Slide 1: Introduction to Docker

დღეს ვისაუბრებთ ერთ ერთ მნიშვნელოვან და ძლიერ ხელსაწყოზე, რომელსაც Docker ეწოდება. დოკერი პლატფორმაა რომელიც საშუალებას გვაძლევს შევკრათ ერთ პაკეტში ჩვენი აპლიკაცია და ის ყველაფერი რაზეც ჩვენი აპლიკაცია დამოკიდებულია. ასეთი კონტეინერი გვაძლევს საშუალებას დარწმუნებულები ვიყოთ, რომ ჩვენი აპლიკაცია ყველანაირ გარემოში ერთნაირად იმუშავებს. Docker ის დახმარებით ამ ერთიანად შეკრული პაკეტების სხავდასხვა გარემოში გადატანა ძალიან მარტივია.

დოკერი ხელსაწყოა რომელიც ამზადებს და უშვებს კონტეინერს. კონტეინერები პატარა ყუთებივითაა, რომლებიც ყველაფერს შეიცავენ რაც აპლიკაციის ასამუშავებლად საჭიროა. ანუ კონტეინერში გვაქვს თავად აპლიკაცია, საჭირო ბიბლიოთეკები, კოფნიგურაციის ფაილები და სხვა დამოკიდებულებები. ყველა კონტეინერი ერთი OS ის სერვისებს იყენებს. ვინაიდან როგორც ვთქვით docker image შეიცავს იმ ყველაფრის აღაწერას რაც საჭიროა კონტეინერში აპლიკაციის ფუნქციონირებისთვის, შესაბამისად ამ ჩვენს კონტეინერებს თუ სხვა დოკერის გარემოში გადავიტანთ (სხვა მანქანაზე სადაც დოკერია) მაინც ყველაფერი უპრობლემოდ იმუშავებს.

Docker იყენებს ოპერაციული სისტემის დონის ვლირტუალიზაციას კონტეინერების იზოლაციისთვის. ეს გულისხმობს იმას, რომ დოკერი კერნელის რესურსების იზოლაციას იყენებს იმისათვის რომ რამდენიმე კონტეინერი ერთ მანქანაზე გაუშვას. ეს მიდგომა ტრადიციული ვირტუალური მანქანებისგან განსხვავდება იმით რომ ტრადიციულ შემთხვევაში სათითაოდ ყოველ ვირტუალურ მანქანაზე იყო საჭირო ოპერაციული სისტემის გაშვება და იქ კოდის მოთავსება. OS Level Virtualization ის შედეგად თითოეულ კონტეინერში გაშვებულ აპლიკაციას უნიკალური, თავისი, გარემო აქვს ისევე როგორც VM ებშ ექნებოდა, მაგრამ ბევრად მეტი რესურსით.

ოპერაციული სისტემის დონეზე ვირტუალიზაცია aka OS-Level virtualization ტექნოლოგიაა რომელიც საშუალებას იძლევა მრავალი იზოლირებული ინსტანსის გაშვებისა ერთ ფიზიკურ მანქანაზე. ეს ინსტანსები, რომლებსაც კონტეინერებს ვუწოდებთ, იზიარებენ ერთ ოპერაციულ სისტემას კერნელს, თუმცა ერთმანეთისგან დამოუკიდებლად მოქმედებენ. ეს მიდგომა განხვავდება ტრადიციული ვირტუალური მანქანებისაგან, რომლებიც მოითხოვდნენ, რომ მთლიანი ვირტუალკა ამოწეულიყო თავის OS იანად.

**როდის გვადგება დოკერი?**

მიუხედავად იმისა, რომ დოკერის გამოყენება ნებისმიერი სახის დეველოპმენტსა და დანერგვაში შეგვიძლია გამოვიყენოთ, იგი ძირითადად უფრო კონკრეტულ შემთხვევებშია გამოსადეგი:

1. Continuous deploying soft: დოკერის ტექნოლოგია და კარგი დევოპსური მიდგომები საშუალებას იძლევა რომ კონტეინირიზებული აპლიკაციები რამდენიმე წამში დავნერგოთ, განსხვავებით ძველებური მონოლითური აპლიკაციებისა რომლებსაც ბევრად მეტი დრო სჭირდებოდათ დასანერგად (ვინაიდან საჭიროებდა ხელით ჩარევას, სერვისების გათიშვას, ფაილების გადატანას და სხვა მანუალურ პროცესებს). აპლიკაციაში შეტანილი ცვლილებები ბევრად უფრო მალე ინერგება როდესაც ჩვენი კონტეინერი CI/CD pipeline ის ნაწილია.
2. Building microservice-based architecture: თუ ვიყენებთ მიკროსერვისულ არქიტექტურას, რომელიც თავის თავში გულისხმობს იმას რომ ძველებური მონოლითისგან განსხვავებით გვაქვს შედარებით ბევრი დანერგვადი კომპონენტი ამ შემთხვევაში დოკერი საკმაოდ გამოსადეგი ხდება, ვინაიდან მრავალი კონტეინერის მართვისთვისაა ზუსტადაც შექმნილი კონტეინერების ორკეტრაციის ხელსაწყოები, როგორიცაა დოკერ სვარმი და საშუალება გვექნება რომ კონტეინერების დახმარებით ჩვენი აპლიკაცია უფრო მოქნილი და გამძლე გავხადოთ.
3. Legacy app migration: თუ გვინდა ლეგასი აპლიკაციის შეცვლა, ახალ ტექნოლოგიაზე გადასვლა ანდაც უბრალოდ რაიმე ექსპერიმენტების ჩატარება, ამაშიც დოკერი დაგვეხმარება. ლეგასი პროექტები ხშირ შემთხვევაში არიან მიბმულები თავის ინფრასტრუქტურასა და გარემოზე და იქ ცვლილებების კეთება არის საკმაოდ სარისკო და პრობლემატური. ამ დროს დოკერი ძალიან მარტივად გვთავაზობს ჩვენი აპლიკაციისთვის ისეთ გარემოს სადაც რაც საჭიროა ამ აპლიკაციას ექნება. შესაბამისად თავისუფლად შეგვიძლია ვიექსპერიმენტოთ, ლეგასი აპლიკაციები ვშალოთ/ვანახლოთ და ა.შ. ისე რომ რამე არ გავაფუჭოთ.
4. hybrid cloud and multi-cloud applications: ასევე დოკერის დახმარებით, გამომდინარე იქიდან რომ ჩვენი აპლიკაცია მზადდება როგორც ერთი მთლიანი პაკეტი თავისი ყველაფრით და ნებისმიერ გარემოზე გაშვებადია, შეგვიძლია რომ რამდენიმე სახის ქლაუდის სერვისი გამოვიყენოთ ერთად. შეგვიძლია აპლიკაცია გვქონდეს სადღაც aws ზე და უცებ ბევრი თავის ტკივილის გარეშე შეგვეძლება გადავაცუნცულოთ azure ზე

Docker კიდევ ერთ მნიშვნელოვან პრობლემას აგვარებს, რომელიცაა „ჩემს მანქანაზე მუშაობს“. გამომდინარე იქიდან, რომ დოკერის კონტეინერი შეიცავს არა მარტო აპლიკაციას არამედ ყველა ამ აპლიკაციის დამოკიდებულებას, კონფიგურაციასა და ბიბლიოთეკას, თუ ჩვენს აპლიკაციას შევამოწმებთ დოკერში კონტეინერში გაშვების შემდეგ, შეგვეძლება ვთქვათ რომ იგი იმუშავებს სატესტო სერვერზეც და პროდზეც. ეს მიდგომა გვიხსნის გარემოს გამოწვეული პრობლემების მეტ წილს.

Docker ის ერთ ერთი მნიშვნელოვანი დადებითი თვისებაა ეფექტურობა. კონტეინერები სისტემის რესურსს უფრო ეფექტურად ყენებენ ვიდრე ტრადიციული ვირტუალკები და მათი რესტარტი/ჩართვა/გამორთვა წამებში შეიძლება. ეს საშუალებას გვაძლევს რომ დანერგვები შევასრულობთ უფრო სწრაფად და მოქნილად.

მოდი დოკერის მნიშვნელოვან ნაწილებს გავეცნოთ.  
Docker Engine არის ის ძირითადი ტექნოლოგია რომელიც ასრულებს ბრძანებებსა და ვორქფლოუებს, რომლებიც დაკავშირებულია კონტეინერის აპლიკაციების დაბილდვასთან. Engine ქმნის სერვერის მხარეს მომუშავე background worker ს (dockerd) რომელიც უზრუნველყოფს იმიჯების, კონტეინერების, ნეთვორქების და სანახი ვოლიუმების დაჰოსტვა მენეჯმენტს. Docker ის სხვა მნიშვნელოვანი კომპონენტებია:

Docker Hub:

დოკერ ჰაბი არის software as a service ხელსაწყო რომელიც საშუალებას გვაძლევს გავაზიაროთ ან თავად გამოვიყენოთ სხვისი გაზიარებული იმიჯები. ჩვენ შეგვიძლია გამოვიყენოთ როგორც საჯარო დოკერ ჰაბი ასევე შეგვიძლია შევქმნათ პირადი რეპოზიტორი და ვიმუშავოთ პირად სივრცეში გაზიარების გარეშე.

Trusted Registry: ეს დოკერ ჰაბის მსგავსი რეპოზიტორია უბრალოდ აქ გამოქვეყნებული იმიჯები უფრო მეტად კონტროლდება.

Docker swarm: დოკერ სვარმის დახმარებით შეგვიძლია რამდენიმე დოკერის ჰოსტს ერთდროულად ვემუშავოთ და უპრობლემოდ დავნერგოთ ცვლილებები რამდენიმე მანქანაზე.

Compose: რამდენიმე კონტეინერის ერთად მოსაგროვებლად, დასაკონფიგურირბლად და სამართავად.

დოკერის პლიუსები და მინუსები:

დოკერის პლიუსერბია:

* portability (მარტივად გადატანითობა?) რათა მომხმარებლებმა შეძლონ კონტეინერების გაზიარება სხვადასხვა ჰოსტებზე
* დაბალი რესურსების მოხმარება
* სწრაფი დანერგვის დრო VM ებთან შედარებით

თუმცა აქვე უნდა აღვნიშნოთ გამოწვევები დოკერში:

* დიდ კომპანიებში არსებული კონტეინერების მართვა გამომდინარე რეაოდენობიდან რთულდება
* რაც უფრო მეტი აპლიკაცია ინერგება ერთი სოფტის ჭრილში მით მეტად იხლართება პროდაქშენი, დამოკიდებულებები და ა.შ. ანუ ორკესტრაცია რთულდება

რომ შევაჯამოთ, Docker არამხოლოდ აპლიკაციის დანერგვას გვიმარტივებს, არამედ ასევე გარანტიას გვაძლევს რომ ჩვენი აპლიკაცია ნებისმიერ გარემოში ერთნაირად იმუშავებს.

ვინაიდან დოკერს უკვე გავეცანით და ვიცით ვინაა და რას წარმოადგენს, ახლა შეგვიძლია მის რამდენიმე კომპონენტს დეტალურად გავეცნოთ.

Docker Images and Containers:

https://www.techtarget.com/searchitoperations/definition/Docker-image

მოდი გავაგრძელოთ საუბარი დოკერის ორ მნიშვნელოვან კონცეპტზე: იმიჯები და კონტეინერები. მათი გაცნობა დაგვეხმარება დოკერის ზედა დონეზე გაგებაში.

დავიწყოთ დოკერ იმიჯებით. წარმოვიდგინოთ რომ იმიჯი არის ნახაზი, რომელზე წარმოდგენილი ინსტრუქციების მიხედვითაც იქმნება კონტეინერი. Docker image ები ინსტრუქციების კრებულივითაა რომელთა თანმიმდევრობით განხორციელების შედეგად კონტეინერს მივიღებთ. იგი შეიცავს როგორც აპლიკაციის კოდს ასევე სხვა აპლიკაციის დამოკიდებულებებს რომლებიც მის გასაშვებად საჭიროა, მაგალითად: ბიბლიოთეკებს, თულებს ფაილებს და ა.შ. ერთი იმიჯი შეგვიძლია რამდენჯერმე გავუშვათ და მივიღოთ რამდენიმე მუშა კონტეინერი. Image ები გათვლილად კეთდება ისე რომ იყოს ეფექტური და მრავალჯერადი მრავალი შრის გამოყენებით (multiple layer).

1. **შრეებიანი სტრუქტურა(Layered Structure):** თითოეული იმიჯი იქმნება რამდენიმე შრის ერთმანეთზე დალაგების შედეგად. თითოეული შრე ამატებს რამე ახალ ფაილებს ან ცვლის რაიმეს წინა შრიდან მოწოდებული ფაილებიდან. თითოეული ეს შრე იქმნება DockerFile ის ინსტრუქციის მიხედვით, რომელშიც შეიძლება გაწერილი იყოს რაიმე სოფტის ინსტალაცია, ფაილების კოპირება, გენერაცია და ა.შ.
2. **უცვლელი შრეები (Immutable Layers):** მას შემდეგ რაც შრის შექმნა დასრულდება მისი ცვლილება შეუძლებელია რადგან იგი read only ხდება.
3. **სისწრაფე და ეფექტურობა (Speed and Efficiency):** როცა დოკერი იმიჯს ბილდავს ქმნის მას მხოლოდ იმ შრეების განახლება სჭირდება რომლებიც წინა ბილდის შემდეგ შეიცვალა, დანარჩენები უცვლელი რჩება და შესაბამისად ზედმეტი დრო არ იხარჯება.
4. **ჩაწერადი შრე (Writable Container Layer):** როცა უკვე ჩვენს კონტეინერს გავუშვებთ დოკერი კიდევ ერთ შრეს ამატებს ზემოდან უკვე არსებულებს, რომლის მოდიფიკაციაც შესაძლებელია. ეს გულისხმობს იმას რომ ამ შრეში შეგვიძლია ჩავწეროთ სოფტი, შევცვალოთ ფაილები და ა.შ. ეს შრე კონტეინერის შრეა. იგი მხოლოდ ერთ კონტეინერს ეხება და სხვა კონტეინერებზე და თავად იმიჯზე გავლენა არ აქვს.