**Introduction to Containerization Project**

Containerization has emerged as a transformative solution in modern software development, offering unparalleled efficiency, scalability, and portability. In this project, we embark on a journey to containerize an existing application, exploring the pivotal role of container technologies in revolutionizing deployment practices. By delving into real-world use cases, market insights, and practical demonstrations, we unravel the power of containers to optimize resource utilization, ensure consistency across environments, and streamline the deployment process. Join us as we navigate through the intricacies of Docker and embark on a hands-on exploration of containerization principles and practices. The focus of this project will be containerization of an existing Java application.

**Advantages of Containerization**

* Cost savings through resource optimization.
* Portability and consistency across environments.
* Repeatability and ease of deployment.

**Tools and Prerequisites**

* Docker as the container runtime environment.
* Docker Compose and Docker Hub for image management.
* Understanding of containers and Docker.
* Hands-on experience with Docker basics.

**Understanding the Scenario**

* Management of multi-tier application stacks.
* Challenges with traditional deployment methods.
* Resource wastage and operational expenses.
* Human errors in deployments.

**Project Steps**

1. Setting up the V-profile stack manually.
2. Finding base images from Docker Hub.
3. Fetching source code from Git repository.
4. Writing Dockerfiles for customization
5. Pushing custom images to Docker Hub.
6. Building and testing container images.
7. Hosting custom images on Docker Hub.
8. Testing containers with Docker Compose.

**Architectural Design**

A diagram of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screen shot of a computer

Description automatically generated

A screen shot of a computer program

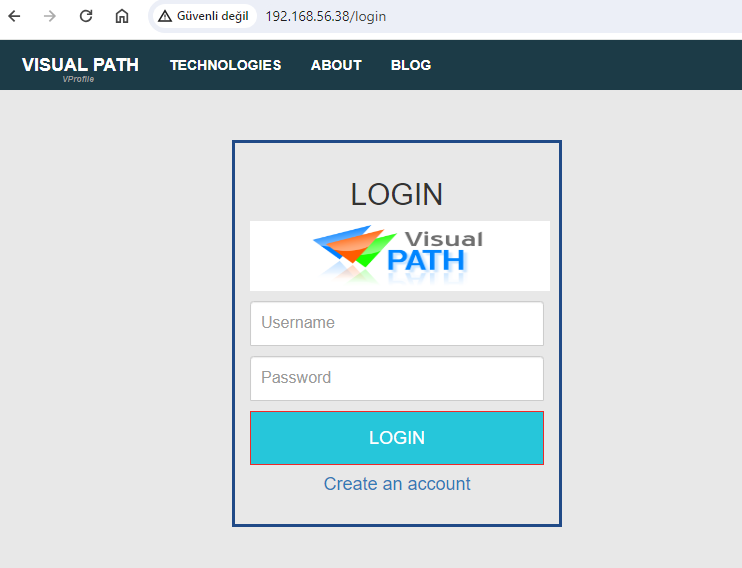
Description automatically generated

A black background with white text

Description automatically generated

A screenshot of a computer

Description automatically generated



A screenshot of a website

Description automatically generated

A close-up of a white background

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer screen

Description automatically generated

A screenshot of a computer

Description automatically generated

A screenshot of a computer

Description automatically generated

A white text with black text

Description automatically generated

A diagram of a docker build

Description automatically generated

A screen shot of a computer

Description automatically generated

**MICROSERVICES CONTAINERIZATION**

A diagram of software development

Description automatically generated

Bu projede bir microservice uygulamasını nasıl containerize hale getirebileceğimizi göreceğiz. Bir DevOps olarak microservice uygulamalarını ve öncelikle onu nasıl containerize edeceğinizi bilmelisiniz. Bu yapıda toplam dört hizmetimiz var. API GW olarak Nginx, tüm isteklerin geldiği frontend konumundadır ve microservice’ler arasındaki tüm iletişim bu API ağ geçidi aracılığıyla gerçekleşir. NginX, bağlantılarda gördüğünüz URL'lere göre (/, /api, /webapi) isteği dinleyecektir. Yani istek hangi rota üzerinden gelirse, mesela / URL'den gelen istekler, Angular'da yazılmış olan istemci microservice’e gönderilir. Bu, web sitesinin frontend verilerini yükler. Backend verileri için, NodeJS'de yazılmış olan API hizmeti mevcuttur. Bu API'lerin veritabanı ihtiyacı için burada bir NoSQL veritabanı olan MongoDB kullanıyoruz. Java’da yazılmış diğer servis ve bir SQL veritabanı olan MySQL veritabanını da /webapi’den gelen isteklerde kullanıyor olacağız.

Bu uygulama birden fazla microservice olan bir e-ticaret uygulamasıdır. Öyleyse GitHub'daki kaynak koduna gidelim. Tüm microservice’lerin kaynak kodu burada tek bir depodadır. Bu tür depolara mono repo adı verilmektedir. Gereksinime bağlı olarak, client için ayrı, Java API için ayrı, NodeJS API'si için ayrı ve Nginx için ayrı bir depoya sahip olacağınız gibi birden fazla depo kullanılabilir. Bu, tüm bunlar için ayrı CI/CD pipeline’lar oluşturmak açısından çok faydalıdır. Burada microservice stack’inin nasıl containerize edileceğini incelediğimizden hepsini tek bir alanda depoladık. Aşağıda daha net anlaşılması için dosya yapımızı görebilirsiniz.

A screenshot of a computer

Description automatically generated

Teker teker ilerleyelim, aşağıda client için oluşturulmuş dockerfile ‘ımızı görebilirsiniz. Bu multi-stage bir docker dosyasıdır. İlk aşama artifacts’leri oluşturmak, ikinci aşama ise bunları asıl image’e kopyalamak için kullanılıyor. Bu sayede gereksiz yükleri ilk aşamada bırakarak image’imizin boyutunu küçültüyoruz.

A computer screen shot of a program

Description automatically generated

Dockerhub’dan uygulamamız için bir nodejs image’i (node:14) bulalım. Çalışma dizinimize geçerek kaynak kodunu /client dizinine kopyalayalım. İlk aşama build edildiğinde kodumuz usr/src/app/client dizininde oluşacaktır. Client dizinine giderek NPM install ve npm run komutlarını çalıştırarak build aşamasını tamamlıyoruz.

Burada Node ve Angular hakkında hiçbir fikriniz olmayabilir. Burada önemli olan yapıyı nasıl derleyeceğinizi bilmektir. Sadece bir projeye her girdiğinizde, yapıyı oluşturmak için sürecin geliştiricisinden bilgi almanız gerektiğini unutmayın. Bu Docker dosyası derleme adımlarını içerir. Uygulamayı nasıl oluşturacağınızı bilmelisiniz. Yani kaynak koddan çıktıyı nasıl alacaksınız?

Artık artifact’lerimiz client dizininde oluşmuştur. Bunu asıl image’e kopyalamamız gerekiyor. Build komutunu çalıştırdığınızda, dist client'taki yapıt nerede olacak ve bu yolda bu nginx görüntüsüne koyacağınız HTML dosyaları nerede olacak.

Neden bu yol? Çünkü nginx görüntüsü bu şekilde oluşturuldu. Dolayısıyla, Docker hub'ındaki nginx görüntüsünün resmi belgelerini okursanız, HTML dosyanızı bu dizine yerleştirmeniz gerektiğini göreceksiniz. Ayrıca Nginx yapılandırma dosyasını da bu yola kopyalıyoruz. Bu yüzden kendi özelleştirilmiş konfigürasyonumuza ihtiyacımız var.

A screenshot of a computer program

Description automatically generated

O halde yukarıda bulunan nginx.conf dosyasını kontrol edelim. Bu yapılandırma dosyası basitçe, birisinin bu nginx container’ına root dizininden erişmesi durumunda /usr/share/nginx/html dizindeki index.html ve index.htl dosyalarının tarayıcıya yükleneceğini söylüyor. Yani eserinizde temel olarak nginx container’ının kullanıcıya sunduğu bu index.html dosyasına sahip olacaksınız.

A screen shot of a computer program

Description automatically generated

Nodeapi’ nin dockerfile dosyasına gidelim. Yine Node.js üzerinde artifactlerin oluşturulduğunu görüyoruz. Bir öncekiyle benzer süreçle çalışma dizinini kuruyoruz, kaynak kodunu geçerli dizinden kopyalıyoruz. Böylece tüm kaynak kodu nodeapi dizinine kopyalanacaktır. Bu dizine gidin ve NPM install komutunu çalıştırın. Bu komut artifact’leri oluşturacaktır. Image’I 5000 portundan expose ediyoruz. Node uygulamamızı çalıştırmak için bu komutu ( CMD[“….”] satırı) çalıştıracağız. Ve nodejs uygulamanızı başlatacak olan npm start komutunu çalıştırın. Bu da ikinci microservice’in Docker dosyasıydı.

A screen shot of a computer

Description automatically generated

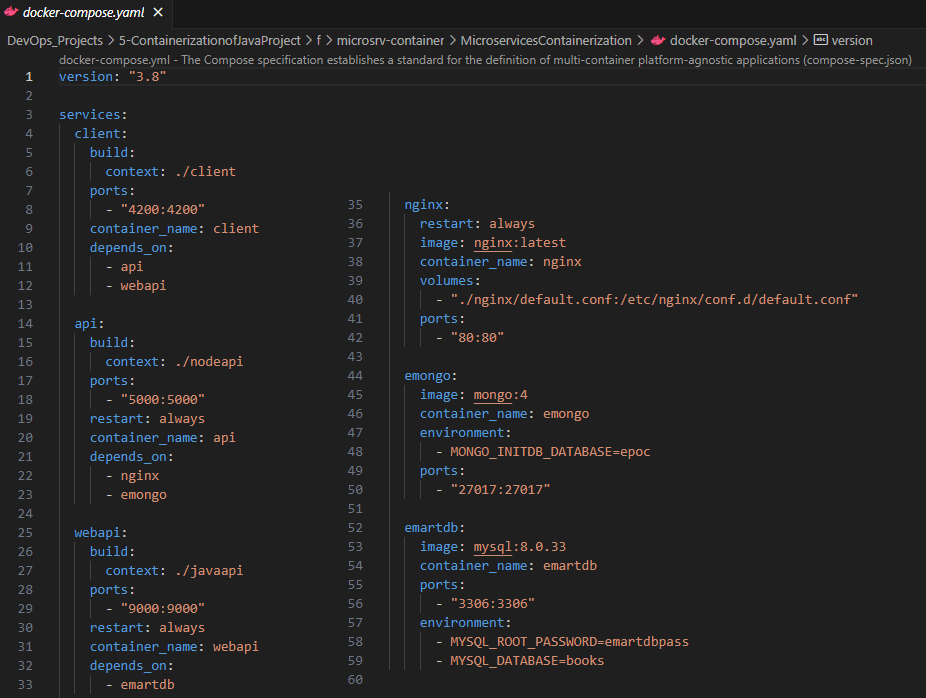
Şimdi Java API'sine geçelim. Java dosyalarını build etmek için Maven'e ihtiyacımız olduğu için dockerhub’dan uygun bir openjdk image’i buluyoruz. Çalışma dizinimizi ayarlayıp maven'i yüklemek için apt update ve apt install maven'i çalıştırıyoruz. Kaynak kodunu /usr/src/app dizinine kopyalıyoruz ve artifact’leri oluşturacak mvn install komutunu çalıştırıyoruz. Tekrar JDK’nın çalışma dizinini açalım ve artifact’leri kopyalayalım. Port 9000 üzerinden expose edelim. Java Jar komutunu ve Java uygulamasını çalıştıracak komutu giriyoruz.

A screen shot of a computer program

Description automatically generated

Burada tüm yapıyı yönetecek API Gateway’imizi yapılandırıyoruz. Burada resmi nginx imajını kullanabiliriz. Burada dockerfile’a gerek yok sadece konfigürasyonumuz (default.conf) var. Bu default.conf dosyasını bir birim olarak ekleyeceğiz, böylece ayrı bir nginx görüntüsü oluşturmamıza gerek kalmayacak çünkü tek ihtiyacımız olan bu yapılandırmanın aktif edilmesidir.

Yapılandırmanın isteğin geldiği path’lere göre yönlendirmeleri yapacak şekilde olduğuna dikkat edin. Yani bu konfigürasyonun temelde üç mikro hizmetimiz için üç yönlendirme kuralı vardır.



Son olarak servisleri yapılandırmak için Docker Compose dosyamıza geliyoruz. Dört servisin yanı sıra iki adet veritabanı var. VHepsini bir arada çalıştırmak veya bir araya getirmek istediğinizde Docker compose'a ihtiyacınız var. Burada servisler container’lar anlamına gelir. Daha önceki projelerde dockercompose dosyasını incelediğimiz için burada ayrıca tek tek açıklamıyoruz. Sadece depends\_on anahtarının, bu servisi oluşturmadan önce bağımlı olduğu diğer servisi çalıştır anlamına geldiğini belirtelim. Daha önceki adımlarda gösterdiğimiz context dizinlerini burada her container için tanımlıyoruz.

Burada verilen dosyaların oluşturulması konusunda deneyiminiz varsa, bunu çok hızlı bir şekilde yapacaksınız. Bu konuda deneyiminiz yoksa geliştiricilerle çalışmanız gerekir. Derleme süreçlerini ve kodun nasıl host edileceini anlayın. İşinizin Docker dosyası yazmak olacağını öğrendikten sonra Dockercompose, hepsini bir arada çalıştırmak için dosya oluşturur ve bu zaman alır. Bir projede bunu gerçek zamanlı olarak yaparken, inşa ederken ve çalıştırırken pek çok hatayla karşılaşabilirsiniz. Yani bunu yaparken, sıfırdan yazarken sabırlı olun. Acele etmeyin. Geliştiricilerle çalışın. Docker dosyalarını inceleyin, oluşturma dosyasını inceleyin, her seçeneği anlayın.