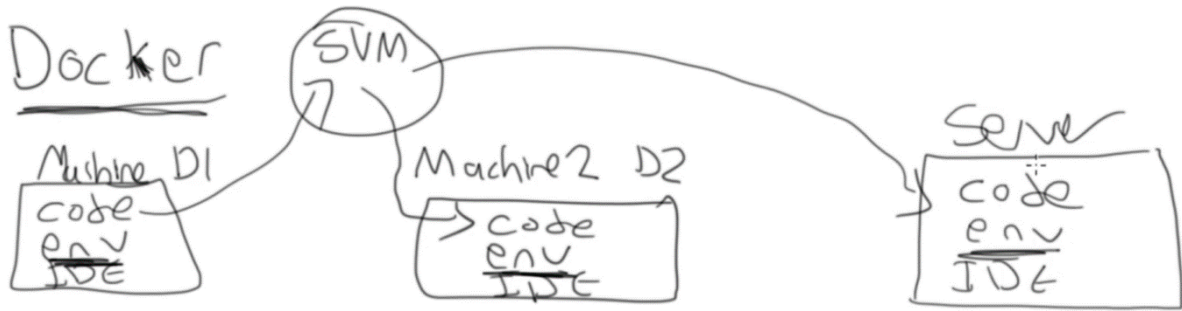


## WHAT IS DOCKER AND WHY USE IT?

Software development için önemli bir problem söz konusuydu: "It works on my machine but it doesn't work on the other machine/production machine"

Örneğin bir developer'ın bir machine'ı var bu machine üzerinde code'u, environment'ı, IDE'si vesaire var, her şey tamam program çalışıyor, şimdi diyor ki biz bu programın code'unu bir remote repository üzerinden bir başka machine'e aktaralım, bu machine bir başka developer da olabilir veya production server da olabilir.

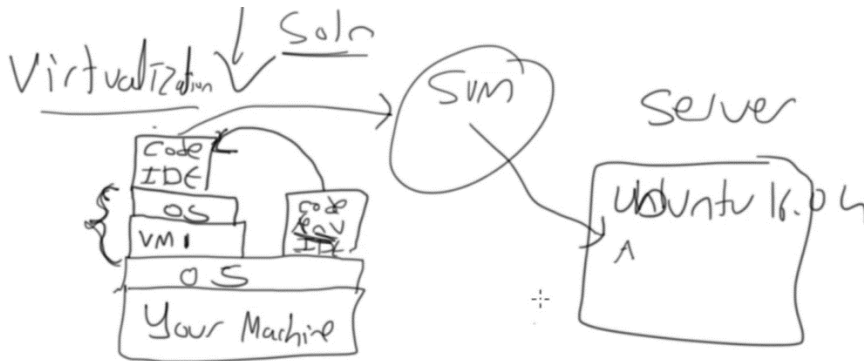
Ancak bu machine2'nin environment'ı, ide'si vesaire machine1 ile aynı olmadığı için kod aynı olsa da, programı çalıştırmak mümkün olmuyor, bir sürü hata söz konusu oluyor, şu library eksik, bu syntax error, vesaire.



Bu problemin temel sebebi environment'ların farklı olması, environment ile kastımız: all the libraries on the system, operating system, everything on that machine. Dolayısıyla iki environment'ın aynı olmasını sağlayamıyoruz bu da problemlere neden oluyor.

Biz istiyoruz ki bir makinede çalışan kod, istediğimiz bir başka makinede de çalışsın, bunun için ilk akla gelen çözüm: **virtualization**di.

Burada yapılan şu, programın hangi server'da çalışacağını önceden biliyoruz bu server'ın environment bilgilerini elde ediyoruz, daha sonra developer machine üzerine virtual box gibi bir programla bir virtual machine kurulur, onun üzerine server'ın OS'i ve server'ın environment'ı set edilir, code burada geliştirilir. Böylece developer machine üzerinde, production machine ile identical bir virtual machine setup kurmuş oluruz, bu virtual machine üzerinde çalışan programlar production server üzerinde de çalışmak zorundadır!

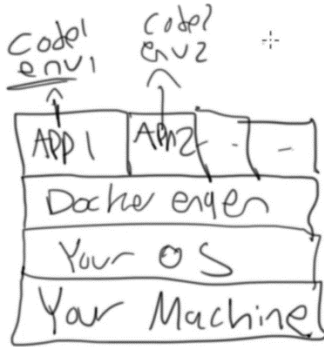


Bu yaklaşımın dezavantajı very resource heavy olması, kodun yazıldığı yer ile hardware arasında çok fazla layer var, kod çok fazla dönüştürülüyor, bu da development sırasında yavaşlığa ve inefficiency'e neden olur.

---

Peki bu durumu nasıl çözebiliriz? **DOCKER**.

Docker yeni bir fikir değil, ancak son birkaç yılda çok popüler oldu. Docker sayesinde, os üzerine bir docker engine layer yerleşir ve bu layer sayesinde farklı code'ları farklı environments üzerinde çalıştırabiliriz.



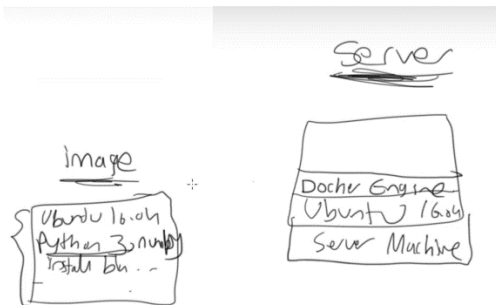
Docker engine operating system ve apps arasında bir translator gibi davranarak işleri hızlandırır ve bize farklı uygulamaları farklı environments ile çalıştırma imkanı verir.

---

Burada biraz terminology konuşalım, şekilde görülen app'lere app demeyiz de **CONTAINER** deriz, çünkü bunlar mutlaka application olmak zorunda değil, belki bunlardan birinde bir database de çalışabilir, yani container daha genel bir isim.

Container is made out of an **IMAGE**. Image'ı ilgili container'ın run edileceği environment bilgilerini tutan bir text file gibi düşünebiliriz. Örneğin app1 için diyebiliriz ki bu container'daki uygulama ubuntu 16.04 OS üzerinde çalışacak, python3 yüklü olsun, onun yanında şu şu paketler install edilsin vesaire, işte tüm bu environment bilgileri image içerisinde text olarak tutulacak.

Sonuç olarak programı deploy edeceğimiz zaman, server üzerinde docker yükleriz. Elimizde app1 container'ı için image da var, yani tüm gerekli environment bilgileri var, o halde bu image'ı server'a pass edersek, server üzerinde de aynı environment ile container'ı oluşturabilir ve aynı app'i server üzerinde sorunsuz şekilde run edebiliriz.



## DOCKERIZING OUR FIRST APPLICATION

Daha önceki kısımlarda, dört işlem yapabilen basit bir restful api yapmıştık, bu api için kullandığımız app.py dosyası masaüstünde api\_1 klasörü altında duruyor. Şimdi bu flask uygulamasını, dockerize edeceğiz. Yani yapmak istediğimiz şey, bu uygulamayı doğrudan kendi environment'ımızı(ubuntu, vesaire) üzerinde çalıştırmak yerine, docker'ı kullanarak istediğimiz kritlerde(yine ubuntu, python3, flask, flask\_restful olan) bir environment üzerinde çalıştırmak, yani bir container yaratıp orada çalıştırmak. Bu bize ne sağlayacak? Şunu sağlayacak ki oluşturulan projeyi kolay bir şekilde bir başka makine üzerinde docker kullanarak çalıştırabileceğiz.

Docker Hub sitesi üzerinde, farklı sistemleri describe eden bir çok docker images store edilir. Örneğin biz kendi oluşturacağımız sistem üzerinde python olsun istiyoruz, official python docker image'ını burada bulabiliriz. Dockerfile'ı yazarken, doğrudan python' pull et diyeceğiz, işte pull ettiği yer dockerhub sitesi olacak.

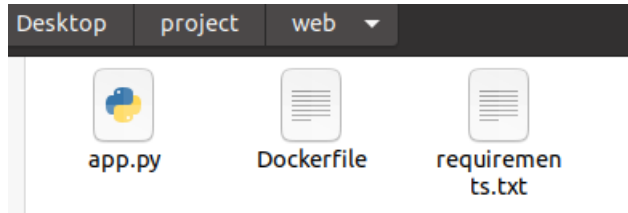
Şimdi daha önce yazdığımız web service'imizi dockerize etmeye başlayalım. Bunun için ilk olarak ayrı bir klasör oluşturdum ismine **project** dedim, şimdi web service app'imi bir container'da çalıştıracığım için project altına **web** isminde yeni bir klasör açıyorum ve bu **web** klasörü içine daha önce hazırladığım app.py dosyasını yani flask uygulamasını kopyalıyorum.

Eğer farklı container'lar da olsun istersem yeni bir klasör açarım örneğin database için **db** isminde yeni bir klasör de açtık, şuan kullanmayacağız. Şuan sadece web container'ına odaklanacağız.

Şimdi **web** container'ı ile çalışacağız, bu klasörün içine konumlanıyoruz, burada şuan sadece **app.py** dosyası var, aşağıdaki komutlarla web klasörü içerisinde bir **Dockerfile** ve **requirements.txt** oluşturuyoruz:

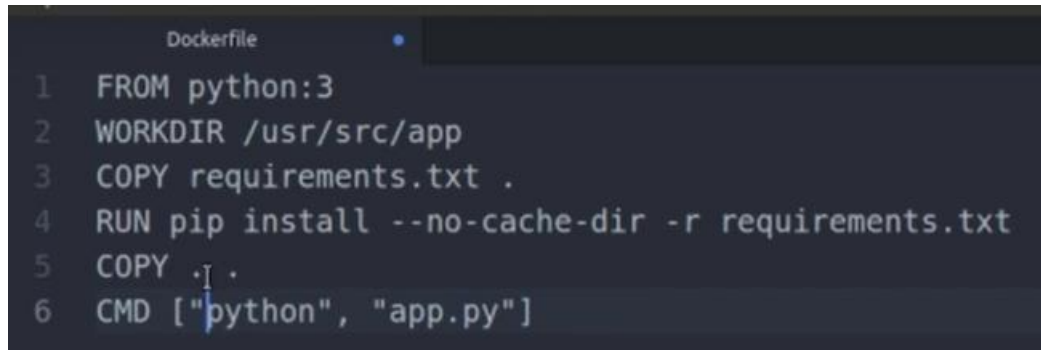
```
touch Dockerfile
touch requirements.txt
```

Sonuçta project > web altında aşağıdaki üç file yer alıyor.



## DOCKERFILE

Şimdi uygulamamızı dockerize etmek için gerekli olan Dockerfile içeriğini dolduralım ve satır satır açıklayalım:



```
1 FROM python:3
2 WORKDIR /usr/src/app
3 COPY requirements.txt .
4 RUN pip install --no-cache-dir -r requirements.txt
5 COPY .I .
6 CMD ["python", "app.py"]
```

Dockerfile'ı ilgili container'ı (web ismindeki) set etmek için yazılan bir takım talimatlar olarak düşünebiliriz. Kendi işletim sistemimiz üzerinde kurulacak container'ın kullanacağı environment özelliklerini burada belirtiyorum.

Bizim burada yapmaya çalıştığımız, docker kullanarak kendi işletim sistemimiz üzerinde istediğimiz bir environment ile container oluşturmak ve web app'imizi bu container üzerinde çalıştırmak. Container'ı başka bir bağımsız machine gibi düşünebiliriz. Yani container'ı set ederken sanki bağımsız bir machine'ı set ediyoruz gibi düşün.

### 1.

İlk satırda söylenen şey şu, bu container machine için python:3 image'ını kullan. Docker bu image'ı nerden indiriyor peki? Docker hub sitesinden. Ayrıca şu önemli python:3 image'ı içerisinde bir çok daha başka images da içeriyor örneğin, ubuntu image'ını da içeriyor, bu yüzden bir OS belirtmeden direkt python:3 istedik.

Python:3 image'ını kullandığımızda bu container için gerekli bir çok environment ögesi (işletim sistemi vb.) zaten along with python 3 set edilmiş oluyor.

Yani ilk satır docker tarafından execute edildiğinde artık web container'ı ubuntu işletim sistemini kullanıyor ve python 3 yüklenmiş şekilde hazır hale gelecek. Bomboş bir machine'i bu instructions ile set ediyoruz, diyoruz ki sen bu container için bu machine'e ubuntu indir, üzerine de python3 kur.

### 2.

İkinci satırda container machine için working directory set ediyoruz. Burada seçilen working directory başka bir yeri de gösterebilir tamamen convention.

### 3.

Container machine üzerinde ubuntu ve python3 kurulu olsa da şuan bu machine üzerinde flask app çalıştıramayız, çünkü flask ve flask\_restful paketlerini install etmedik.

Hatırlarsan kendi bilgisayarımızda da flask app run etmeden önce, python 3 yükledik sonra gerekli paketleri pip3 ile install ettik ancak daha sonra flask app'i run edebildik.

İşte bu sebeple, container machine üzerinde de aynı şeyleri yapmamız gerek, bu direktifleri burada veriyoruz.

Diyoruz ki, dockerfile'ın hemen yanında duran requirements.txt dosyasını al, kopyala ve container machine'in working directory'sine yapıştır.

Yani buradaki web klasörünü bir blueprint olarak kullanıyoruz, bu klasör sayesinde, docker container machine'i oluşturacak ve ilgili uygulamayı container üzerinde çalıştıracak.

### 4.

Bir önceki adımda, requirements.txt dosyasını container machine'in working dir'ine yapıştırmıştık, şimdi de diyoruz ki container machine pip3 kullanarak bu requirements.txt dosyasının içerisindeki paketleri install etsin.

### 5.

Bu adımda, current directory'deki dosyalar (app.py ve requirements.txt) container machine'in working directory'sine kopyalanacak.

App.py'ın zaten kopyalanması gerek çünkü sonuçta flask app olarak run edilecek file bu file, ancak requirements'ı zaten kopyalamıştık sanki neden tekrar kopyaladık tam anlamadım. Şimdilik bunu sadece app.py dosyası container machine'e taşınyormuş gibi düşün.

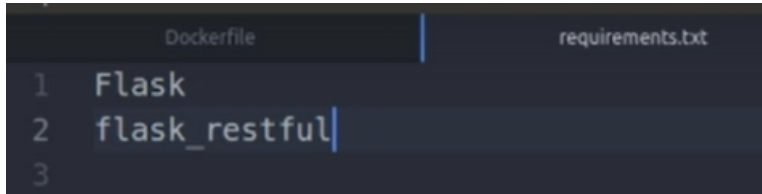
### 6.

Son olarak CMD kullanarak docker'a bir command run etmesini söylüyoruz, diyoruz ki container machine üzerinde python uygulaması, app.py isimli dosyayı run etsin.

Artık web klasörü altında Dockerfile'ımız hazır, bu dockerfile içerisinde bir container machine'in nasıl set edileceği, hangi dosyaları nereye taşıyacağı ve hangi flask app'i run edeceği yazıyor.

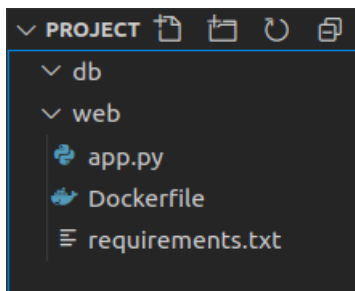
## REQUIREMENTS

Şimdi localdeki requirements.txt dosyası içerisine, ilgili container machine'in flask app'i run edebilmesi için gerekli packages'i yazacağız. Container machine set edilirken bu packages install edilecek.



```
1 Flask
2 flask_restful
3
```

Şuanda project klasörümüzün içi aşağıdaki gibi, iki farklı container var, ilki db container'ı ancak şuan bu boş daha sonra kullanılacak. Diğerisi ise, web app'imizi (dört işlem api'ı) çalıştıracığımız container, bu container'ı set etmek ve ilgili flask app'ini çalıştırmak için gerekli dosyalar ve instruction'lar burada yer alıyor.

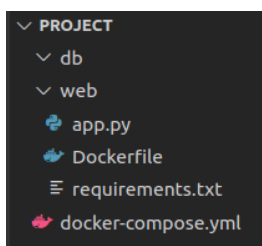


## DOCKER COMPOSE

Şimdi yapmak istediğimiz, web klasörünün Dockerfile'ı içerisinde belirtilen instructions'ı otomatik olarak uygulayacak ve bir web container'ı oluşturarak bunun içerisinde app.py flask uygulamasını çalıştıracak bir yöntem oluşturmak.

Bu amaçla, project klasörü içerisinde db ve web ile aynı seviyede bir **docker-compose.yml** dosyası oluşturuyoruz.

```
touch docker-compose.yml
```



Bu yml dosyasını farklı containers'ı kontrol edip aralarında bir iletişim sağlayan bir yapı olarak düşünebiliriz, hangi container'ın run edileceği, hangi port üzerinde run edileceği vesaire gibi bilgileri bu docker-compose file'ı içerisinde veririz.

Şuan için sadece tek bir container'ımız (service'imiz olacak) ancak birden fazla service'imiz olsaydı bunları docker-compose file'ından aşağıdaki gibi kontrol edebilirdik.

```
docker-compose.yml
1  version: '3'
2
3  services:
4    web:
5      build: ./web
6      ports:
7        - "5000:5000"
8    db:
9      build: ./db
10   web2:
11     build: ./web2
```

Satır satır inceleyelim,

İlk satırda, version : '3' olarak belirttik, bu docker-compose version'ıdır, farklı versiyonları var, syntax değişebilir.

Daha sonra her bir service'i (container'ı) hangi directory'den ve hangi port üzerinden build edebileceğimizi burada belirtiyoruz.

Docker compose web service'ini build etmek için ./web lokasyonu içerisindeki Dockerfile'ı kullanacak.

Port için de anladığım kadarıyla olay şu, container machine içerisinde 5000 portunu açıyoruz ve bunu da local machine'in 5000 port'u ile bağlıyoruz.

Docker compose kullanarak, daha farklı container'larda çalışan farklı app'lerin de tek seferde build edebiliriz, complex uygulamalar için gayet pratik bir çözüm. Ancak şimdilik sadece web service'i kullanılacak db ve web2'yi söylebiliriz.

## BUILDING THE WEB APP

Şimdi yapacağımız işlem docker-compose.yml dosyasının olduğu directory içerisindeyken, bu dosyayı kullanarak burada belirtilen service'leri build etmek, bunun için aşağıdaki satırı kullanıyoruz:

```
sudo docker-compose build
```

Bu satırla birlikte, ilgili services (şuanda sadece web) için gerekli containers Dockerfiles kullanılarak build ediliyor.

**ANCAK BU İŞLEMİ YAPMADAN ÖNCE ÖNEMLİ BİR DÜZELTME YAPMAMIZ GEREK, APP.PY İÇERİSİNDE HOST'U AŞAĞIDAKİ GİBİ BELİRTMELİYİZ.**

```
if __name__ == "__main__":  
    app.run(host='0.0.0.0')
```

Anladığım kadarıyla bunun sebebi şu, eğer bunu eskisi gibi bırakırsak service container'ın 127.0.0.1 host'u üzerinde çalıştırılacak ve bu host'a sadece container içerisinde ulaşılabilir, biz ise postman ile request gönderirken container içerisinde değil, local computer üzerinden göndereceğiz, bu host'u 0.0.0.0 set ederek, uygulamaya public interface'i dinlemesi gerektiğini söylüyoruz, böylece dışarıdan gelen request'lerde uygulamaya ulaşabilir.

So why you need to put host='0.0.0.0'. It is because if in the application you put:

`app.run(host='127.0.0.1')` then the application will listen to the container's localhost so nothing from outside can access the app. So from inside the container you would be able to access the app.

But by saying `app.run(host='0.0.0.0')` you are telling the app to listen on the public interface. Meaning things from outside the container can access it.



## RUNNING THE BUILT CONTAINERS/SERVICES

Artık services'i çalıştıracak containers hazır, sıradaki işlem bu services'i run etmek, yine docker-compose.yaml ile aynı seviyedeyseniz:

```
sudo docker-compose up
```

Artık, ilgili services (şuan için sadece web service) çalışıyor ve daha önce olduğu gibi localhost:5000 bu service tarafından dinleniyor, buraya gönderilecek request'ler service tarafından handle edilecektir.

Artık web service'imiz container'ın 0.0.0.0 host'unda çalışıyor, bu hosta dışarıdan da ulaşabiliyoruz.

```
Attaching to project_web_1
web_1 | * Serving Flask app "app" (lazy loading)
web_1 | * Environment: production
web_1 | WARNING: This is a development server. Do not use it in a p
oduction deployment.
web_1 | Use a production WSGI server instead.
web_1 | * Debug mode: off
web_1 | * Running on http://0.0.0.0:5000/ (Press CTRL+C to quit)
```

Postman kullanarak localhost:5000'e gönderilen request'ler web container'ı üzerinde çalışan service tarafından handle ediliyor.

Burada localhost:5000'e gönderilen request'in nasıl container'ın 0.0.0.0 host'una yönlendiğini tam olarak anlayamadım, ama çalışıyor.

Eğer app.py üzerinde bir değişiklik yapmak istersek, run işlemini CTRL+C ile durdurup, değişikliği yapıp tekrardan

```
sudo docker-compose up
```

Diyebiliriz, dockerfile değiştirmediysek veya yeni bir dosya vesaire eklemediyse tekrar build etmek şart değil.