyız bu birinci yöntem. Eğer illa ben localdeki dbye bağlanmak istiyorsam sql serveri çalıştıracak bir container ayağa kaldırırız bu containerında kendi portu olacak daha sonrasında o sql serveri ayağa kaldıracak containerımızın bilgilerini veritabanı yolu gibi bilgileri appsetting.json içerisine verebiliriz. Yani hem kendi projemiz bir containerda ayağa kalkar hemde sql server çalıştıracak container ayağa kalkar. Bu durumda ise docker compose kullanmamız gerekiyor yani birden fazla containeri aynı anda ayağa kaldırabileceğimiz bir dosyayı kullanmış oluyoruz. Bunu sadece dockerize edilmiş projemizde local db kullanmak istiyorsak yapıyoruz yoksa normal şartlarda uzak sunucudaki kurulan db nin yolu kullanılır ayrı bir db container ayağa kaldırmaya gerek yok. Azure kullanabiliriz uzak sunucu db için.

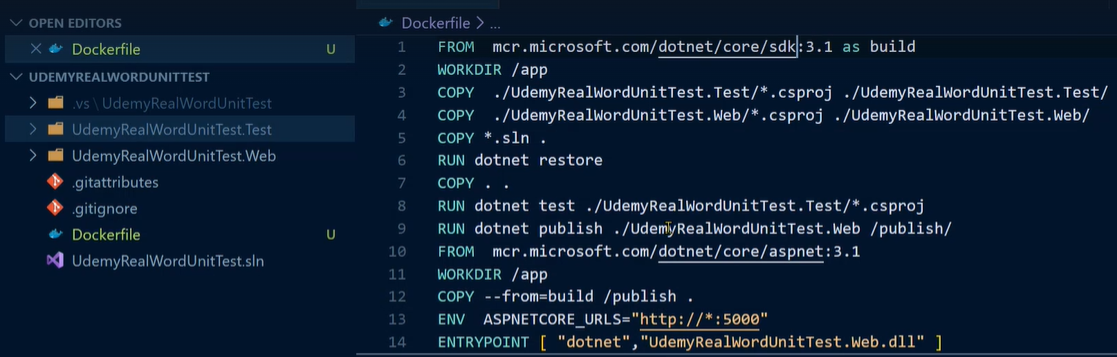


Çok katmanlı projede Dockerfile dosyasını solution kapsamında oluşturmak gereklidir. Diğer projede bulunan katmanlara erişimimiz kolay olması açısından. Dockerfile dosyası içerisinde projemizin tüm dosyalarını COPY . . diyip WORKDIR alanına atmak yerine (burada ilk nokta source ikinci bu sourcenin atılacağı yer yani dist anlamına geliyor) projemizin her katmanını ayrı ayrı layer olarak WORKDIR yani çalışma alanına atarız çalışma alanında da bu katmanları ayrı ayrı klasörlere alırız oralara ilgili katmanları ilgili klasörlere atarız. Çünkü mesela sadece dataccess katmanında değişiklik yaptığım zaman sadece bu katman etkilensin yani docker her seferinde diğer tüm katmanları sıfırdan oluşturmaya uğraşmasın diye. Diğer katmanları cacheden getirecek hız kazanmış olacağız docker işlemlerimizde.



**UNİT TEST İÇEREN ASP.NET CORE PROJESİNİ DOCKERİZE ETMEK**

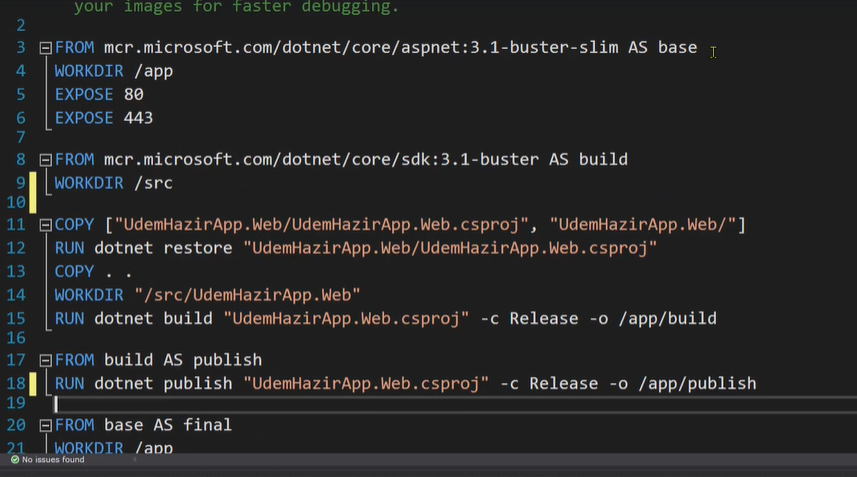
Yine katmanlı mimaride olduğu gibi Dockerfile dosyamızı solution kapsamında oluşturmalıyız. Dotnet test komutu ile unittesti başlatmış oluyorum bu komutu kullanacağız.



BU kod hata verir bu satırı düzelt;



**Visual Studionun proje oluştururken bize sorduğu docker kullanayım mı sorusuna** eğer cross bir uygulama kullanıyorsak linux seçmeliyiz çünkü linux mini işletim sistemi windowsa göre çok daha az yer kaplar linux seçtikten sonra bize ototmatik olarak verdiği Dockerfile dosyasının içeriği bu şekilde. Burada sdk imajı olarak buster kullanırken runtime imagesi olarak buster-slim kullanıyor. Bu kullanılan imageler normal runtime ve sdk imagelerine göre daha düşük boyuttalar dolayısıyla bu imageleri kullanmak daha mantıklı. Sdk imagesi üzerinden önce uygulamayı restore işlemi yapılıyor bu sayede eksik library veya dependencyler tamamlanıyor veya güncellenmesi gerekirse güncelleniyor bunu yapabilmek için ise ilk COPY komutunda ilgili katmandaki .csproj dosyası /src klasörü altında UdemyHazir.Web/ adında bir klasör oluşturulup bu klasör içine atılıyor bu sayede o atılan dosya restore yapılabiliyor sonrasında ikinci COPY komutuyla beraber projede geri kalan tüm dosyalar /src klasörünün altına kopyalanıyor ancak burada yanlışlar var ilk COPY komutunun en başındaki UdemyHazirApp.Web/ ifadesini silmeliyiz zaten Dockerfile dosyasının olduğu yerde direkt olarak bu .csproj dosyası var ve ikinci COPY komutunda tüm proje dosyalarını /src altına değil /src altındaki MicroService1.API/ klaösrünün altına kopyalamak için ikinci nokta yerine ./MicroService1.API/ yazmalıyız ilk COPY ile restore işlemi yaparak bu proje içerisindeki tüm eksik paketleri tamamlamış veya güncellemiş oluyorum. Çünkü paketler veya dependencyler projenin .csproj dosyasına tanımlanır. Eğer indirilen paket restore alınıp .csproj dosyasında include edilmezse bu paketi projemde kullanamam. Yani burada .csproj dosyasında include edilmiş tüm paketleri image’ın içerisinde kullanılabilir hale getiriyoruz. Bunu ayrı bir COPY komutu ile yapma sebebi ise diyelim ki projeye yeni bir paket eklemeden sadece yeni bir controller veya class ekledin bu eklenen class bu ilk .csproj u kopyaladığımız komutu çalıştırmaz bu komut cacheden getirilir dolayısıyla sonrasındaki restore komutu da cachden getirilir bu durum performans için önemli. Restore komutuyla beraber image içerisinde bir obj klasörü ve verdiğimiz .csproj dosyası oluşuyor. Bu obj klasörü .csproj dosyası içerisinde include edilmiş tanımlamış paketleri kullanılabilir hale getiriyor. İşte burada asıl amaç projeye en küçük dosya eklediğimizde restore komutu çalışmasın bu .csproj obj ve restore komutunu docker cacheden getirsin diye paket yüklemediğim sürece bu işlemler cacheden gelsin tekrar tekrar çalışmasınlar. Biz bir paket yüklediğimizde arka planda otomatik olarak restore çalışıyor ve indirilen paket kullanılabilir hale getiriliyor ama eğer biz paketi indirmeden .csproj doyasında ismini tanımlarsak mutlaka bu paketin kullanılabilir hale gelmesi için restore yapmamız ve bu restore komutunun o paketi obj klasörüne tanımlaması gerekir mantık böyle. Daha sonrasında artık çalışma klasörümü değiştiriyorum WORKDIR /src/MicroService1.API diyerek çünkü yukarıdaki kodlarda projenin kendisini bu klasör içerisine aktardık dolayısıyla çalışma klasörünü projenin bulunduğu alan olarak değiştirmeliyiz. Arkasından ise içerisine build yapıyor. Build yaparken ise Release modda build yapıyor ve WORKDIR’in içerisinde /app/build klasörüne build yapılan veriyi göndermiş oluyor. sonrasında build işlemi yapılıyor bu sayede uygulamadaki kodlar hem güncelleniyor yani son güncel haline getiriliyor hem uygulamada bir hata var mı diye bakılıyor eğer build işlemi başarılıyla tamamlanırsa publish işlemi yapılıyor çünkü uygulamamızı dockerize edeceksek publish edilmiş data üzerinden dockerize etmeliyiz containerlarda local’e ait hiçbir nokta olmamalı production ortamı gibi düşünmeliyiz. Production ortamında da projenin publish edilmiş datası kullanılır ve sonrasında publish edilmiş projenin datası ile buster-slim imagesi üzerinden bir image tanımlanıyor ve daha önce sdk üzerinden hazırlanılmış restore yapılmış build alınmış ve publish alınmış projemizin dataları bu düşük boyutlu runtime imagesine kopyalanıyor bu sayede sdk nın büyük boyutundan kurtulunuyor. Sdk sadece dotnet cli komutlarını çalıştırmak için kullanılıyor. Bu yapılan işleme multistage build adı veriliyor. Burada eğer containerlarımızı birbiri ile haberleştirmeyeceksek expose komutlarını kullanmamıza gerek yok ayrıca. Sonrasında Docker CLI komutları ile uğraşmak yerine direkt olarak projeyi ayağa kaldırma seçenekleri kısmında eğer dockeri seçersek visual studio üzerinden docker işlemlerini cli kodları yazmadan gerçekleştirebileceğimiz bir pencere açılır buradan tüm docker işlemlerini yönetebiliriz ayrıca docker üzerinden projeyi ayağa kaldırdığımız anda vs bizim yerimize bir image oluşturur ve bu imageden bir container ayağı kaldırarak bu container üzerinden projeyi ayağa kaldırmış olur.

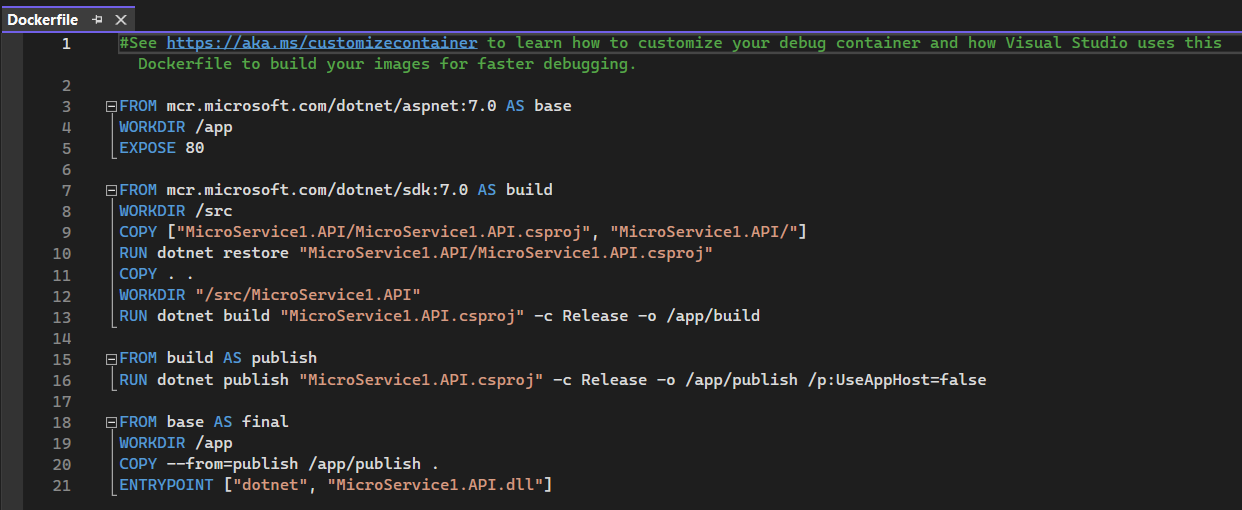




**DOCKER COMPOSE**

**NEDİR?** Bilgisayarımıza docker kurduğumuz zaman bir Docker CLI geliyor birde Docker Compose CLI geliyor. Bu Docker Compose CLI bir tooldur. Bu tool ile beraber containerlarımızı imagelerimizi özellikle birden fazla containeri birdan fazla imageyi kolay bir şekilde yönetmemize imkan sağlıyor. Mikroservis yapıda birden fazla bir çok containerımız olacağı için docker compose’u çok sık kullanıyoruz. Mikroservis yapıda tek tek tüm containerlarımı elimle ayağa kaldırmak oldukça yorucu bir işlem ben istiyorum ki öyle bir tool olsun ki tek komut yazayım o tek komut ile beraber kaç tane mikroservis belirttiysem hepsi tek seferde ayağa kalksın veya dursun veya silinsin. İşte burada devreye Docker compose giriyor. Docker composeu development amacıyla kullanıyoruz bunu production ortamda kullanmıyoruz. Development esnasında birden fazla containerimizi ve imagelarımızı kolay bir şekilde yönetmemize imkan sağlıyor. Production ortamda containerlarımızı veya imagelarımızı yönetmek istediğimizde Strom veya Kubernet gibi sistemler kullanıyoruz. Dolayısıyla docker composeu productionda kullanmak uygun bir yaklaşım değildir. Formatı YAML formatıdır. JSON formatından daha sadedir JSONDA süslü parantezler varken YAML’da bunlar bulunmamaktadır girintilerle belirtiyoruz hangi parçanın hangi parça altında olacağını özellikle Configuration tarafında JSON’dan YAML’a doğru bir geçiş bulunuyor çünkü okuması JSON’a göre çok daha temiz duruyor. Docker compose CLI yine arka planda aslında Docker CLI ile haberleşmektedir.

**IDELERİMİZİN ÜRETTİĞİ DOCKERFİLE DOSYASINI İNCELEYELİM** Bir çok idenin dockerfile desteği var. VS Code kullanıyorsak bunu bir plugin ile sağlayabiliriz vs nun ise kendisinde docker desteği var. Bunların kendileri bize bir Dockerfile üretmektedir. Projemizin üstüne sağ tıklayıp Add menüsünden Docker Supportu seçersek eğer projemize bir Dockerfile dosyası ekler ancak Container Orhestrator Sapport seçeneğini seçersek projemize Docker compose dosyası ekleyecektir.



İlk olarak runtime base image alıyor ve buna as diyerek base ismini veriyor. Arkasından WORKDIR ile çalışma klasörünü /app olarak belirliyor. Ardından **EXPOSE** 80 diyerek eğer expose yapmazsak bunu bir image yaptık container olarak ayağa kaldırdık container hangi porttan ayağa kalkıyorsa o porta dışarıya açılan portla yani portu maplayip (port forward) erişip container içerisindeki uygulamaya erişmiş oluruz ya da porta gerek olmadan container içerisine basescirpt ile bağlanabiliriz ve gün sonunda container içerisinde çalışan mikroserice’e erişebiliriz ancak başka containerlarımın içinde çalışan mikroservislerim bu containerin içerisindeki microservise erişemezler. Dolayısıyla ben bunu EXPOSE ettiğim anda docker içerisinde bu container içerisinde çalışan bu microservisime başka container içerisinde çalışan microservisler 80 portundan bu arkadaşa erişebilir demiş oluyoruz. Yani bu containeri docker içerisinde diğer containerlara açmış oluyoruz. Ama hala docker dışından -p ile yani port numarası ile bu containeri dışarı açmadığımız sürece bu containera dışarıdan erişemeyiz sadece docker içerisinde haberleşmeye açık olur dış dünya ile haberleşemez. Httpnin default portu 80 Https in ise default portu bu noktada 443 tür https ‘i işin içine dahil etseydik birde EXPOSE 443 yazmamız gerekirdi bu noktada. Şöyle önemli bir nokta var diyelim ki EXPOSE kullanmadık ama -p 5000:80 komutunu kullanıp bu 80 portunda çalışan containeri 5000 portundan dışarıya açtık burada gizli bir EXPOSE oluyor dışarıya açtığımız 5000 portundan gittiğim zaman 80 portunda çalışan containera erişiyorum docker böyle bir şeyi gördüğü anda madem ki bu 80 portu bu container için dış dünyaya açıldı o zaman bu 80 portuna docker içerisindeki başka containerlarda erişebilir diyor yani aslında gizli bir EXPOSE işlemi olmuş olmuyor. Dışarıya açılan bir servise iç dünyadan erişememek saçma olurdu. Yan 3 durum var; İlk durum EXPOSE hiç yazmadık ve -p kullanarak bu containeri dışarıya açmadık bu durumda ne dışarıdan bu containera erişebiliriz nede iç dünyada diğer bir container bu containera erişebilir sadece bu containera bağlanırsın bu container içerisinden bu servise erişebilirsiniz. İkinci durum EXPOSE kullandım ama -p kullanmadım bu durumda bu containera iç dünyadan diğer containerlar erişilebilir ama dış dünyadan bu containera erişilemez. Üçüncü durum EXPOSE yapmadım ama -p yaptım bu durumda aynı zamanda gizli bir EXPOSE oluyor hem dışarıdan containera erişlebiliyor hem de iç dünyadan diğer containerlarda bu containera erişebiliyor. Best Pratice açısından her zaman EXPOSE kullanarak yazmak önemlidir ayrıca her zaman port forward yapacaksın diye bir durum yoktur. Sağlıklı olan durum ise her bir container sadece bir tane uygulamayı ayağa kaldırsın bir container birden çok servis uygulama ayağa kaldırabilir ama sağlıklı değildir best preatice açısından da uygun değildir. Bu durum kübernet içinde geçerlidir.

**2.KISIM COPY** Bir kopyalama işlemi gerçekleştiren bir komuttur. Burada en başta 2 adet COPY komutu kullanma sebebi cachleme yapmak ve build süresini kısaltmak. Dockerfile dosyası içerisinde her bir satır cachlenir o komut satırı ile ilgili bir değişiklik meydana gelmediyse o satırı cacheden okuyor. İlk COPY kodunda /src klasörü altında bir MicroService1.API adında bir klasör oluşturup, API katmanının .csproj dosyasını bu klasörün içerisine atıyor. Bunu yapma sebebeimiz ise restore komutunu çalıştırabilmek için. Ardından ikinci COPY’de ise projedeki geri kalan her şeyi /src klasörünün altına kopyalıyor. Ancak burda bir takım yanlışlar var ilk COPY komutundaki en başta bulunan MicroService1.API/ kısmını silmeliyiz çünkü .csproj dosyası zaten Dockerfile dosyasıyla aynı konumda bulunuyor ikinci olarak ise ikinci COPY komutunda tek . yaptığımızda WORKDIR ‘ı ifade etmiş oluyoruz projenin geri kalan tüm dosyalarını /src altına değil /src altındaki MicroService1.API/ klasörüne kaydetmek için ikinci nokta yerine ./MicroService1.API/ yazmalıyız. Bu sayede 2 COPY kullanarak önce ilk COPY ile restore işlemi yaparak bu proje içerisindeki tüm eksik paketleri tamamlamış veya güncellemiş oluyorum. Çünkü paketler veya dependencyler projenin .csproj dosyasına tanımlanır. Eğer indirilen paket restore alınıp .csproj dosyasında include edilmezse bu paketi projemde kullanamam. Yani burada .csproj dosyasında include edilmiş tüm paketleri image’ın içerisinde kullanılabilir hale getiriyoruz. Bunu ayrı bir COPY komutu ile yapma sebebi ise diyelim ki projeye yeni bir paket eklemeden sadece yeni bir controller veya class ekledin bu eklenen class bu ilk .csproj u kopyaladığımız komutu çalıştırmaz bu komut cacheden getirilir dolayısıyla sonrasındaki restore komutu da cachden getirilir bu durum performans için önemli. Restore komutuyla beraber image içerisinde bir obj klasörü ve verdiğimiz .csproj dosyası oluşuyor. Bu obj klasörü .csproj dosyası içerisinde include edilmiş tanımlamış paketleri kullanılabilir hale getiriyor. İşte burada asıl amaç projeye en küçük dosya eklediğimizde restore komutu çalışmasın bu .csproj obj ve restore komutunu docker cacheden getirsin diye paket yüklemediğim sürece bu işlemler cacheden gelsin tekrar tekrar çalışmasınlar. Biz bir paket yüklediğimizde arka planda otomatik olarak restore çalışıyor ve indirilen paket kullanılabilir hale getiriliyor ama eğer biz paketi indirmeden .csproj doyasında ismini tanımlarsak mutlaka bu paketin kullanılabilir hale gelmesi için restore yapmamız ve bu restore komutunun o paketi obj klasörüne tanımlaması gerekir mantık böyle. Daha sonrasında artık çalışma klasörümü değiştiriyorum WORKDIR /src/MicroService1.API diyerek çünkü yukarıdaki kodlarda projenin kendisini bu klasör içerisine aktardık dolayısıyla çalışma klasörünü projenin bulunduğu alan olarak değiştirmeliyiz. Arkasından ise içerisine build yapıyor. Build yaparken ise Release modda build yapıyor ve WORKDIR’in içerisinde /app/build klasörüne build yapılan veriyi göndermiş oluyor.

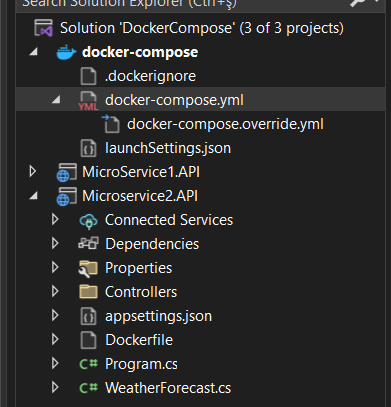
**3.KISIM PUBLISH** Sonrasında FROM build AS publish diyerek build takma isimli imageden 0 bir tane base image alıyor dikkat edersek sdk imagesini alıyor ve buna publish ismini takıyor. Sonraki komutta ise bu arada WORKDIR hala yukarıdaki ile aynı değişmedi. Burada MicroService1.API.csproj bu projeyi publish et Release modda publish et ve WORKDIR altında /app/publish adında bir klasöre publish et ve /p:UseAppHost=false ile, normalde publish yatığımızda bir .exe birde .dll dosyası oluşur burdaki exe biz insanlar için diye düşünebiliriz tıklayıp hemen açalım diye ama docker bu .dll dosyasını bilir veya linuxa attığımız zaman .dll’i bilir veya bir IIS te açtığımız zaman bu .dll ‘i bilir. Bu exe dosyası bizler insanlar için geçerli tıklayıp bu uygulamayı çalıştıralım diye var olan bir şey. Burada false diyerek diyorum ki bu exenin oluşmasına gerek yok çünkü docker sadece bu .dll’i biliyor docker içerisinde .exe dosyasına gerek olmadığı için burda false diyoruz. Ayrıca linuxta .exe çalışmaz bu sadece Windwosta olan bir şey bu bilgiyi de bilelim. Daha sonraki satırda ise artık base’den yani runtime imagesinden final isminde bir image alıyor. Daha sonra çalışma klasörünü /app yapıyor. Sonraki satırda ise –from ifadesi ile beraber publish imagesine git bu imagein içerisindeki /app/publish klasörünün içerisindeki tüm kodları . diyerek WORKDIR alanına at diyoruz daha sonrasındaki satırda ENTRYPOINT ile beraber container ayağa kaldırıldığında dotnet komutuyla beraber buradaki .dll dosyasını ayağa kaldır diyoruz.

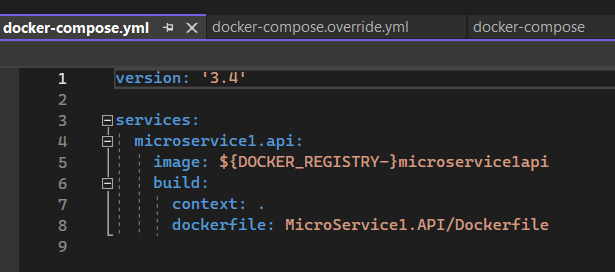
**COMPOSE TARAFI**

Docker Compose birden fazla imagelarımızı tek bir komut ile containerlarımızı ayağa kaldırmamıza imkan sağlıyor. Bu sayede özellikle mikroservis mimari geliştirirken birden fazla servisimizi tek bir komutla ayağa kaldırabiliyoruz tek bir komutla kapatabiliyoruz farklı ortamlara göre farklı seneryolarda containerlarımızı ayağa kaldırabiliyoruz işte productionda şu env ile ayağa kalksın productionda production dbsine bağlansın testte ayağa kalktığı zaman containerlarım test dbsine bağlansın gibi farklı farklı yaklaşımlar sağlayabiliyoruz.

Mikroservislerimi için her mikroservisin kendi Dockerfile dosyasını oluşturduktan sonra ki önceki gibi değişiklikler yapmaya gerek kalmadan direkt vs aracılığıyla bunları oluşturmak bu noktada mümkün.

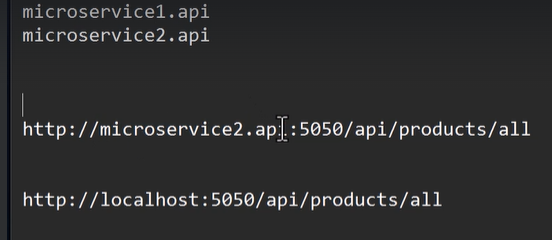
Mikroservicelerimizin kendi Dockerfile’larını oluşturduktan sonra ki bu noktada önceden yaptığımız gibi bu dosyaları düzenlememize gerek yok çünkü compose bu dosyaları görecek. Herhangi bir microservice’in üzerinde tıklayıp Add Container Orchestrator Supportu seçip Docker Compose diyip linuxu seçip compose dosyamı ekliyorum. Bizim için burada önemli olan docker-compose.yml ve docker-compose.override.yml dosyaları neden 2 dosya var çünkü biz docker composeları farklı farklı ortamlar ile ayağa kaldırmak isteyebilriiz. Docker-compose.yml base kodlamayı yapacağımı yer override ise ezmek istediğimiz veya değişik konfigürasyonla ayağa kaldırmak istediğimiz yer olacak. Aynı appsettings.json ve appsettings.Development.json dosyalarıyla aynı mantık. Appsettings base keywordlerimizi barındırdığımız yer Development ise developmenta özgü olarak kodları barındırdığımız yer.

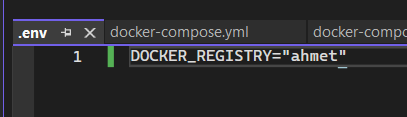




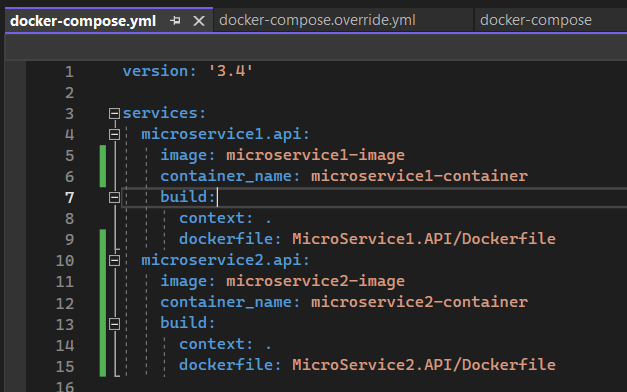
Buradaki Base dosyasına bir odaklanalım. Docker compose dosyalarının her birinin bir versiyonu var. Versiyon değişimi ile compose dosyasında bazı değişiklikler oluyor dolayısıyla versiyon belirtmek zorundayız. Daha sonra services’in altında service ismi belirtiyoruz. Bu isim önemli çünkü docker içerisinde mikroservisler birbirlerine istek yapmak istediklerinde bu servis isimleri üzerinden istek yapmak isteyecek yani bunlar aslında benim birere gizli domain name isimlerim. Buraya bizim ilk microservisimizi belirtmiş mesela. Bu servisin içerisinde bir image diye keyword var burada bu mikroservisin imagesinin ismi belirtiliyor. Bu image ismi bu service için oluşacak olan yeni image ismidir aynı şekilde burda oluşacak olan yeni container isminide belirtebiliriz. Hangi dockerfiledan bu imagei oluşturacağını ise altta build kısmında belirtiyoruz. ${DOCKER\_REGISTRY} ile bir env belirtmiş. Bu env başta boş oluyor eğer biz bu envi kullanmak istersek solutiona sağ tıklayıp Add kısmından new item diyoruz .env yazıyoruz ve .env dosyası oluşturuyoruz.

Buradaki services kısmında belirttiğimiz servis isimleri gizli domaindir demiştik. Docker içerisinde bir mikroservice bir başka mikroservise istek atmak istediği zaman base domain ismi işte burada belirttiğimiz servis isimleri oluyor. Normalde olan localhost yerine bu isimler geliyor. Hatırlarsak dockerda localle alakalı hiçbir şey olmamalıydı. Containerlar içinde çalışan mikroservisler birbirine istek atmak istediği zaman base domain olarak buradaki verdiğimiz isimleri kullanacaklar özetle. İşte vurada developtment ortamındayken bu localhost domaini oku product ortamındayken miroservice1,2 domainlerini oku gibi konfigürasyonlar yapacağız. Mesela ileride rabbitmq ‘ye bağlanıcaz rabbitmq diye bir servis ismimiz var buna bağlanmak istediğimizde bu servis ismi üzerinden bağlanıcaz önceden ise rabbitmq localhost:5601 portundan ayağa kalkardı mesela. Rabbiqmq:5601 olacak burası localhost yerine servis ismi gelecek.

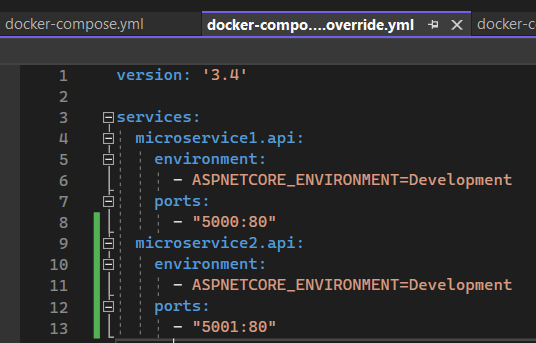




Burayı daha sonra dolduracağız. Arkasıdan bir build keywordumuz var. Context demek buraya . verirsek bizim projede docker-compse.yml dosyamız nerdeyse burası bir referans noktasıdır artık bu referans noktasına göre mikroservisimizin içerisindeki Dockerfile dosyasına gidebilir. Yani contextte docker-compose.yml dosyasını refrans olarak belirtiyoruz dockerfile keywordunda ise bu referans noktasına göre bu mikroservis projesinin dockerfile dosyası neredeyse o konumu orada belirtiyoruz. Bu referans noktası her zaman . olmalı. Şimdi bir microservis daha ekleyelim mesela. Ayrıca container ismide çok değişmez burası base olduğu için container ismini de burada belirtebilriiz. Ancak burada aynı pyhton gibi girinti çıkıntılara dikkat edelim kapsamlar buradaki boşluklarla belirleniyor YML dosyası olduğu için JSONdaki gibi süslü parantezler yok.



Birinci aşamayı tamamladık base’i yazdık bu base hiçbir zaman değişmeyecek ama ortama bağlı olarak değişecek olan override dosyasını güncelleyelim.



Bu dosyadada önce bir versiyon belirlenmiş sonrasında yine servis ismi belirleniyor ve içeirisnde bir docker environment belirleniyor bunu daha önce yapmıştık. Arkasından ise bu container içeirsinde çalışacak olan mikroserviceimiz hangi port üzerinden ayağa kalksın bunu belirtiyoruz. Bu noktada bunu port forward yaparak dışarıya da bu containeri port numarası olarak açabiliriz. Tamam 80 portunda docker içinde ayağa kalkacak docker dışında da bu containerda 5000 portu üzerinden erişebileyim mesela. Proje http olarak default olarak 80 portu üzerinden ayağa kalktığı içni buraya 80 diyoruz https olasydı default olarak 443 portundan ayağa kalkar. Dolayısıyla buradaki numaraları projenin ayağa kalktığı port ile aynı yapmak zorundayız ancak bu default olarak kullanılan portları değiştirmemiz mümkündür. Artık temel olarak dosyalarımız ayağa kalkmaya hazır. İleride farklı ortamlar için envler kullanıcaz envler bir uygulamayı restart etmeden o uygulamanın davranışını değiştirmemizi sağlıyor. Override içindeki tanımladığımız envler her zaman appsettings içindeki tanımlanan değerleri ezerler bunuda bilelim.

**BUİLD AŞAMASI**

Build komutu docker tarafındaki aynı işlemi gerçekleştiriyor yani belirtmiş olduğumuz dockerfilelara ve docker compose’a göre image oluşturuyor. Docker composeun konumunda terminalde docker compose build komutunu çalıştırmam yeterli. Tek bir komutla docker-compose dosyasında belirttimiz kadar mikroservisin kendi imageleri verdiğimiz isimlerle oluşacaktır. Ancak vs’da projey başlattığımız yerde docker-compose da kalmamasına dikkat edelim orada herhangi bir microservise’i seçebiliriz.

**3.KISIM CREATE VE START** imagelerimizi tek bir kod ile oluşturmuş olduk. Create komutu image oluştuysa containerlarımızı oluşturmak ile ilgili bir komuttu ama containerları çalıştırmak stop durumunda bırakır start ise containerlarımızı çalıştırmak için kullanacağımız komuttur. docker compose create bu kodu terminalde docker-compose dosyasının bulunduğu konumda yazdığım anda benim için oluşan imagelerimin containerlarını oluşturacaktır. Containerlarım oluştu ama ayakta değiller ayağa kaldırmak için ise yine terminale docker compose start demem yeterli.

**4.KISIM STOP VE RM** Stop çalışan containerlarımızı durdurur. Rm ise durmuş containerları silme işlemi yapar. docker compose stop diyebiliriz sonrasında ise docker compose rm ile silme işlemi yapabilriiz.

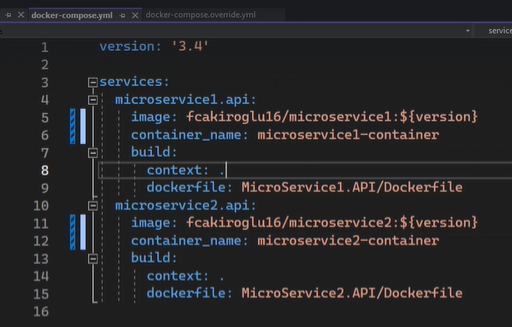
**5.KISIM UP (BUİLD,RECREATE,START) VE DOWN (STOP,REMOVE)** Up ve Down komutu en sık kullancağımız2 komut olacak. Up komutu imagelerimizi 0 dan build eder containerlarımızı 0 dan oluşturur ve containerlarımızı başlatır bu 3 işlemi birden sırayla yapar, Down komut uise containerları durdurur ve arkasından containerları siler yalnız sadece containerları değil up komutu ile ayağa kalkan bir volume varsa volumeları da siler veya bir network oluşturulmuşsa o networkuda siler. Yani up komutu ile beraber ne ayağa kalkmışsa down komutunda tam tersi işlem gerçekleşir sadece container olarak düşünmeyelim. Kullanmak için ise docker compose up diyebiliriz eğer burada control c dersek containerlarımız duracaktır ve docker compose down diyebiliriz. Eğer ayağa kalkan containerlara attach olma loglarını terminalde görmek istemiyorum dersek docker compose up -d komutunu kullanabiliriz.

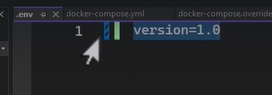
**6.KISIM PAUSE UNPAUSE VS STOP** bir container ayağa kalktığı zaman ram kullanıyor bu ramleri serbest bırakmıyor stop ise bir containeri stop yaptığınızda memory’i serbest bırakıyor. Yani eğer uygulamamız memoryde bir değer tutuyorsa stop yaptıktan sonra tekrar start yaparsanız o mermoyde tutulan değerlerin kaybolduğunu görürüz. Pause’da ise memoryde tutulan değerler kaybolmuyor tekrar unpause yaptığımız anda memoryde tutulan değerler ile çalışmaya devam ediyoruz. Yani stop memoryi serbest bırakıyor pause ise evet container duruyor ama memory hala tutuluyor aradaki fark bu. Sadece bir servicei durdurmamız docker compose stop serviceismi kodu ile mümkün.

**7.KISIM EXEC** Container içerisinde bir komut çalıştırammıza imkan sağlıyor. Daha çok bash komut satırına bağlanmak için kullanıyoruz. Komut satırına bağlanalım ki işletim sistemi üzerinde gerekli değişiklikler yapmak isteyebiliriz. Exec komutuyla container içerisindeki işletim sistemi üzerinde bir komutta çalıştırabiliriz ya da bash e bağlanarakta komut satırınada erişebiliriz. docker compose exec servisismi /bin/bash diyerek komut satırına bağlanabiliriz. Sonrasınsa pwd dediğim zaman olduğum klasörü söyler. Ya da bazen bu klasör içerisindeki dosyalarda değişiklik yapmamız gerekir o zaman da bir komut satırı editörüne ihtiyacım olabilir nano kullanabiliriz. Önce apt-get update diyelim sonra apt-get install nano diyerek kurulum yaparız yes deriz. Artık ls dediğimde dosyalarım gelir. Appsettigs.json u görmek istiyorum mesela nano appsettings.json diyorum ve içerisini bana editör gösteriyor. Bu şekilde o işletim sistemi üzerinde klasörleri dosyaları görüntüleyebiliyorum.

**8.KISIM SCALE** bu komut yardımıyla bir containerimizin birden fazla instance’ini tek seferde ayağa kaldırabiliyoruz ve sayılarınıda ayrı ayrı belirtebiliyoruz işte şu servisten 5 tane ayağa kalk gibi. Özellikle local tarafında test yaparken birden fazla instance ile uygulamamız gerçekten doğru mu çalışıyor çalışmıyor mu şeklinde kullanabileceğimiz güzel bir komut. Bu işlemi kullandığımızda docker containerlara kendisi isim atayacaktır. Bizim docker-compose dosyasında belirttiğimiz container ismi sadece tek bir container için geçerli oluyor. Port kısmında ise docker-compose dosyasında sadce :80 şeklinde kullanırsam docker compose oluşacak containerlar için boş bulduğu portları kendisi containerlara atayacaktır ya da istersem docker-compsoe dosyasında port aralığı belirtebilriim 5100-5200:80 burada 5100 ile 5200 arasındaki port numaralarını bu servis için kullan diyebiliyorum. Komutu kullanmak için ise docker compose up - -scale servisismi=3 bu şekilde kullanım yaparsak bu belirttiğimiz servisten 3 adet container ayağa kalkar diğer servislerden ise 1 tane ayağa kalkar belirtmediğimiz için ama şöyle de yapabiliriz docker compose up - -scale servisismi1=3 - -scale servisismi2=2

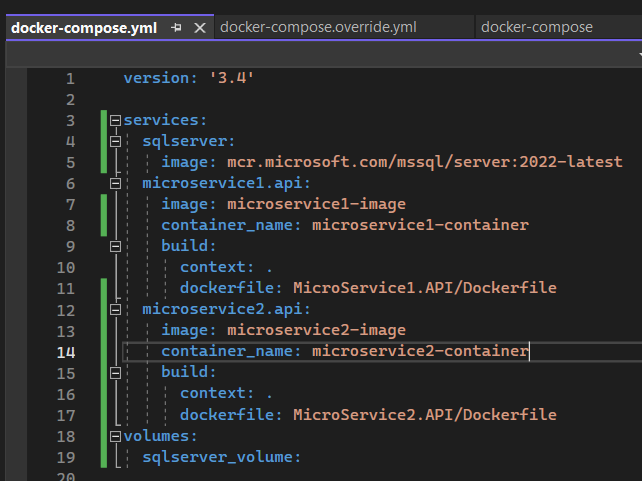
**9.KISIM PUSH** push komutuyla beraber harhangi bir container registery servisine bu imagelerimizi gönderebiliyoruz. Bu docker hub olabilir, azure container register servisi, amozon web container registery servisi olabilir farklı farklı servislere gönderebiliyoruz. Docker push dediğimde sadece bir imagemi gönderebiliyorum ama docker compose tarafında ne kadar image varsa tek bir komutla hepsini tek seferde gönderebiliyorum. Kullanım videoda var docker-compose dosyasında image isimlendirme alanlarında değişimler oluyor version’u env olarak vs ekliyoruz tag olarak image ismine veriyoruz. Ayrıca repository isminide ekliyorum. Docker hubda repository oluşturduktan sorna bu kullanımı yapıyorum.

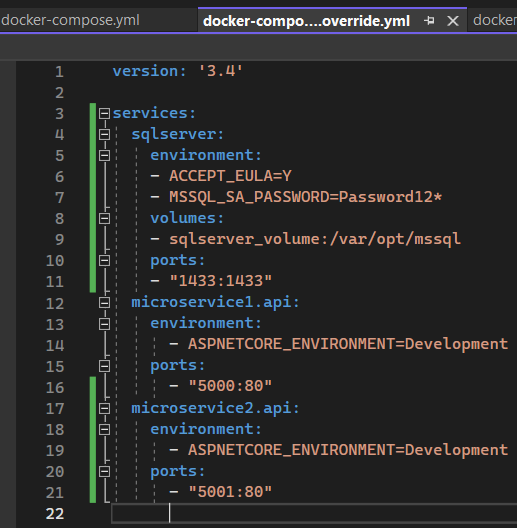




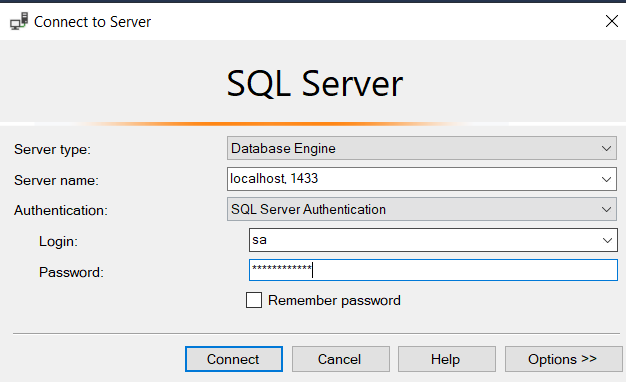
Buradaki fcakioglu16/microservie1 ve 2 ayrı farklı respoistorylerimin isimleri oluyor daha önce bunun kullanımı Dockerfile üzerinde yapmıştık aynı mantıkla çalışıyor. Daha sonrasında benim build alarak yeni imageleri oluşturmam gerekiyor. İmageler oluştuktan sonra önce docker logout ile çıkış yapıyorum docker login ile login olma işlemini gerçekleştiriyorum bilgilerimi doğru bir şekilde girerek. Sonrasında docker compose push komutunu çalıştırdığım anda imagelerim dockerhubtaki respositorylerime gidecektir. Biz oluşturmuş olduğumuz imagelarımızı container registery servislere atmalıyız ki DevOpscu arkadaşımız bu imageleri kübernet üzerinden çekebilsin.

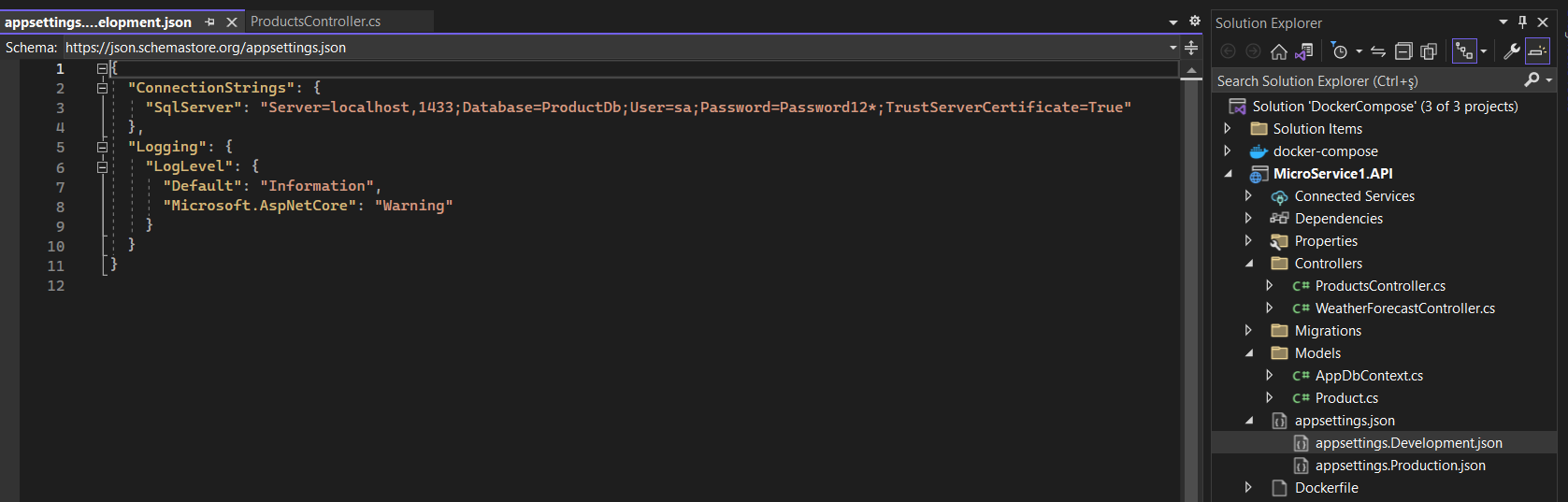
**SQLSERV ER 1.KSIIM SQL SERVER SERVİCE** Biz Docker compose içerisinde uygulamalarımızı ayağa kaldırdığımızda bu uygulamalarımız bir veritabanı ile nasıl haberleşebilir. Sql serveri bir container olarak ayağa kaldırıcaz. İlk olarak docker hub a gidip sql server imagesine bir bakalım. Gerekli envleri docker-compose override tarafında verelim ve bir volume tanımlayarak /var/opt/mssql yani bu konuma kaydedielcek database bilgileri kaybolmasın diye bu database verilerimizi volume içerisine atalım veriler kaybolmasın.



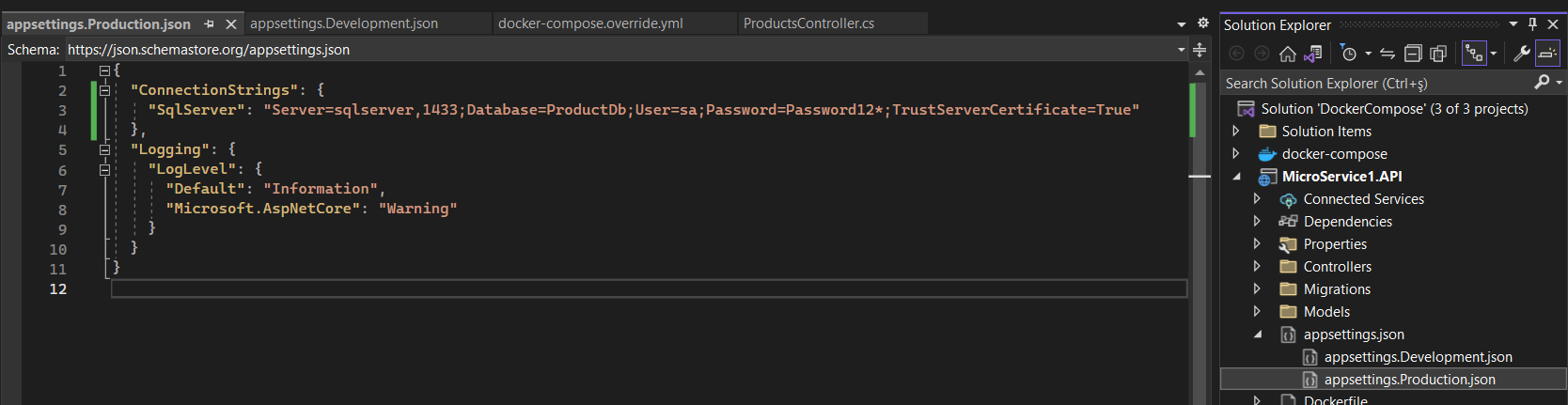


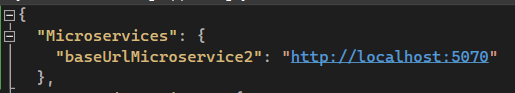
Ardından docker compose up -d sqlserver komutunu çalıştırarak sadece bu sqlserver imagesini oluşturup container olarak ayağa kaldıralım. Sonrasında sql server management studio yu açarak localhosttan bu dışarıya açtığımız port numarasına bağlanıyor ve sa diyip docker compose override tarafında belirlediğimiz passwordu yazarak veritabanına bağlanabiliyoruz. Bu noktadan sonra tüm istekler container içerisine gidiyor oluşturduğumuz dbler tablolar veya kayıtlar oluşturduğumuz volume’da depolanıyor. Containeri durdursak veya silsek baştan ayağa kaldırsak veya baştan oluştursak bile volume kullandığımız için datalar kaybolmayacktır yine dblerimizi ve tablolalarımızı görebileceğiz.



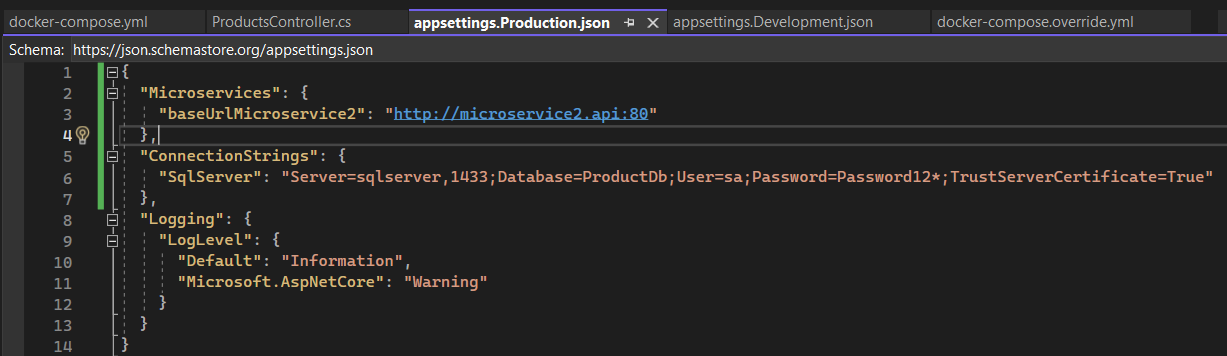
**SQL SERVER 2. KISIM** İlk olarak bilgisayarımızda çalışan uygulamalarımızı mikroservislerimizi docker container içerisinde çalışan sql servera nasıl bağlaanbiliriz? Sadece connection string ifademizde değişiklikler oluyor onun dışında nasıl efcore kullanarak sql bağlıyorduk tamamen aynı farklı hiçbir durum bulunmuyor. Development sırasında projemizi localden ayağa kaldıracağımız için bu connection string tanımlamasını developtment appsettings.json tarafında yaptık. Kendi localimizden projemizi kaldırıp docker içerisinde bir container içerisinde çalışan sql servera bağlanmış olduk.

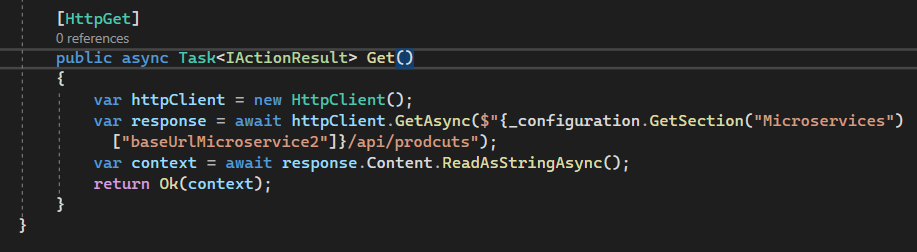
**SQL SERVER 3.KISIM** Bu sefer ise production ortam için çalışacağız. Docker içerisinde containerlarda ayağa kaldırdığımız uygulamaları nasıl yien docker içerisinde containerdan ayağa kalkmış olan sql servere bağlanırız bu işleme bakacağız. Docker compose içeriside localhost diye bir şey yok. Bunun yerine her containerin kendi içerisinde localhost diye bir kavram var. Yani bir mikroservis uygulamamı bulunduran container sql serveri çalıştıran container ile iletişim kurmak istediğinde localhost üzerinden gidemez çünkü localhost bizim domain namemiz bu noktada domain name işte bizim docker-compose dosyası içerisinde belirttiğimiz servis isimleri olacak. Bir uygulamayı publish ettiğimiz zaman bunun default ortamı production oluyor. Dolayısıyla uygulama production ortamından ayağa kalktığında production appsettings.json dosyasını okuyacak. Tek yaptığım değişiklik connection stringte localhost yerine db nin servis ismi olan sqlerver yazıyorum.



**TWO SERVİCE COMMUNİCATİONS** İki servisi birbiri ile nasıl iletişim kurdurabiliriz bunu inceleyeceğiz. Eğer bu işi localhostta yapıyorsak developtment appsettings.json tarafındaki kodumuz bu şekilde olur

Ama eğer projeleri localhost üzerinden değilde docker containerlar ile ayağa kaldırıp birbiri ile haberleştirmek istersek yani production ortamında olacağı için nu işlemi production appsettings.json tarafında servis isimler ile bu şekilde yapmalyıız localhost yerine. Rabbitmq kullanımı ise aynı mantık önce rabbitmqyu containerdan ayağa kaldırıyoruz sonrasında burada olduğu gibi rabbitmqnun servis ismini ve port nuamrasını vererek böyle bu url üzerinden ona bağlanıyoruz production ortamda ama development ortamda yine rabbitmqnun localhost adresi üzerinden bağlanabiliriz. Bu durum redis ve diğerleri içinde aynı unutmayın servis isimleri bizim domain isimlermiz olarak geçiyor production ortamda localhost yerine.



****