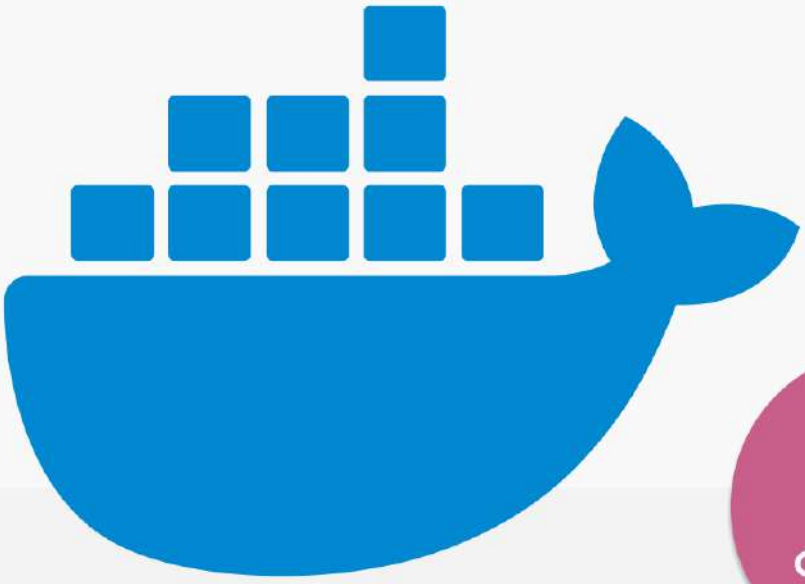
DOCKER

မြန်မာဘာသာ



သီအိုရီ
+
လက်တွေ့

"အခြေခံမှ လေ့လာလိုသူများအတွက်
ပြီးပြည့်စုံသော စာအုပ်

CODE with Thura

CODE WITH THURA

Docker

မြန်မာဘာသာ

အခြေခံမှ စ၍ လက်တွေ့ လုပ်ငန်းခွင်အထိ၊

ကိုယ်တိုင် လေ့လာလိုသူများအတွက် ပြီးပြည့်စုံသော စာအုပ်။

Thura (Code with Thura)

Version 1.0

စာရေးသူအကြောင်း

- ပညာရေးနယ်ပယ်နှင့် နည်းပညာနယ်ပယ် နှစ်ခုစလုံးတွင် လုပ်ကိုင်နေသော IT အင်ဂျင်နီယာ တစ်ဦးဖြစ်သည်။
- နည်းပညာတက္ကသိုလ် (မန္တလေး) မှ IT အင်ဂျင်နီယာဘာသာရပ် အထူးပြုဖြင့် B.E (IT) ဘွဲ့ရရှိခဲ့သည်။
- ၂၀၁၇ ခုနှစ် နောက်ပိုင်းတွင် Development နယ်ပယ်သို့ စတင်ဝင်ရောက်ခဲ့ပြီး ပြည်တွင်းရှိ နည်းပညာ ကုမ္ပဏီများ နှင့် ပြည်ပအခြေစိုက် ကုမ္ပဏီများတွင် လုပ်ကိုင်ခဲ့သည်။
- နည်းပညာဆိုင်ရာ ဗဟုသုတများကို ရိုးရှင်းစွာ ပြန်လည်မျှဝေပေးရန်နှင့် ပညာရေးကဏ္ဍတွင် အထောက်အကူ ပြုစေရန် ရည်ရွယ်၍ ‘Code with Thura’ Platform ကို ၂၀၂၄ ခုနှစ်တွင် စတင်ခဲ့သည်။
- လက်ရှိတွင် ပညာရေးနယ်ပယ်၌ Postgraduate Researcher တစ်ဦးအဖြစ် ပါဝင်ဆောင်ရွက်လျက်ရှိပြီး Development Project များနှင့် သင်ကြားရေးလုပ်ငန်းများကို လုပ်ကိုင်လျက်ရှိသည်။

ဆက်သွယ်လိုပါက thura.00011@gmail.com သို့လည်းကောင်း၊ ‘Code with Thura’ ၏ တရားဝင် ဝက်ဘ်ဆိုဒ် စာမျက်နှာဖြစ်သည့် https://www.codewiththura.com/contact သို့လည်းကောင်း ဝင်ရောက် ဆက်သွယ် မေးမြန်းနိုင်ပါသည်။


CODE
with Thura

အခန်း (၁) - Docker မပေါ်မီခေတ်က နည်းပညာ အခက်အခဲများ

၁.၁ နည်းပညာအခက်အခဲတွေ ရှိခဲ့တဲ့ ခေတ်ဟောင်း

ယနေ့ခေတ် ကျွန်တော်တို့ရဲ့ နေ့စဉ်ဘဝ ဖြတ်သန်းမှုတွေမှာ Software Application တွေဟာ မရှိမဖြစ် အရေးပါတဲ့ နေရာကနေ ပါဝင်နေပါတယ်။ ဒီဖက်ခေတ် Application တွေဟာ အသုံးပြုသူ သန်းပေါင်းများစွာကို တပြိုင်နက်တည်း ဝန်ဆောင်မှုပေးနိုင်သလို၊ နေ့စဉ် ဝင်ရောက်နေတဲ့ Traffic ပေါင်းများစွာကိုပါ ချောချောမွေ့မွေ့ ကိုင်တွယ်ဖြေရှင်းပေးနေရပါတယ်။ ကျွန်တော်တို့ရဲ့ သွားလာလှုပ်ရှားမှုတွေ၊ ဆက်သွယ်ရေးနဲ့ အလုပ်ကိစ္စတွေ အားလုံးဟာ ဒီ App တွေပေါ်မှာ အများကြီး မှီခိုနေရတဲ့အတွက်၊ App တစ်ခုခု Down သွားတာနဲ့ အရာအားလုံး လိုက်ပြီး ရပ်တန့်သွားနိုင်တဲ့အထိ သက်ရောက်မှု ကြီးမားပါတယ်။

ဒါပေမဲ့ ဒီလိုမျိုး အသုံးပြုသူ သန်းချီကို အလွယ်တကူ ဝန်ဆောင်မှုပေးနိုင်ဖို့ဆိုတာ အရင်ခေတ်ကတော့ လွယ်ကူတဲ့ကိစ္စ မဟုတ်ခဲ့ပါဘူး။ လွန်ခဲ့တဲ့ ဆယ်စုနှစ်တစ်ခုလောက်ကိုပဲ ပြန်ကြည့်လိုက်မယ်ဆိုရင် Application တစ်ခုကို Server ပေါ်မှာ အဆင်ပြေပြေ Run နိုင်ဖို့ဆိုတာ နောက်ကွယ်မှာ အချိန်၊ ငွေကြေးနဲ့ လူအင်အား အမြောက်အမြားကို ရင်းနှီးမြှုပ်နှံခဲ့ရပါတယ်။ အဲဒီခေတ်က အသုံးပြုခဲ့တဲ့ နည်းပညာတွေမှာ အားနည်းချက်တွေ၊ ကန့်သတ်ချက်တွေ အများကြီး ရှိနေခဲ့တာကြောင့်၊ App တစ်ခုကို Server ပေါ်တင်ပြီး Deploy

လုပ်ရတဲ့ လုပ်ငန်းစဉ်ဟာ တကယ့်ကို ခက်ခဲတဲ့ စိန်ခေါ်မှုတစ်ခု ဖြစ်ခဲ့ပါတယ်။

အဲဒီလို အခက်အခဲတွေ ဖြစ်ရတဲ့ အဓိက တရားခံက အဲဒီခေတ်က ကျယ်ကျယ်ပြန့်ပြန့် အသုံးပြုခဲ့တဲ့ One-App-Per-Server ဆိုတဲ့ စနစ်ကြောင့်လို့ ဆိုရပါမယ်။ ဒီစနစ်က Physical Server တစ်လုံးပေါ်မှာ Application တစ်ခုတည်းကိုသာ သီးသန့် တင်ပြီး အသုံးပြုတဲ့ ပုံစံမျိုးပါ။ ဆိုလိုတာက လုပ်ငန်းတစ်ခုမှာ နောက်ထပ် Application အသစ်တစ်ခု လိုအပ်လာပြီဆိုတာနဲ့၊ အဲဒီ App အသစ်အတွက် Physical Server အသစ်တစ်လုံးကို မဖြစ်မနေ ထပ်မံ ဝယ်ယူရတဲ့ သဘောပါပဲ။

ဒါအပြင် နောက်ထပ် ကြုံတွေ့ရတဲ့ ပြဿနာတစ်ခု ရှိပါသေးတယ်။ လုပ်ငန်းလည်ပတ်နေချိန်မှာ Server ရဲ့ Performance မနိုင်တော့ဘဲ Application အလုပ်လုပ်တာ နှေးကွေးသွားတာ၊ ထစ်ငေါ့ပြီး ရပ်တန့်သွားတာမျိုးကို ဘယ်လုပ်ငန်းရှင်ကမှ အဖြစ်မခံနိုင်ပါဘူး။ ဒါကြောင့် Server အသစ်ဝယ်ယူတဲ့အခါတိုင်း ကိုယ့်လုပ်ငန်း၊ ကိုယ့် Application အတွက် တကယ် လိုအပ်မယ့် ပမာဏထက် အဆပေါင်းများစွာ ပိုပြီး မြင့်မားတဲ့ Overpowered ဖြစ်နေတဲ့ Server တွေကိုသာ အကုန်အကျခံပြီး ဝယ်ယူခဲ့ကြပါတယ်။ ရှေ့ဆက်ဖြစ်လာနိုင်တဲ့ ပြဿနာတွေကို ကြိုတင် ကာကွယ်တဲ့အနေနဲ့ အခုလို လိုတာထက်ပိုပြီး ဝယ်ယူခဲ့ကြတာပါ။

ဒါပေမဲ့ ဒီလိုမျိုး လိုအပ်တာထက် အလွန်အကျွံ ပိုဝယ်ထားတဲ့အတွက် နောက်ဆက်တွဲအနေနဲ့ ဆိုးကျိုးတွေ ဖြစ်ပေါ်လာခဲ့ပါတယ်။ ကုမ္ပဏီတွေရဲ့

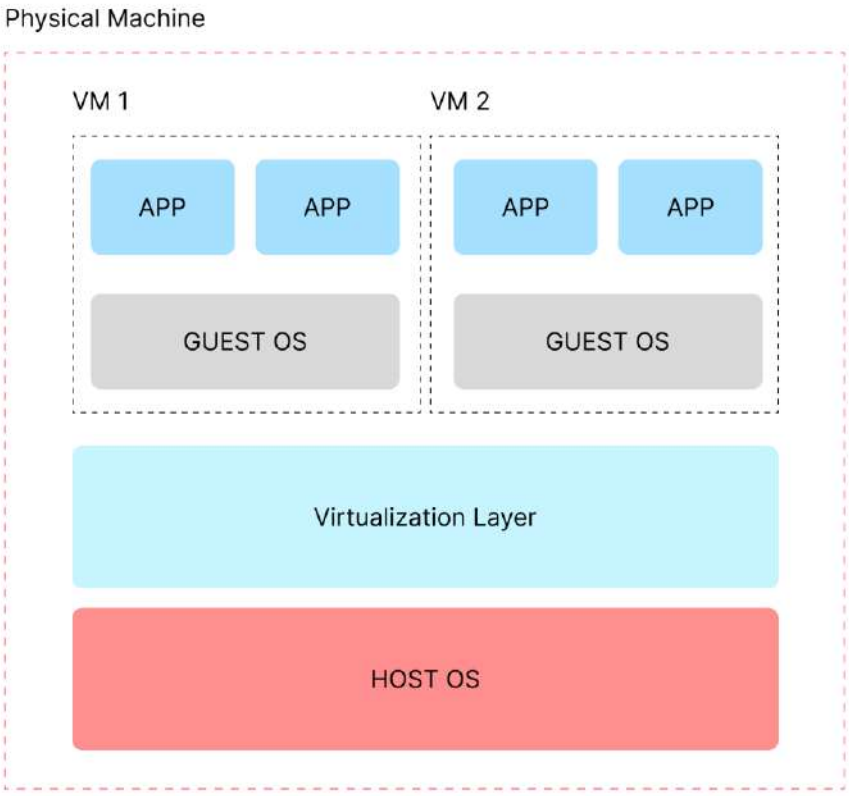
Data Center တွေထဲမှာ တကယ်တမ်း လက်တွေ့ အပြည့်အဝ အလုပ်မလုပ်ဘဲ ဒီအတိုင်း အလဟဿ ရှိနေရတဲ့ Server တွေ စုပုံလာခဲ့တာပါ။ ဒါဟာ လုပ်ငန်းရဲ့ ငွေကြေး အရင်းအနှီးတွေကို အလဟဿ ဖြစ်စေရုံမက၊ လျှပ်စစ်မီးနဲ့ အခြားစွမ်းအင်တွေ လိုမျိုး သဘာဝပတ်ဝန်းကျင် အရင်းအမြစ်တွေကိုပါ ဆိုးဆိုးရွားရွား ဖြုန်းတီးစေတဲ့ အကြောင်းတရား ဖြစ်လာခဲ့ပါတယ်။

၁.၂ Virtualization နဲ့ VMware ရဲ့ အလှည့်အပြောင်း

ဒီလိုမျိုး ငွေကြေးနဲ့ အရင်းအမြစ်တွေ အလဟဿ ဖြုန်းတီးနေရတဲ့ ပြဿနာတွေကို အဆုံးသတ်ဖို့အတွက် VMware ကုမ္ပဏီကနေ နည်းပညာသစ်တစ်ခုကို ကမ္ဘာကြီးဆီ စတင် မိတ်ဆက်ပေးခဲ့ပါတယ်။ အဲဒါကတော့ နည်းပညာလောကကို တစ်ခေတ်ဆန်းစေခဲ့တဲ့ "Virtual Machine" (VM) လို့ခေါ်တဲ့ နည်းပညာပါပဲ။ ဒီ VMware ရဲ့ နည်းပညာ ပေါ်ထွက်လာတာနဲ့ တစ်ပြိုင်နက်၊ အရင်က ကြုံတွေ့နေရတဲ့ အခက်အခဲတွေ၊ ကန့်သတ်ချက်တွေ အားလုံးဟာ ချက်ချင်း ဆိုသလို အကောင်းဘက်ကို ပြောင်းလဲသွားခဲ့ပါတယ်။

ဒီ VM နည်းပညာက Server စက် တစ်လုံးတည်းပေါ်မှာတင် Application အများကြီးကို တစ်ခုနဲ့တစ်ခု လုံးဝ မထိခိုက်၊ မနှောင့်ယှက်စေဘဲ သီးခြားစီ ခွဲပြီး Run ပေးနိုင်ပါတယ်။ ပိုပြီး နားလည်သွားအောင် ရှင်းပြရရင်၊ အရင်ကလို Application တစ်ခုအတွက် Server တစ်လုံး သီးသန့်ပေးစရာ

မလိုတော့ဘဲ၊ Server တစ်လုံးတည်းရဲ့ လုပ်ဆောင်နိုင်စွမ်းအား (CPU, RAM စတာတွေ) ကို လိုအပ်သလို ပိုင်းခြားပြီး Application တွေကို လုံလုံခြုံခြုံ Run လို့ရသွားတာ ဖြစ်ပါတယ်။ ဒါကို နည်းပညာအခေါ်အဝေါ်အရ Isolated Environment လို့ ခေါ်ပါတယ်။ Application တစ်ခု ပျက်သွားခဲ့ရင်တောင် အဲဒီစက်ပေါ်က တခြား Application တွေကို ထိခိုက်ခြင်း မရှိအောင် သီးသန့် ကာကွယ်ပေးထားနိုင်တဲ့ စနစ်မျိုး ဖြစ်ပါတယ်။



ပုံ ၁-၁၊ Virtual Machine တည်ဆောက်ပုံ

ဒီအချက်ဟာ စီးပွားရေးလုပ်ငန်းတွေအတွက်တော့ တကယ့်ကို ကြီးမားတဲ့ နည်းပညာပိုင်းဆိုင်ရာ အလှည့်အပြောင်းတစ်ခု ဖြစ်လာခဲ့ပါတယ်။ လုပ်ငန်းတွေအနေနဲ့ သူတို့ ဝယ်ယူထားပြီးသား Server တွေထဲမှာ အလဟဿ ပိုလျှံနေတဲ့ စွမ်းဆောင်ရည် (Excess Capacity) တွေကို

အကျိုးရှိရှိ ပြန်လည်အသုံးပြုပြီး၊ ရှိပြီးသား Server တွေပေါ်မှာပဲ Application အသစ်တွေကို တိုးချဲ့ Run လာနိုင်ကြပါတယ်။ ဒီလိုမျိုး ရှိပြီးသား အရင်းအမြစ်တွေကို စုစည်းပြီး အကျိုးအမြတ် အများဆုံး ပြန်လည်ရရှိအောင် ဆောင်ရွက်တဲ့ လုပ်ငန်းစဉ်ကို Resource Consolidation လို့ ခေါ်ပါတယ်။ ဒီ VM နည်းပညာကြောင့်ပဲ Resource Consolidation ကို လက်တွေ့မှာ အထိရောက်ဆုံး အကောင်အထည် ဖော်နိုင်ခဲ့ကြတာ ဖြစ်ပါတယ်။

၁.၃ Virtual Machine တွေရဲ့ အားနည်းချက်များ

VM နည်းပညာဟာ အရင်ခေတ်က ကြုံတွေ့ခဲ့ရတဲ့ Physical Server ပြဿနာတွေကို ထိထိရောက်ရောက် ဖြေရှင်းပေးနိုင်ခဲ့တာ မှန်ပေမဲ့၊ သူတို့ကိုယ်တိုင်ကတော့ ပြီးပြည့်စုံနေတဲ့ နည်းပညာတစ်ခုတော့ မဟုတ်ခဲ့ပါဘူး။ VM တွေမှာလည်း သူ့ဟာနဲ့သူ ကန့်သတ်ချက်တွေ ရှိနေပါသေးတယ်။ ဒီအထဲကမှ VM Model ရဲ့ အကြီးမားဆုံးသော အားနည်းချက်တစ်ခုကတော့ VM တစ်ခုချင်းစီတိုင်းမှာ " Guest Operating System" လို့ ခေါ်တဲ့ သီးသန့် OS တစ်ခုကို မဖြစ်မနေ အသုံးပြုရတဲ့ အချက်ပဲ ဖြစ်ပါတယ်။

ဒီလိုမျိုး VM တစ်ခုချင်းစီတိုင်းမှာ သီးသန့် OS တစ်ခုစီ ပါဝင်နေတဲ့အတွက် နောက်ဆက်တွဲ ပြဿနာတွေ ထွက်လာပါတယ်။ အဓိကကတော့ အဲဒီ Guest OS လည်ပတ်နိုင်ဖို့အတွက် စက်ထဲက RAM နဲ့ တခြားသော

Resource တွေကို ကြားဖြတ် အသုံးပြုခွင့် ပေးလိုက်တာ ဖြစ်ပါတယ်။ ကိုယ်တကယ် အလုပ်လုပ်စေချင်တဲ့ Application တွေဆီ Resource အပြည့်အဝ မပေးနိုင်တော့ဘဲ၊ ကြားခံထားရတဲ့ OS က Resource တော်တော်များများကို ယူသုံးသွားတဲ့ သဘောပါပဲ။

ဒါ့အပြင် ဆိုးကျိုးတွေက ဒီလောက်နဲ့ မပြီးသေးပါဘူး။ VM အရေအတွက် များလာတာနဲ့အမျှ အထဲမှာပါတဲ့ OS အရေအတွက်တွေကလည်း များလာပါတယ်။ အဲဒီအခါမှာ OS တစ်ခုချင်းစီတိုင်းကို ပုံမှန် Patching လုပ်ပေးရတာ၊ Security Update တွေ တင်ပေးရတာနဲ့ အမြဲမပြတ် Monitoring လုပ်ပြီး စောင့်ကြည့်နေရတာစတဲ့ အလုပ်တွေက တကယ့်ကို ခေါင်းခဲစရာ ဖြစ်လာပါတယ်။ ဒီလိုမျိုး ထိန်းသိမ်းရတဲ့ Maintenance Overhead ကုန်ကျစရိတ်တွေ၊ လူအင်အားနဲ့ အချိန်တွေကလည်း လုပ်ငန်းတွေအတွက် အတော်လေးကို ဝန်ထုပ်ဝန်ပိုး ဖြစ်စေပါတယ်။

ဒါတွေတင်မကသေးပါဘူး၊ VM တစ်ခုစီတိုင်းမှာ OS ကြီးတစ်ခုလုံး ပါဝင်နေတဲ့အတွက် စက်စတင်ဖွင့်တဲ့ Boot တက်တဲ့ လုပ်ငန်းစဉ်မှာ အတော်လေးကို အချိန်ကြာမြင့်တတ်ပါတယ်။ ဒါ့အပြင် Server တစ်ခုကနေ နောက် Server တစ်ခုဆီကို ပြောင်းရွှေ့ဖို့ လိုအပ်လာပြီဆိုရင်လည်း ပါဝင်တဲ့ File Size က ကြီးမားလွန်းတဲ့အတွက် သယ်ဆောင်ရွှေ့ပြောင်းရတာ အင်မတန်မှကို လက်ဝင်ပြီး အချိန်ကုန်စေတဲ့ ကိစ္စတစ်ခု ဖြစ်လာပါတော့တယ်။

၁.၄ Container ခေတ်ဦးနှင့် Google ၏ လျှို့ဝှက်လက်နက်

ကျွန်တော်တို့အများစုက VM တွေရဲ့ VM တွေရဲ့ လေးလံထိုင်းမှိုင်းတဲ့ ပြဿနာတွေ၊ ကန့်သတ်ချက်တွေနဲ့ လုံးပန်းနေရချိန်မှာပဲ၊ Google လိုမျိုး နည်းပညာ ထိပ်သီး ကုမ္ပဏီကြီး တွေကတော့ ဒီ VM အဆင့်ကို ကျော်လွန်ပြီး "Container" လို့ ခေါ်တဲ့ နည်းပညာကို သူတို့ရဲ့ လုပ်ငန်းစဉ်တွေမှာ အစောကတည်းက လျှို့ဝှက်လက်နက်တစ်ခုလို ကျယ်ကျယ်ပြန့်ပြန့် အသုံးပြုနေ ခဲ့ကြတာ ဖြစ်ပါတယ်။

VM နဲ့ Container ကြားက အဓိက ကွာခြားချက်ကို ရှင်းရရင်တော့ OS ရယူအသုံးပြုပုံခြင်း မတူညီတာပါပဲ။ အရင်က VM တွေမှာဆိုရင် VM တစ်ခုချင်းစီတိုင်းအတွက် သီးသန့် Guest OS တစ်ခုစီ မဖြစ်မနေ လိုအပ်ပါတယ်။ ဒါပေမဲ့ Container တွေမှာကျတော့ အဲဒီလို သီးသန့် OS ထည့်သွင်းပေးစရာ မလိုတော့ပါဘူး။ အဲဒီအစား Container တွေဟာ သူတို့ အလုပ်လုပ်နေတဲ့ အောက်ခံ Host Server ရဲ့ မူလ OS (Host OS) တစ်ခုကိုသာ အချိုးကျ မျှဝေပြီး သုံးစွဲသွားကြတာ ဖြစ်ပါတယ်။

ဒီလိုမျိုး Container တွေထဲမှာ OS ကြီးတစ်ခုလုံး သီးခြားစီ ထည့်သွင်းထားစရာ မလိုတော့တဲ့အတွက်၊ အရင်က ကြုံတွေ့နေရတဲ့ OS Overhead ဆိုတဲ့ အရင်းအမြစ် လေလွင့်မှုတွေ လုံးဝ မရှိတော့ပါဘူး။ ဒါကြောင့်လည်း Container တွေဟာ VM တွေထက်စာရင် အဆပေါင်းများစွာ ပိုမို ပေါ့ပါးသွားတာ ဖြစ်ပါတယ်။ ဥပမာ၊ အရင်က VM ၁၀ ခုပဲ Run နိုင်တဲ့ Host Server တစ်လုံးတည်းပေါ်မှာ၊ Container

နည်းပညာကို ပြောင်းလဲအသုံးပြုလိုက်မယ်ဆိုရင် Container အခု ၅၀ ကနေ ၁၀၀ အထိ အဆင်ပြေပြေ Run လာနိုင်မှာ ဖြစ်ပါတယ်။ ဒါဟာ ရှိပြီးသား Resource တွေကို အလဟဿ မဖြစ်စေဘဲ အများကြီး ပိုပြီး Efficient ဖြစ်အောင် သုံးစွဲနိုင်သွားတာပဲ ဖြစ်ပါတယ်။

ဒါ့အပြင် အမြန်နှုန်းပိုင်းကို ကြည့်မယ်ဆိုရင်လည်း Container တွေဟာ OS ကြီးတစ်ခုလုံးကို စောင့်ပြီး Boot တက်စရာ မလိုတော့တဲ့အတွက် စက္ကန့်ပိုင်းအတွင်းမှာတင် ချက်ချင်း အလုပ်လုပ်နိုင်ပါတယ်။ ဖိုင်အရွယ်အစား သေးငယ် ပေါ့ပါးတဲ့အတွက် Server တစ်ခုကနေ တစ်ခုကို ပြောင်းရွှေ့တာပဲဖြစ်ဖြစ်၊ သယ်ဆောင်ရွှေ့ပြောင်းရတဲ့ အပိုင်းမှာလည်း အလွန်ကို လွယ်ကူမြန်ဆန်သွားပါတယ်။ ဒီလိုမျိုး ထူးခြားတဲ့ အားသာချက်တွေကြောင့်ပဲ ဒီနေ့ခေတ် Modern Software Development လောကကြီးမှာ Container နည်းပညာဟာ မရှိမဖြစ် လိုအပ်တဲ့ အရေးပါဆုံး အစိတ်အပိုင်းတစ်ခုအဖြစ် အခိုင်အမာ နေရာယူလာခဲ့တာပဲ ဖြစ်ပါတယ်။

၁.၅ Linux ရင်ခွင်ထဲက ပေါက်ဖွားလာတဲ့ Container များ

ခေတ်သစ် Container တွေရဲ့ ဇာစ်မြစ်ကို ပြန်ကြည့်မယ်ဆိုရင် Linux လောကရဲ့ နှစ်ပေါင်းများစွာ ကြိုးပမ်းအားထုတ်မှုတွေကနေ ထွက်ပေါ်လာတဲ့ ရလဒ်တစ်ခု ဖြစ်တယ်ဆိုတာကို တွေ့ရမှာပါ။ Container နည်းပညာဆိုတာ ရုတ်တရက် ပေါ်ထွက်လာတာမျိုးတော့ မဟုတ်ပါဘူး။ Google လိုမျိုး နည်းပညာ ထိပ်သီး ကုမ္ပဏီကြီးတွေဟာ Container တွေ အလုပ်လုပ်နိုင်ဖို့အတွက် အဓိက အရေးပါတဲ့ Linux Kernel ရဲ့ Core အစိတ်အပိုင်းတွေထဲကို သူတို့ရဲ့ နည်းပညာတွေ ပါဝင် Contribute လုပ်ပေးခဲ့ပါတယ်။

ဒီလိုမျိုး ပံ့ပိုးမှုတွေ ရှိခဲ့တဲ့အတွက်ကြောင့်သာ Container တွေရဲ့ အသက်ဝိညာဉ်တွေဖြစ်တဲ့ Kernel Namespaces, Control Groups (cgroups) နဲ့ Capabilities လိုမျိုး နည်းပညာတွေ ပေါ်ပေါက်လာခဲ့တာပါ။ အကြမ်းဖျင်း ရှင်းပြရရင်၊ ဒီ နည်းပညာတွေဟာ Container တစ်ခုနဲ့ တစ်ခုကို သီးခြားစီ လုံလုံခြုံခြုံ ခွဲခြားပေးဖို့နဲ့၊ Container တွေအတွက် လိုအပ်တဲ့ CPU, RAM စတဲ့ Resource တွေကို စနစ်တကျ ခွဲဝေ ထိန်းချုပ်ပေးနိုင်ဖို့ အလွန်အရေးပါပါတယ်။ ဒါပေမဲ့ ပြဿနာတစ်ခုက ဒီနည်းပညာတွေဟာ အစွမ်းထက်သလောက် အင်မတန်မှလည်း ကိုင်တွယ်ရ ခက်ခဲပြီး ရှုပ်ထွေးလွန်းနေတာပါ။ ဒါကြောင့်မို့လို့ Docker ဆိုတဲ့ နည်းပညာ မပေါ်လာခင် အချိန်အထိ Linux Container တွေကို တိုက်ရိုက် အသုံးပြုဖို့ဆိုတာ သာမန် Developer တစ်ယောက်အတွက် အလွန် ခက်ခဲတဲ့ အလုပ်တစ်ခုဖြစ်နေခဲ့ပါတော့တယ်။

၁.၆ Docker - နည်းပညာလောကရဲ့ ဇာတ်လိုက်ကျော်

ဒီလိုမျိုး အသုံးပြုရ ခက်ခဲပြီး ရှုပ်ထွေးနေတဲ့ အခြေအနေတွေကြားထဲကို Docker ဆိုတဲ့ "ဇာတ်လိုက်ကျော်" နည်းပညာ ရောက်ရှိလာခဲ့ပါတယ်။ Docker ဟာ အရင်က အသုံးပြုဖို့ ခက်ခဲလှတဲ့ Linux Container ရဲ့ နည်းပညာပိုင်းဆိုင်ရာ Layer တွေကို သုံးစွဲသူတွေအတွက် အလွန်လွယ်ကူရှင်းလင်းတဲ့ Interface တစ်ခုအနေနဲ့ ဖုံးအုပ် ပြင်ဆင်ပေးလိုက်ပါတယ်။ ဒီလို ပြင်ဆင်ပေးလိုက်တဲ့အတွက် Developer တိုင်းဟာ Container ရဲ့ အစွမ်းသတ္တိတွေကို မိမိတို့ရဲ့ လက်ထဲမှာတင် “မီးကိုရှာဖွေတွေ့ရှိသလို” အလွယ်တကူ အသုံးချလာနိုင်ပါတယ်။

Docker ကြောင့်သာ Developer တွေအနေနဲ့ ရှုပ်ထွေးလှတဲ့ အောက်ခြေ Infrastructure အပိုင်းတွေကို ခေါင်းစားခံပြီး စဉ်းစားနေစရာ မလိုတော့ဘဲ၊ မိမိတို့ ရေးသားထားတဲ့ Code တွေကို ဘယ်နေရာ၊ ဘယ် Server မှာမဆို အဆင်ပြေပြေ Run နိုင်မယ့် ခေတ်သစ် Software Development လောကကြီး ဖြစ်ပေါ်လာခဲ့တာပါ။ အခုဆိုရင် Docker ဟာ နည်းပညာလောကမှာ မရှိမဖြစ် အရေးပါတဲ့ အစိတ်အပိုင်းတစ်ခု ဖြစ်နေပါပြီ။ ဒီ Docker ရဲ့ အလုပ်လုပ်ပုံ အသေးစိတ်တွေနဲ့ သူ့ရဲ့ Ecosystem အကြောင်းတွေကိုတော့ လာမယ့် အခန်းတွေမှာ ကျွန်တော်တို့ ဆက်လက် ဆွေးနွေးသွားကြပါမယ်။

၁.၇ Windows ရဲ့ Container အိမ်မက်

Linux ဘက်မှာ Container နည်းပညာတွေ အရှိန်အဟုန်နဲ့ လူသုံးများ အောင်မြင်လာတာနဲ့အမျှ Microsoft ကုမ္ပဏီအနေနဲ့လည်း ဒီ Container နည်းပညာတွေကို သူတို့ရဲ့ Windows Platform ပေါ် ယူဆောင်လာဖို့ အပြင်းအထန် ကြိုးစားအားထုတ်ခဲ့ပါတယ်။ ဒီကြိုးပမ်းမှုရဲ့ ရလဒ်အနေနဲ့၊ ဒီနေ့ခေတ် Windows Desktop တွေနဲ့ Server တွေမှာဆိုရင် Windows Container တွေရော၊ Linux Container တွေကိုပါ အဆင်ပြေပြေ အသုံးပြုနိုင်တဲ့ အခြေအနေကို ရောက်ရှိလာခဲ့ပြီ ဖြစ်ပါတယ်။

ဒီနေရာမှာ အနည်းငယ်ရှင်းရရင်၊ Windows Container တွေဆိုတာက Windows Application တွေကို သီးသန့် ခွဲခြားပြီး Run ဖို့အတွက် အဓိက ရည်ရွယ်ဖန်တီးထားတာပါ။ ရှေ့မှာ ကျွန်တော်တို့ ဆွေးနွေးခဲ့သလိုပဲ Container တစ်ခုဟာ သူအလုပ်လုပ်မယ့် အောက်ခံ Host System ရဲ့ Kernel ကို မျှဝေသုံးစွဲရပါတယ်။ ဒါကြောင့် Windows Container တွေကို Run ဖို့ဆိုရင် အောက်ခံ Host System မှာ Windows Kernel ရှိနေဖို့ မဖြစ်မနေ လိုအပ်ပါတယ်။ လက်ရှိအချိန်မှာတော့ Windows 10, Windows 11 နဲ့ နောက်ဆုံးထွက် Windows Server Version တွေ အားလုံးမှာ ဒီ Windows Container တွေကို တိုက်ရိုက် Native အသုံးပြုနိုင်အောင် Support ပေးထားပြီး ဖြစ်ပါတယ်။

ဒါ့အပြင် Microsoft ဘက်ကနေ ထပ်မံ မိတ်ဆက်ပေးလိုက်တဲ့ အရေးအပါဆုံး နည်းပညာ တိုးတက်မှုတစ်ခုကတော့ WSL 2 (Windows

Subsystem for Linux) လို့ခေါ်တဲ့ နည်းပညာပဲ ဖြစ်ပါတယ်။ အရင်ကဆိုရင် Windows ပေါ်မှာ Linux နည်းပညာတွေကို သုံးဖို့ဆိုတာ အတော်လေး ခက်ခဲပါတယ်။ ဒါပေမဲ့ ဒီ WSL 2 နည်းပညာ ပါဝင်လာတဲ့အခါမှာတော့ Windows OS ပေါ်မှာတင် Linux Container တွေကို ချောချောမွေ့မွေ့နဲ့ အပြည့်အဝ Run နိုင်စွမ်း ရှိလာတာ ဖြစ်ပါတယ်။

ဒီအချက်ရဲ့ အဓိကအကျိုးကျေးဇူးကတော့ Developer တွေအနေနဲ့ ကွန်ပျူတာ စက်တစ်လုံးတည်း၊ Platform တစ်ခုတည်းပေါ်မှာတင် မိမိတို့ လိုအပ်တဲ့ Windows Container တွေကိုရော၊ Linux Container တွေကိုပါ တစ်ပြိုင်နက်တည်း Develop လုပ်တာတွေ၊ Test လုပ်တာတွေကို စိတ်ကြိုက် လွယ်လွယ်ကူကူ ဆောင်ရွက်နိုင်သွားတာပဲ ဖြစ်ပါတယ်။ ဒီလို ပြည့်စုံတဲ့ စွမ်းဆောင်နိုင်မှုတွေ ကြောင့်ပဲ Windows ဟာလည်း Software Developement အတွက် အလွန် အစွမ်းထက်တဲ့ Platform တစ်ခုအဖြစ် ပြန်လည် ရပ်တည်လာနိုင်ခဲ့တာ ဖြစ်ပါတယ်။

၁.၈ Linux Container များ၏ အသာစီးရမှု

လက်တွေ့ လုပ်ငန်းခွင် နယ်ပယ်တွေကို လေ့လာကြည့်မယ်ဆိုရင် Windows Container တွေထက်စာရင် Linux Container တွေကိုသာ အဓိကထားပြီး ကျယ်ကျယ်ပြန့်ပြန့် အသုံးပြုနေကြတာကို တွေ့ရမှာပါ။ ဘာကြောင့် ဒီလောက်တောင် Linux Container တွေက အသာစီးရပြီး နေရာယူထားနိုင်ရတာလဲ ဆိုတဲ့ အကြောင်းရင်းကတော့ ရှင်းလင်းပါတယ်။

ပထမအချက်အနေနဲ့ အခြေခံအားဖြင့် Linux Container တွေဟာ Windows Container တွေနဲ့ ယှဉ်လိုက်ရင် ဖိုင်အရွယ်အစား အများကြီး ပိုမို သေးငယ် ပေါ့ပါးတဲ့အပြင်၊ အလုပ်လုပ်ရာမှာလည်း ပိုပြီး မြန်ဆန် သွက်လက်လို့ ဖြစ်ပါတယ်။

ဒီလို ပေါ့ပါးမြန်ဆန်မှုတွေထက် ပိုပြီး အရေးကြီးတဲ့ အချက်တစ်ခု ရှိပါသေးတယ်။ အဲဒါကတော့ Linux-based Tooling Ecosystem ကြီးပါပဲ။ Container နည်းပညာနဲ့ ပတ်သက်ရင် Linux မှာ ရှိတဲ့ Software Tool တွေ၊ Library တွေနဲ့ Community တွေရဲ့ ပံ့ပိုးကူညီမှုတွေက အခြားစနစ်တွေထက် အဆပေါင်းများစွာ ပိုမို ပြည့်စုံများပြားနေတာကို တွေ့ရမှာပါ။ ဒါကြောင့် လုပ်ငန်းခွင်မှာ နည်းပညာပိုင်းဆိုင်ရာ အခက်အခဲ တစ်ခုခု ကြုံလာခဲ့ရင်တောင် လိုအပ်တဲ့ အဖြေနဲ့ အထောက်အကူတွေကို အလွယ်တကူ ရှာဖွေ ရရှိနိုင်ပါတယ်။

ဒီလိုမျိုး လက်တွေ့လုပ်ငန်းခွင်မှာ အသုံးအများဆုံးနဲ့ အားသာချက်တွေ အများကြီး ရှိနေတဲ့အတွက်ကြောင့်၊ ကျွန်တော်တို့ရဲ့ ဒီစာအုပ်မှာ ရှေ့ဆက်ပြီး ဆွေးနွေးသွားမယ့် အကြောင်းအရာတွေ၊ လက်တွေ့ လုပ်ငန်းစဉ်တွေနဲ့ ဥပမာတွေ အားလုံးကို Linux Container တွေအပေါ်မှာပဲ အဓိက အခြေခံပြီး ဆက်လက် ရှင်းပြသွားမှာ ဖြစ်ပါတယ်။

၁.၉ Mac OS နှင့် Container နည်းပညာ

"Mac Container" ဆိုတဲ့ အသုံးအနှုန်းကို ကြားလိုက်တဲ့အခါ လေ့လာသူတွေအနေနဲ့ Mac OS သီးသန့်အတွက် ထုတ်လုပ်ထားတဲ့ Container အမျိုးအစားများ ရှိနေမလားလို့ ထင်ကောင်း ထင်နိုင်ပါတယ်။ တကယ်တော့ "Mac Container" ဆိုတာမျိုး သီးသန့် မရှိပါဘူး။ ဒါပေမဲ့ Apple ရဲ့ Mac ကွန်ပျူတာတွေဟာ Container တွေနဲ့ အလုပ်လုပ်ဖို့အတွက် အင်မတန် အစွမ်းထက်ပြီး အဆင်ပြေလွန်းတဲ့ Platform တွေ ဖြစ်နေတာကတော့ ငြင်းလို့မရတဲ့ အမှန်တရားပါပဲ။ လုပ်ငန်းခွင်ထဲက ပညာရှင်အများစုကလည်း နေ့စဉ် နည်းပညာပိုင်းဆိုင်ရာ အလုပ်တွေ လုပ်ဆောင်တဲ့အခါ Mac ပေါ်မှာပဲ Container တွေကို အဓိက အသုံးပြုနေကြပါတယ်။

Mac ပေါ်မှာ Container တွေနဲ့ အလုပ်လုပ်ဖို့အတွက် အသုံးအများဆုံးနဲ့ လူကြိုက်အများဆုံး နည်းလမ်းကတော့ Docker Desktop ကို အသုံးပြုတာ ဖြစ်ပါတယ်။ ဒီနေရာမှာ နည်းပညာပိုင်းဆိုင်ရာ စိတ်ဝင်စားစရာ အချက်တစ်ခုက Mac OS ရဲ့ Kernel ဟာ Linux Kernel မဟုတ်တဲ့အတွက်၊ Linux အတွက် ထုတ်လုပ်ထားတဲ့ Container တွေကို Mac ပေါ်မှာ တိုက်ရိုက် (Native) Run လို့ မရဘူး ဆိုတာပါပဲ။

ဒီလို တိုက်ရိုက် Run လို့မရတဲ့ အခြေအနေကို ဖြေရှင်းဖို့အတွက် Docker Desktop က Mac ရဲ့ Background မှာ အင်မတန် ပေါ့ပါးလှတဲ့ Lightweight Linux VM အသေးစားလေး တစ်ခုကို ဖန်တီးပေးလိုက်ပြီး၊ အဲဒီ Linux VM

ထဲမှာပဲ Container တွေကို အဆင်ပြေပြေနဲ့ အသုံးပြုနိုင်အောင် ဖန်တီးပေးလိုက်တာ ဖြစ်ပါတယ်။

လက်ရှိမှာတော့ Mac ပေါ်မှာ Container တွေ အသုံးပြုဖို့အတွက် Docker Desktop တစ်ခုတည်းသာမကဘဲ၊ Podman ဒါမှမဟုတ် Rancher Desktop လိုမျိုး အခြားသော Open-source Tool တွေကလည်း အင်မတန် ကောင်းမွန်တဲ့ ရွေးချယ်စရာတွေ ဖြစ်လာပါပြီ။ ဒါကြောင့် စာဖတ်သူအနေနဲ့ Mac ကို အသုံးပြုနေတယ်ဆိုရင်တောင် ဒီ Tool တွေကို အသုံးပြုပြီး Container နည်းပညာကို အပြည့်အဝ လက်တွေ့ အသုံးချ လေ့လာနိုင်မှာ ဖြစ်ပါတယ်။

၁.၁၀ အခန်းအကျဉ်းချုပ်

ဒီအခန်းမှာ ကျွန်တော်တို့ ဆွေးနွေးခဲ့သမျှကို ပြန်လည် ခြုံငုံကြည့်မယ်ဆိုရင် ကွန်ပျူတာ နည်းပညာလောကရဲ့ ဆင့်ကဲပြောင်းလဲလာတဲ့ သမိုင်းကြောင်းကြီးကို မြင်တွေ့ရမှာပါ။ ကျွန်တော်တို့ဟာ တစ်ချိန်က Application အသစ်တစ်ခု လိုအပ်လာတိုင်း Physical Server အသစ်စက်စက် တစ်လုံးကို မဖြစ်မနေ ဝယ်ယူခဲ့ကြရပါတယ်။ ၊ ဒီလိုစနစ်ကြောင့် ငွေကြေးနဲ့ Resource တွေ အမြောက်အမြား လေလွင့်ဆုံးရှုံးခဲ့ကြပါတယ်။

အဲဒီနောက်မှာတော့ VMware ရဲ့ **Virtual Machine (VM)** နည်းပညာ ပေါ်ပေါက်လာခဲ့ပါတယ်။ VM တွေကြောင့် ရှိပြီးသား Physical Resource တွေကို အကျိုးရှိရှိ ခွဲဝေသုံးစွဲနိုင်တဲ့ အလှည့်အပြောင်းတစ်ခုကို ရောက်ရှိခဲ့ပေမဲ့ VM တစ်ခုချင်းစီမှာ Operating System (OS) တစ်ခုစီ ထည့်သွင်းရတဲ့အတွက် OS Overhead ပြဿနာတွေဖြစ်လာခဲ့ပါတယ်။

ဒီလို VM တွေရဲ့ အားနည်းချက်တွေကို ကျော်လွှားဖို့အတွက် ပိုပြီး ပေါ့ပါးတယ်၊ ပိုမြန်ဆန်တယ်၊ ပြီးတော့ ဘယ်နေရာကိုမဆို အလွယ်တကူ သယ်ဆောင်ရွှေ့ပြောင်းနိုင်တဲ့ Container ဆိုတဲ့ နည်းပညာသစ် တစ်ခု ထပ်မံ ထွက်ပေါ်လာခဲ့ပြန်ပါတယ်။ Container နည်းပညာဟာ အစပိုင်းမှာ Linux Kernel ရဲ့ ရှုပ်ထွေးတဲ့ အလုပ်လုပ်ပုံတွေအပေါ် အခြေခံထား တဲ့အတွက် သာမန် Developer တွေ အသုံးပြုဖို့ အလှမ်းဝေးခဲ့ပါတယ်။ ဒါပေမဲ့ Docker ဆိုတဲ့ နည်းပညာ ဝင်ရောက်လာတဲ့ အချိန်မှာတော့ ဒီရှုပ်ထွေးမှုတွေ အားလုံးကို အသုံးပြုရ လွယ်ကူသွားအောင် ဖြေရှင်းပေး လိုက်တဲ့အတွက်၊ Developer တိုင်းရဲ့ လက်ထဲကို Container ရဲ့ အစွမ်းသတ္တိတွေ အလွယ်တကူ ရောက်ရှိလာခဲ့တာပဲ ဖြစ်ပါတယ်။

ဒီသမိုင်းကြောင်း တစ်လျှောက်လုံးကို ကြည့်လိုက်မယ်ဆိုရင် လူသားတွေအနေနဲ့ နည်းပညာလောက မှာ ပိုမို ထိရောက်ကောင်းမွန်တဲ့ စနစ်တွေ၊ နည်းပညာတွေကို အချိန်နဲ့အမျှ ဘယ်လို ဆန်းသစ် ရှာဖွေ ဖန်တီးနေကြသလဲ ဆိုတာကို ထင်ထင်ရှားရှား မြင်တွေ့နိုင်မှာပဲ ဖြစ်ပါတယ်။

အခန်း (၂) - Docker မိတ်ဆက်

ဒီနေ့ခေတ် Software တွေကို တည်ဆောက်တဲ့ နေရာမှာ Docker ဟာ မပါမဖြစ် အရေးပါဆုံး အစိတ်အပိုင်းတစ်ခု ဖြစ်လာပါပြီ။ Modern Software တွေရဲ့ Build – Ship – Runing ဆိုတဲ့ လုပ်ငန်းစဉ် တစ်လျှောက်လုံးမှာ Docker ဟာ ခွဲခြားလို့မရအောင် ပါဝင်ပတ်သက်နေပါတယ်။

ဒါ့အပြင် Software Developer တွေ အမြဲတမ်း ရင်ဆိုင်ရလေ့ရှိတဲ့ Dependency လိုအပ်ချက် ပြဿနာတွေနဲ့၊ ကွန်ပျူတာ တစ်လုံးနဲ့ တစ်လုံး Environment မတူညီမှုကြောင့် ဖြစ်ပေါ်လာတဲ့ အခက်အခဲတွေကို Docker က Containerization ဆိုတဲ့နည်းပညာကို အသုံးပြုပြီး ထိထိရောက်ရောက် ဖြေရှင်းပေးနိုင်ပါတယ်။

ဒါပေမဲ့ အခုမှ လေ့လာမယ့်သူတွေအတွက် Docker ဆိုတာ ဘာလဲ၊ ဘယ်နေရာမှာ ဘာအတွက် သုံးရမှန်းမသိ၊ နောက်ကွယ်မှာ ဘယ်လို အလုပ်လုပ်မှန်းမသိဘဲ ရှုပ်ထွေးနေနိုင်ပါတယ်။ ဒါကြောင့် ဒီအခန်းမှာ Docker ရဲ့ အခြေခံ သဘောတရားတွေ၊ ဘာကြောင့် ပေါ်ပေါက်လာရသလဲ ဆိုတဲ့ အကြောင်းရင်းတွေနဲ့ သူရဲ့ တည်ဆောက်ပုံ အစိတ်အပိုင်းတွေကို တစ်ဆင့်ချင်းစီ အသေးစိတ် လေ့လာသွားကြပါမယ်။

၂.၁ "ကျွန်တော့်စက်မှာတော့ အလုပ်လုပ်တယ်ဗျ"

Software ရေးသားဖူးသူတိုင်း တစ်ကြိမ်မဟုတ်တစ်ကြိမ် ကြုံတွေ့ရမယ့်၊ အင်မတန် ခေါင်းခဲရတဲ့ ပြဿနာတစ်ခု ရှိပါတယ်။ အဲဒါကတော့ ကိုယ်ရေးထားတဲ့ Application ဟာ ကိုယ့်ရဲ့ ကွန်ပျူတာ (Local Machine) မှာ ဘာအမှားအယွင်းမှ မရှိဘဲ အေးအေးဆေးဆေး အလုပ်လုပ်နေပေမဲ့၊ အဲဒီ Application ကို တခြား Developer တစ်ယောက်ရဲ့ စက်ထဲကို ပြောင်းထည့်လိုက်တဲ့အခါ ဒါမှမဟုတ် တခြား Server တစ်ခုခုပေါ်ကို ရွှေ့လိုက်တဲ့အခါမှာ လုံးဝ အလုပ်မလုပ်တော့ဘဲ Error တွေ တက်လာတာမျိုးပါ။

ဒီလို အခြေအနေမျိုး ကြုံလာရတိုင်း Developer တွေက "ကျွန်တော့်စက်မှာတော့ ပုံမှန် အလုပ်လုပ်နေတာပဲဗျ" ဆိုပြီး အမြဲတမ်း လက်သုံးစကားလို ပြောလေ့ရှိကြပါတယ်။ အဲဒီအခါ "User ဆီကို မင်းစက်ကြီး သွားပို့ထားလို့မှ မရတာ" ဆိုတဲ့ အချင်းချင်း ပြန်လည် နောက်ပြောင်မှုတွေနဲ့အတူ ရယ်မောရခက်၊ ငိုရခက် အခြေအနေမျိုးတွေ ဖြစ်ကြပါတယ်။ ဒါဟာ IT Team တိုင်းမှာ အမြဲလိုလို တွေ့ကြုံနေရတဲ့ အခြေအနေတစ်ခုပါ။

“ဒါဆိုရင် ဘာကြောင့် ဒီလိုမျိုး ကိစ္စတွေက အမြဲတမ်း ဖြစ်နေရတာလဲ။”

အဓိက အကြောင်းရင်းကတော့ Environment Inconsistency လို့ခေါ်တဲ့ ကွန်ပျူတာ တစ်လုံးနဲ့ တစ်လုံးကြားက ပတ်ဝန်းကျင် အခြေအနေ မတူညီမှုတွေ ကြောင့်ပါ။ ဥပမာအားဖြင့် ကိုယ့်စက်မှာ Python Version 3.10 ကို သုံးပြီး Code ရေးထားပေမဲ့ တကယ် Run မယ့် Server မှာက Python 3.8 ပဲ ရှိနေတာမျိုး၊ ကိုယ့်စက်မှာ ထည့်သွင်းထားတဲ့ Library Version နဲ့ Server ပေါ်က Library Version ကွဲလွဲနေတာမျိုးတွေကြောင့် ဖြစ်တတ်ပါတယ်။ ဒါ့အပြင် ကိုယ်က Windows Operating System (OS) ပေါ်မှာ ရေးသားခဲ့ပေမဲ့ တကယ်တမ်း သွား Run မယ့် Server ကြီးက Linux OS ဖြစ်နေတာမျိုးစတဲ့ OS မတူညီမှုတွေနဲ့ လိုအပ်တဲ့ Dependencies တွေ မပြည့်စုံတာမျိုးတွေက အဓိက အကြောင်းရင်းတွေ ဖြစ်ပါတယ်။

ဒီလို သေးငယ်တဲ့ ကွဲလွဲမှုလေးတွေက Application တစ်ခုလုံးရဲ့ အလုပ်လုပ်ပုံကို သိသိသာသာ ထိခိုက်စေပြီး အလုပ်မလုပ်တော့တဲ့အထိ ဖြစ်စေနိုင်ပါတယ်။ ဒီပြဿနာကို ရေရှည်အတွက် ထိထိရောက်ရောက် ဖြေရှင်းဖို့ဆိုရင် Application တစ်ခုတည်းကိုပဲ သီးသန့် ခွဲထုတ်တာမျိုး မလုပ်ဘဲ၊ အဲဒီ Application အလုပ်လုပ်ဖို့အတွက် လိုအပ်တဲ့ Library တွေ၊ Runtime တွေ၊ Dependency တွေ အားလုံးကို တစ်နေရာတည်းမှာ စုစည်းပြီး Package လုပ်လိုက်ဖို့ လိုအပ်လာပါတယ်။ ဒီအခက်အခဲကို အထိရောက်ဆုံး ဖြေရှင်းပေးခဲ့တဲ့ နည်းပညာကတော့ Docker ပဲ ဖြစ်ပါတယ်။

၂.၂ Software တွေအတွက် စံသေတ္တာ

Docker ရဲ့ အလုပ်လုပ်ပုံကို အလွယ်တကူ နားလည်နိုင်ဖို့ဆိုရင် ပင်လယ်ရေကြောင်း သယ်ယူပို့ဆောင်ရေးမှာ အသုံးပြုတဲ့ ကုန်သေတ္တာတွေကို ဥပမာပေးရပါလိမ့်မယ်။

လွန်ခဲ့တဲ့ နှစ်ပေါင်း ၅၀ ကျော်လောက်က၊ အခုလို ကုန်သေတ္တာတွေ မပေါ်ခင်အချိန်တုန်းကဆိုရင် သင်္ဘောပေါ်ကို ကုန်ပစ္စည်းတွေ တင်ဖို့ဆိုတာ အရမ်းကို ခက်ခဲကြန့်ကြာခဲ့ပါတယ်။ ကုန်ပစ္စည်းတွေက အိတ်တွေ၊ စည်ပိုင်းတွေ၊ သေတ္တာအသေးလေးတွေ စသဖြင့် ပုံစံအမျိုးမျိုး၊ အရွယ်အစားအမျိုးမျိုးနဲ့ ဆိပ်ကမ်းကို ရောက်လာတတ်ပါတယ်။ အဲဒီအခါ ကုန်တင်ကုန်ချ လုပ်သားတွေအနေနဲ့ ပစ္စည်းအမျိုးအစားပေါ် မူတည်ပြီး ကိုင်တွယ်ရတဲ့ နည်းလမ်းတွေကလည်း အမျိုးမျိုး ကွဲပြားနေခဲ့ပါတယ်။ ဒါကြောင့်မို့လို့ ပစ္စည်းတင်ရတာ အချိန်အရမ်း ကြန့်ကြာခဲ့သလို၊ လမ်းခရီးမှာ ပျက်စီးဆုံးရှုံးမှုတွေလည်း အများကြီး ရှိခဲ့ပါတယ်။

အဲဒီနောက်မှာတော့ ကုန်စည်ပို့ဆောင်ရေး လောကအတွက် စံသတ်မှတ်ချက် (Standard) တစ်ခု ဖြစ်လာမယ့် Shipping Container လို့ခေါ်တဲ့ ကုန်သေတ္တာတွေ ပေါ်ပေါက်လာပါတယ်။ ဘယ်လိုကုန်ပစ္စည်း အမျိုးအစားပဲ ဖြစ်နေပါစေ၊ ကုန်သေတ္တာတွေထဲကို စနစ်တကျ ထည့်သွင်းပြီး သယ်ယူပို့ဆောင်လိုက်တဲ့အတွက် သင်္ဘောတွေနဲ့ ဝန်ချီစက်တွေ အနေနဲ့ ကိုင်တွယ်ရတာ အဆင်ပြေ လွယ်ကူသွားပါတယ်။

ဒီစနစ်ကြောင့် ကုန်စည်ပို့ဆောင်ရေး လုပ်ငန်းစဉ်ကြီးဟာ အရင်ကထက် အဆပေါင်းများစွာ ပိုမိုမြန်ဆန်၊ လုံခြုံပြီး ထိရောက်လာခဲ့ပါတယ်။

Docker က ဒီသဘောတရားကို Software လောကထဲ ယူဆောင်လာပေးခဲ့ပါတယ်။ Application တစ်ခုလုံး အလုပ်လုပ်ဖို့ လိုအပ်သမျှ အစိတ်အပိုင်းအားလုံးကို Docker က "Container" လို့ခေါ်တဲ့ စနစ်ထဲမှာ စနစ်တကျ ထည့်သွင်း ထုပ်ပိုးပေးလိုက်ပါတယ်။ ဆိုလိုတာက ကျွန်တော်တို့ ရေးသားထားတဲ့ Application Code တွေ အပါအဝင်၊ အဲဒီ Application အလုပ်လုပ်ဖို့အတွက် မဖြစ်မနေ လိုအပ်တဲ့ Libraries တွေ၊ Runtime တွေ၊ System Tools တွေနဲ့ အခြားသော Dependencies တွေ အားလုံးကို စုစည်းပြီး Container လို့ခေါ်တဲ့ Standard Package တစ်ခုအဖြစ်နဲ့ ထုပ်ပိုးပေးလိုက်တာ ဖြစ်ပါတယ်။

ဒီလိုမျိုး လိုအပ်တာတွေ အားလုံးကို စနစ်တကျ ထုပ်ပိုးထားတဲ့ Container တွေကို Docker Install လုပ်ထားတဲ့ ဘယ်လို ကွန်ပျူတာ၊ ဘယ်လို Server မျိုးပေါ်မှာမဆို အလွယ်တကူ အဆင်ပြေပြေ Run လို့ ရသွားပါတယ်။ ဒါကြောင့်မို့လို့ လက်ရှိ Software Deployment လုပ်ငန်းစဉ်ဟာ အရင်ကထက် အဆပေါင်းများစွာ ပိုမို မြန်ဆန်၊ လုံခြုံပြီး ထိရောက်မှု ရှိလာခဲ့တာ ဖြစ်ပါတယ်။

၂.၃ Docker က ဘာလဲ

နည်းပညာလောကထဲမှာ **"Docker"** လို့ ပြောလိုက်ပြီဆိုရင် အကြမ်းဖျင်းအားဖြင့် ရှုထောင့်နှစ်မျိုး၊ အဓိပ္ပာယ် နှစ်မျိုး သက်ရောက်နိုင်ပါတယ်။ **Docker Platform** နဲ့ **Docker, Inc.** ပါ။

Docker Platform ဆိုတာက Application တွေကို Container တွေအဖြစ် ဖန်တီးဖို့၊ စီမံခန့်ခွဲဖို့၊ လိုအပ်သမျှ နည်းပညာတွေကို တစ်နေရာတည်းမှာ သေသေချာချာ ပေါင်းစပ်ထုပ်ပိုးပေးထားတဲ့ Packaged စနစ်တစ်ခု ဖြစ်ပါတယ်။ ကျွန်တော်တို့ Developer တွေ ကုဒ်တွေရေးပြီး Container အဖြစ် ပြောင်းလဲတဲ့ လုပ်ငန်းစဉ်တွေ အားလုံးကို ဒီ Platform ပေါ်မှာပဲ အဓိက အခြေခံပြီး လုပ်ဆောင်သွားကြတာ ဖြစ်ပါတယ်။

Docker, Inc. ဆိုတာကတော့ Docker Platform ကို စတင် တီထွင်ခဲ့တဲ့ ကုမ္ပဏီဖြစ်ပါတယ်။ Container နည်းပညာလောက တစ်ခုလုံးကို ရှေ့တန်းကနေ ဦးဆောင်နေသလို၊ လက်ရှိအချိန်ထိလည်း လိုအပ်တဲ့ Feature အသစ်တွေကို ဆက်တိုက် ဖန်တီးထုတ်လုပ်နေတဲ့ ကုမ္ပဏီကြီးတစ်ခု ဖြစ်ပါတယ်။

၂.၃.၁ Dotcloud မှသည် Docker, Inc. ဆီသို့

Docker, Inc. ရဲ့ ဇာတ်လမ်းဟာ အမေရိကန်နိုင်ငံ၊ Palo Alto မှာ စတင်ခဲ့ပါတယ်။ ဒီကုမ္ပဏီကို ပြင်သစ်နွယ်ဖွား အမေရိကန် Developer

တစ်ဦးဖြစ်တဲ့ Solomon Hykes က စတင် တည်ထောင်ခဲ့ပါတယ်။ စိတ်ဝင်စားစရာအချက်က အစောပိုင်း ကာလတွေကို ပြန်ကြည့်မယ်ဆိုရင် Docker ဟာ အခုလိုမျိုး သီးသန့် Product တစ်ခုအနေနဲ့ ပေါ်ထွက်လာခဲ့တာ မဟုတ်ခဲ့ပါဘူး။ အဲဒီအချိန်တုန်းက Solomon တည်ထောင်ထားတဲ့ ကုမ္ပဏီရဲ့ နာမည်ဟာ Dotcloud ဆိုတဲ့ အမည်နဲ့ ဖြစ်ပါတယ်။ Platform as a Service (PaaS) လို့ခေါ်တဲ့ Cloud ဝန်ဆောင်မှု ပေးနေတဲ့ [1]လုပ်ငန်းတစ်ခုပါပဲ။

အဲဒီ Dotcloud ဟာ သူတို့ရဲ့ PaaS ဝန်ဆောင်မှုတွေကို Container နည်းပညာပေါ်မှာ အခြေခံပြီး တည်ဆောက်ခဲ့ကြတာပါ။ ဒီလို တည်ဆောက်တဲ့ နေရာမှာ အရေအတွက် များပြားလှတဲ့ Container တွေကို

[1] Credit: Will Buckner on Flickr

စနစ်တကျ Deploy လုပ်ဖို့နဲ့ လွယ်လွယ်ကူကူ စီမံခန့်ခွဲနိုင်ဖို့အတွက် ကုမ္ပဏီအတွင်းမှာပဲ သီးသန့်သုံးမယ့် Internal Tool တစ်ခုကို ဖန်တီးခဲ့ကြပါတယ်။ အဲဒီ Tool လေးကို သူတို့က "Docker" လို့ နာမည်ပေးခဲ့ပါတယ်။

ဒီနေရာမှာ ဗဟုသုတအနေနဲ့ "Docker" ဆိုတဲ့ နာမည်လေး ပေါ်လာပုံကို အနည်းငယ် ရှင်းပြချင်ပါတယ်။ ဒီစကားလုံးဟာ ဗြိတိသျှ အသုံးအနှုန်း ဖြစ်တဲ့ "Dock Worker" (ဆိပ်ကမ်း အလုပ်သမား) ဆိုတဲ့ စကားလုံးကနေ ဆင်းသက်လာတာပါ။ ဆိပ်ကမ်းတွေမှာ သဘော်တွေ ပေါ်ကို ကုန်သေတ္တာတွေ တင်တာ၊ ချတာတွေကို လုပ်ဆောင်ပေးတဲ့ အလုပ်သမားတွေကို ခေါ်ဝေါ်တဲ့ အမည်ဖြစ်ပြီး၊ Container တွေကို စနစ်တကျ စီမံခန့်ခွဲပေးမယ့် သူတို့ရဲ့ Tool လေးနဲ့ လုပ်ဆောင်ချက်ချင်း သွားပြီး ဆင်တူနေတဲ့အတွက် အခုလို မှည့်ခေါ်ခဲ့တာ ဖြစ်ပါတယ်။

နောက်ပိုင်း ၂၀၁၃ ခုနှစ်ထဲကို ရောက်တဲ့အခါမှာ သူတို့ ဖန်တီးထားတဲ့ ဒီ Tool လေးဟာ အင်မတန် အစွမ်းထက်ပြီး နည်းပညာလောကကို ပြောင်းလဲပစ်နိုင်စွမ်း ရှိမှန်း နားလည်လာပါတယ်။ ဒါကြောင့် မူလ လုပ်ကိုင်နေတဲ့ Dotcloud ရဲ့ PaaS လုပ်ငန်းကို ရပ်တန့်လိုက်ပြီး၊ ကုမ္ပဏီနာမည်ကိုပါ Docker, Inc. လို့ ပြောင်းလဲလိုက်ပါတယ်။ အဲဒီအချိန်က စပြီး Container နည်းပညာကို တစ်ကမ္ဘာလုံးက အသုံးပြုနိုင်အောင် အပြည့်အဝ အာရုံစိုက်ပြီး ဖြန့်ဝေပေးခဲ့တာဟာ ဒီနေ့ခေတ် နာမည်ကျော် Docker ဆိုပြီး ဖြစ်လာခဲ့တဲ့အထိပါပဲ။

စာအုပ်အပြည့်အစုံကို ဝယ်ယူဖတ်ရှုနိုင်ပါတယ်။

၂.၆ Container က VM မဟုတ်ပါ

ဒီနေရာမှာ စတင်လေ့လာသူတွေ မေးလေ့ရှိတဲ့ မေးခွန်းတစ်ခု ရှိပါတယ်။ အဲဒါကတော့ "Docker Container က Virtual Machine (VM) နဲ့ ဘာကွာသလဲ" ဆိုတဲ့မေးခွန်းပါ။ VM တွေကလည်း ကွန်ပျူတာ စက်တစ်လုံးတည်းမှာ မတူညီတဲ့ OS တွေကို သီးခြားစီ ခွဲပြီး Run ပေးနိုင်တာပဲလို့ တွေးစရာရှိပါတယ်။ ဒီမေးခွန်းရဲ့ အဖြေကို သေချာ နားလည်မှသာ Docker ရဲ့ တကယ့် အစွမ်းသတ္တိနဲ့ ထိရောက်မှုကို သဘောပေါက်မှာ ဖြစ်ပါတယ်။

ပထမဆုံးအနေနဲ့ Virtual Machine (VM) တွေရဲ့ အလုပ်လုပ်ပုံ သဘောတရားကို အရင် ရှင်းပြချင်ပါတယ်။ VM တွေဟာ အခြေခံအားဖြင့် Hardware Level မှာ သွားပြီး အလုပ်လုပ်တာ ဖြစ်ပါတယ်။ ကွန်ပျူတာပေါ်မှာ Hypervisor လို့ ခေါ်တဲ့ Software တစ်ခုကို အရင်ဆုံး Run ရပါတယ်။ ဒီ Hypervisor က စက်ထဲမှာ ရှိတဲ့ CPU, RAM နဲ့ Hard Drive စတဲ့ Hardware တွေကို အစစ်နဲ့ မခြားတူအောင် Virtualize တုပ ဖန်တီးပေးလိုက်တာပါ။

အဲဒီလို တုပထားတဲ့ Virtual Hardware အပေါ်မှာမှ နောက်ထပ် Guest Operating System (OS) တစ်ခုလုံးကို ထပ်တင်ပေးရပါတယ်။ ဥပမာအားဖြင့် Windows သုံးနေတဲ့ စက်ပေါ်မှာ Linux သုံးချင်ရင် Linux OS တစ်ခုလုံးရဲ့ ဖိုင်တွေ၊ Kernel တွေအားလုံးကို အဲဒီ VM ထဲမှာ သီးသန့် ထည့်သွင်းလိုက်ရတာ ဖြစ်ပါတယ်။ ဒါကြောင့်လည်း VM တွေဟာ အင်မတန်

လေးပါတယ်။ အထဲမှာ OS တစ်ခုလုံးကို Run ထားရတဲ့အတွက် ကွန်ပျူတာနဲ့ RAM နဲ့ Storage ကိုလည်း အမြောက်အများ သုံးစွဲပစ်သလို၊ Boot တက်ဖို့ဆိုရင်လည်း ကွန်ပျူတာတစ်လုံး အသစ်စတင်သလိုမျိုး မိနစ်နဲ့ချီပြီး စောင့်ဆိုင်းရလေ့ ရှိပါတယ်။

Container တွေကတော့ VM တွေနဲ့ ချဉ်းကပ်ပုံ လုံးဝ မတူပါဘူး။ Container တွေဟာ Hardware ကို Virtualize လုပ်တာမျိုး မဟုတ်ဘဲ Operating System Level မှာပဲ အလုပ်လုပ်တာ ဖြစ်ပါတယ်။ Docker Container တွေဟာ သူတို့ Run နေတဲ့ Host OS ရဲ့ Kernel ကိုပဲ တိုက်ရိုက် မျှဝေသုံးစွဲကြတာ ဖြစ်ပါတယ်။ ဒီလို မျှဝေသုံးစွဲလိုက်တဲ့အတွက် Container ထဲမှာ Guest OS ကြီးတစ်ခုလုံးကို ထပ်တင်စရာ မလိုတော့ပါဘူး။ Application နဲ့ သူ့အတွက် လိုအပ်တဲ့ Dependencies (Binaries/Libraries) တွေကိုပဲ သီးသန့်ခွဲထုတ်ပြီး Isolated Environment တစ်ခုအဖြစ် ဖန်တီးပေးလိုက်တာပါ။ အခုလိုမျိုး OS ကြီးတစ်ခုလုံး ထပ်တင်ရတဲ့ OS Overhead ပြဿနာ မရှိတော့တဲ့အတွက် Container တွေဟာ VM တွေနဲ့ ယှဉ်ရင် အင်မတန်ပေါ့ပါး ပါတယ်။ စက္ကန့်ပိုင်းအတွင်းမှာတင် အသင့်အသုံးပြုနိုင်အောင် Boot တက်နိုင်သလို၊ RAM နဲ့ Storage သုံးစွဲမှုကလည်း VM တွေနဲ့ ယှဉ်ရင် အဆပေါင်းများစွာ သက်သာသွားပါတယ်။

ဒီနှစ်ခုရဲ့ ခြားနားချက်ကို မြင်သာအောင် ဥပမာပေးရရင်၊ VM ဆိုတာက ဧည့်သည်တစ်ယောက်ကို ဧည့်ခံဖို့အတွက် ကိုယ့်ခြံထဲမှာ ရေ၊ မီး အစုံအလင်နဲ့ အိမ်အသစ်တစ်လုံး ထပ်ဆောက်ရတာနဲ့ တူပါတယ်။

ကုန်ကျစရိတ်လည်း များသလို ပြင်ဆင်ရတာလည်း အချိန်ကြာပါတယ်။ Container ကတော့ အဲဒီဧည့်သည်ကို ကိုယ့်အိမ်မကြီးပေါ်မှာပဲ အသင့်ပြင်ဆင်ပြီးသား အခန်းတစ်ခန်း သီးသန့် နေရာချပေးလိုက်တာမျိုးပါ။ ဧည့်သည်ကတော့ သူ့အခန်းထဲမှာ သူ လွတ်လွတ်လပ်လပ် နေနိုင်ပေမဲ့ အိမ်ရဲ့ အခြေခံ ရေ၊ မီး တွေကိုတော့ အိမ်ရှင်နဲ့အတူ ဝေမျှ သုံးစွဲနေရသလိုမျိုးပဲ ဖြစ်ပါတယ်။

၂.၇ အခန်းအကျဉ်းချုပ်

နိဂုံးချုပ်အနေနဲ့ ဒီအခန်းကို ပြန်လည်သုံးသပ်ရမယ်ဆိုရင် Docker ဆိုတာဟာ Software လောကမှာ ဆယ်စုနှစ်ပေါင်းများစွာ အမြစ်တွယ်ခဲ့တဲ့ “It Works On My Machine” ဆိုတဲ့ အကြီးမားဆုံးသော အိပ်မက်ဆိုးကို အပြီးတိုင် ဖြေရှင်းပေးဖို့ ထွက်ပေါ်လာခဲ့တဲ့ နည်းပညာတစ်ခုလို့ ဆိုရမှာပါ။ တစ်ချိန်က Application တွေကို မတူညီတဲ့ Server တွေပေါ် တင်တဲ့အခါ Environment ကွဲလွဲမှုတွေကြောင့် ကြုံတွေ့ရတဲ့ အခက်အခဲတွေကို၊ Docker က Containerization ဆိုတဲ့ စနစ်ကို အသုံးပြုပြီး Software တွေအတွက် စံသတ်မှတ်ချက် တစ်ခုတည်းအောက်မှာ စနစ်တကျ စုစည်းပေးပြီး တော်လှန်ပြောင်းလဲပေးနိုင်ခဲ့ပါတယ်။

Docker ရဲ့ ပေါ့ပါးသွက်လက်မှု၊ Virtual Machine (VM) တွေထက် Resource အသုံးပြုမှု အဆပေါင်းများစွာ သက်သာမှု စတဲ့ အားသာချက်တွေဟာ အကြောင်းမဲ့ ပေါ်ပေါက်လာတာ မဟုတ်ပါဘူး။

Docker Image နဲ့ Docker Container ဆိုတဲ့ ရိုးရှင်းပြီး ထိရောက်လှတဲ့ တည်ဆောက်ပုံစနစ်တွေကြောင့်သာ ဒီလို ကြီးမားတဲ့ အကျိုးကျေးဇူးတွေကို ရရှိလာတာ ဖြစ်ပါတယ်။

ဒီလို ဖွဲ့စည်းတည်ဆောက်ထားတဲ့ စနစ်တွေကြောင့်ပဲ သာမန် Developer တစ်ယောက်တည်း ရေးသားထားတဲ့ ကုဒ်တွေကနေစလို့၊ ကုမ္ပဏီကြီးတွေရဲ့ ရှုပ်ထွေးတဲ့ Software Development လုပ်ငန်းစဉ်တွေ၊ DevOps Pipeline တွေအထိ၊ Application တွေကို ဘယ်နေရာ၊ ဘယ်လို ကွန်ပျူတာ စက်ပေါ်မှာမဆို တသမတ်တည်း ပုံမှန် အလုပ်လုပ်နိုင်တယ် ဆိုတဲ့ ခိုင်မာတဲ့ အာမခံချက်ကို အပြည့်အဝ ပေးစွမ်းနိုင်ခဲ့တာပဲ ဖြစ်ပါတယ်။ ဒါကြောင့်မို့လို့ Docker ဟာ ခေတ်သစ် နည်းပညာလောကမှာ မရှိမဖြစ် အရေးပါတဲ့ အစိတ်အပိုင်းတစ်ခု အဖြစ် အခိုင်အမာ ရပ်တည်နေနိုင်တာ ဖြစ်ပါတယ်။

အခန်း (၃) - Docker ထည့်သွင်းခြင်း

ဒီအခန်းမှာ Docker ကို ကွန်ပျူတာထဲမှာ စနစ်တကျ ထည့်သွင်းပြီး လက်တွေ့အသုံးပြုနိုင်မယ့် Environment တစ်ခု အသင့်ဖြစ်လာအောင် အသေးစိတ် ပြင်ဆင်သွားကြပါမယ်။ လေ့လာသူတွေ သတိထားရမယ့် အချက်က၊ Docker ကို Install လုပ်တဲ့ နေရာမှာ မိမိ အသုံးပြုနေတဲ့ Operating System (OS) အမျိုးအစားပေါ် မူတည်ပြီး လုပ်ဆောင်ရမယ့် အဆင့်တွေဟာ တစ်ခုနဲ့တစ်ခု မတူညီဘဲ ကွဲပြားမှုတွေ ရှိပါတယ်။ ဒါကြောင့် Windows သုံးတဲ့သူတွေ၊ macOS သုံးတဲ့သူတွေနဲ့ Linux အသုံးပြုသူတွေ အားလုံးအတွက် အဆင်ပြေပြေ အလွယ်တကူ လိုက်လုပ်နိုင်မယ့် တဆင့်ချင်း လမ်းညွှန်ချက်တွေကို ဖော်ပြပေးသွားမှာ ဖြစ်ပါတယ်။

အကယ်၍ စာဖတ်သူက ဒီစာအုပ်ထဲမှာ ဖော်ပြမထားတဲ့ အခြား OS တွေ (သို့မဟုတ်) Linux Distribution တွေ အသုံးပြုနေတယ်ဆိုရင်တော့ Docker ရဲ့ Official Installation Guide မှာ အသေးစိတ် သွားရောက်ကြည့်ရှုနိုင်ပါတယ်။

နည်းပညာတွေက အမြဲတမ်း အသစ်သစ်တွေ ပြောင်းလဲနေတာကြောင့် ဒီစာအုပ်မှာ အသုံးပြုထားတဲ့ Version တွေနဲ့ နောင်မှာ ထွက်လာမယ့် Version အနည်းငယ် ကွဲပြားတာမျိုး ရှိနိုင်ပါတယ်။ ဒါပေမဲ့ Docker ရဲ့ အခြေခံ အလုပ်လုပ်ပုံ သဘောတရားတွေကတော့ အတူတူပဲဖြစ်လို့ စိုးရိမ်စရာမလိုဘဲ ဆက်လက်လေ့လာလို့ ရပါတယ်။

စာအုပ်အပြည့်အစုံကို ဝယ်ယူဖတ်ရှုနိုင်ပါတယ်။

အခန်း (၄) - Docker Image များ၏ အလုပ်လုပ်ပုံနှင့် သဘောတရား

ရှေ့က အခန်းတွေမှာ ကျွန်တော်တို့ဟာ Docker နည်းပညာရဲ့ အခြေခံ သဘောတရားတွေ၊ နောက်ကွယ်က အလုပ်လုပ်တဲ့ Architecture တည်ဆောက်ပုံတွေအပြင်၊ မိမိရဲ့ ကွန်ပျူတာပေါ်မှာ လက်တွေ့ ဘယ်လို Install လုပ်ပြီး အသင့်ပြင်ဆင်ရမလဲ ဆိုတဲ့ အကြောင်းအရာတွေကို အသေးစိတ် ဆွေးနွေး လေ့လာခဲ့ကြပြီး ဖြစ်ပါတယ်။ အခု ဒီအခန်းမှာတော့ Docker နည်းပညာရဲ့ အဓိက အနှစ်သာရဖြစ်တဲ့ Docker Image တွေအကြောင်းကို စတင် လေ့လာသွားကြပါမယ်။

၄.၁ Image ဆိုတာ ဘာလဲ

Docker Image တွေအကြောင်း ပြောကြမယ်ဆိုရင် မစခင်မှာ အရင်ဆုံး ရှင်းလင်းထားသင့်တဲ့ အသုံးအနှုန်းလေးတွေ ရှိပါတယ်။ Image, Docker Image, Container Image နဲ့ OCI Image ဆိုပြီး အမျိုးမျိုး ခေါ်ဝေါ်ကြပေမယ့် တကယ်တမ်းတော့ ဒီအသုံးအနှုန်း အားလုံးဟာ တူညီတဲ့ အရာတစ်ခုတည်းကိုပဲ ရည်ညွှန်းတာပါ။ Docker Image တစ်ခုကို အဓိပ္ပာယ်ဖွင့်ဆိုရမယ် ဆိုရင် “Application တစ်ခု Run ဖို့ လိုအပ်သမျှ အရာအားလုံး ပါဝင်နေတဲ့ Read-only Template တစ်ခု” လို့ အတိအကျ ပြောရမှာ ဖြစ်ပါတယ်။ "လိုအပ်သမျှ အရာအားလုံး" ဆိုတဲ့ စကားရပ်မှာ

ကျွန်တော်တို့ ရေးသားထားတဲ့ Source Code တွေ၊ အဲဒီ Code ကို Run ဖို့ လိုအပ်တဲ့ Runtime တွေ၊ Library တွေ၊ Environment Variables တွေနဲ့ Configuration File တွေ အားလုံး ပါဝင်ပါတယ်။ Image တစ်ခုတည်းကနေ Container အများကြီးကို ပွားပြီး Start လုပ်နိုင်ပါတယ်။

အကယ်၍ စာဖတ်သူဟာ VMware လို Virtual Machine နည်းပညာတွေနဲ့ ရင်းနှီးသူဖြစ်မယ်ဆိုရင် Docker Image ဟာ "VM Template" နဲ့ အတော်လေး ဆင်တူပါတယ်။ VM Template ဆိုတာ ရပ်ထားတဲ့ VM တစ်ခုဖြစ်သလို၊ Image ဆိုတာကလည်း ရပ်ထားတဲ့ Container တစ်ခုပါပဲ။

Programming မှာ Object-Oriented Programming (OOP) ကို ရင်းနှီးပြီးသားဆိုရင်တော့ Docker Image ကို Class တစ်ခုအဖြစ် မြင်ယောင်ကြည့်လို့ ရပါတယ်။ Container ကတော့ အဲဒီ Class ကနေ ဖန်တီးလိုက်တဲ့ Object နဲ့ တူပါတယ်။ Class တစ်ခုတည်းကနေ Object ပေါင်းများစွာ ဖန်တီးနိုင်သလို၊ Docker Image တစ်ခုတည်းကနေလည်း ထပ်တူညီတဲ့ Container ပေါင်းများစွာကို စိတ်ကြိုက် ဖန်တီးနိုင်ပါတယ်။

စာအုပ်အပြည့်အစုံကို ဝယ်ယူဖတ်ရှုနိုင်ပါတယ်။

၄.၂ Local Repository သို့မဟုတ် Image Cache အကြောင်း

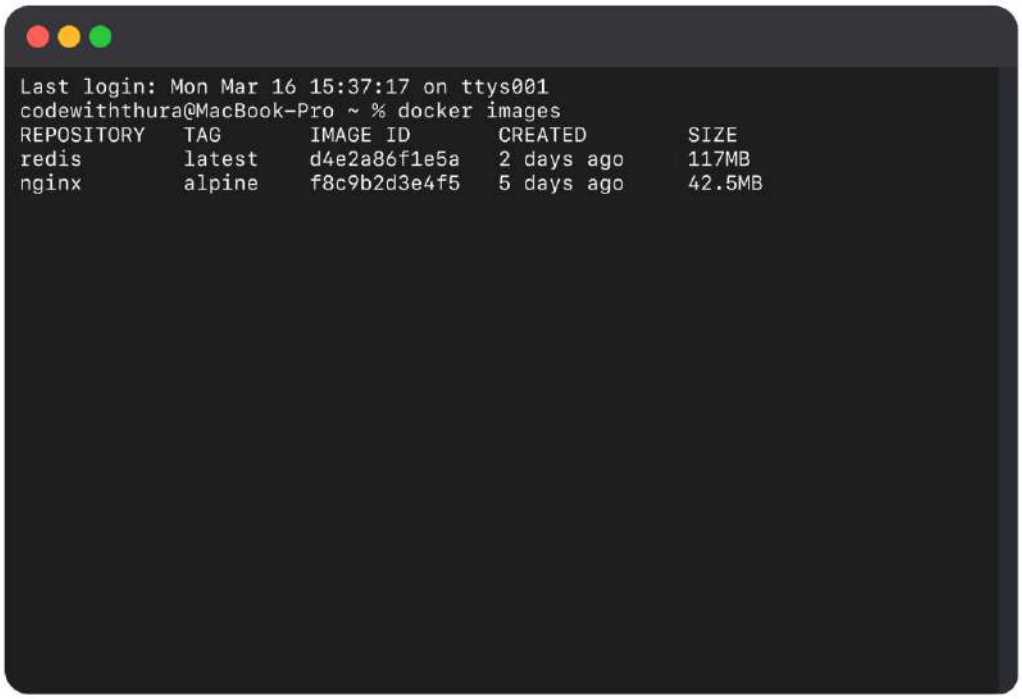
ကွန်ပျူတာထဲမှာ Docker ကို စတင် အသုံးပြုတဲ့အခါ မဖြစ်မနေ သိထားရမယ့် အစိတ်အပိုင်း တစ်ခုက Local Repository ပါ။ နာမည်အရ အနည်းငယ် ရှုပ်ထွေးနိုင်ပေမဲ့၊ ရိုးရိုးရှင်းရှင်း ပြောရရင် ကိုယ့်ရဲ့ Local Machine ထဲမှာ Docker Image တွေကို နောက်တစ်ခေါက် အလွယ်တကူ ပြန်လည် အသုံးပြုလို့ရအောင် အသင့် သိမ်းဆည်းပေးထားတဲ့ နေရာတစ်ခုပါပဲ။ ဒီနေရာကို တချို့က **"Image Cache"** လို့ ခေါ်ကြပါတယ်။

အကယ်၍ စာဖတ်သူရဲ့ ကွန်ပျူတာထဲမှာ Docker ကို အသစ်စက်စက် Installation လုပ်ပြီးပြီးချင်း အခြေအနေဆိုရင်တော့၊ ဒီ Local Repository ထဲမှာ ဘာ Image မှ ရှိမနေသေးဘဲ ပြောင်ရှင်းနေမှာ ဖြစ်ပါတယ်။

ဗဟုသုတအနေနဲ့ ဒီနေရာဟာ ကွန်ပျူတာရဲ့ ဘယ်နားမှာ ရှိနေသလဲဆိုတာ သိထားဖို့ လိုအပ်ပါတယ်။ တကယ်လို့ Linux Operating System ကို အသုံးပြုနေတာ ဆိုရင်တော့ ဒီ Local Repository ဟာ /var/lib/docker/<storage-driver> ဆိုတဲ့ လမ်းကြောင်းအောက်မှာ ရှိလေ့ရှိပါတယ်။ ဒါပေမဲ့ Windows နဲ့ macOS တွေမှာ Docker Desktop ကို အသုံးပြုနေတယ် ဆိုရင်တော့၊ ရှေ့အခန်းတွေမှာ ရှင်းပြခဲ့တဲ့ Docker VM (Virtual Machine) ရဲ့ အတွင်းပိုင်းမှာပဲ ရှိနေမှာ ဖြစ်ပါတယ်။

မိမိရဲ့ ကွန်ပျူတာထဲမှာ လက်ရှိ အချိန်အထိ ဘာ Docker Image တွေ ရှိနေပြီလဲ ဆိုတာကို လက်တွေ့ စမ်းသပ် စစ်ဆေးကြည့်ချင်ရင်တော့ Terminal မှာ အောက်က Command ကို ရိုက်ထည့်ပြီး ကြည့်ရှုနိုင်ပါတယ်။

docker images



အပေါ်က ပြထားတဲ့ ဥပမာ Output မှာဆိုရင်တော့၊ လက်ရှိ စက်ထဲမှာ Image (၂) ခု ရှိနေတာကို တွေ့မြင်ရမှာပါ။ စာဖတ်သူ ကိုယ်တိုင် လက်တွေ့ ရိုက်ကြည့်တဲ့အခါ စက်ထဲက အခြေအနေပေါ် မူတည်ပြီး ဒီပုံစံအတိုင်း အတိအကျ ထွက်လာမှာ မဟုတ်ဘဲ ကွဲပြားနိုင်ပါတယ်။ အခုမှ စသွင်းထားတာဆိုရင် ဘာမှ ပေါ်လာမှာ မဟုတ်တာကိုလည်း သတိပြုပေးပါ။

စာအုပ်အပြည့်အစုံကို ဝယ်ယူဖတ်ရှုနိုင်ပါတယ်။

၄.၆ Tag များကို အသုံးပြု၍ Version သတ်မှတ်ခြင်းစနစ်

Docker Image တွေကို စီမံခန့်ခွဲတဲ့နေရာမှာ နာမည်ပေးတာနဲ့ ဗားရှင်းခွဲခြားတာဟာ Developer တစ်ယောက်အတွက် အနုပညာတစ်ခု လိုပါပဲ။ တော်တော်လေး အရေးကြီးတဲ့ အပိုင်း ဖြစ်ပါတယ်။ စနစ်တကျ မရှိတဲ့ Tagging စနစ်ဟာ ရေရှည်မှာ Project ကြီးလာတာနဲ့အမျှ ရှုပ်ထွေးမှုတွေ၊ မလိုလားအပ်တဲ့ အမှားအယွင်း Deployment Error တွေကို ဖြစ်ပေါ်စေနိုင်ပါတယ်။

ဒါကြောင့် ပရော်ဖက်ရှင်နယ် Developer ကောင်း တစ်ယောက်အနေနဲ့ မိမိတို့ရဲ့ ကိုယ်ပိုင် Image တွေကို ဖန်တီးတဲ့အခါ Semantic Versioning စနစ်ကို အသုံးပြုပြီး စနစ်တကျ Tag နံပါတ်တွေ တပ်ဖို့ အကြံပြုချင်ပါတယ်။ ဒီစနစ်က အခြေခံအားဖြင့် Major.Minor.Patch ဆိုတဲ့ ပုံစံနဲ့ သွားပါတယ်။ သူ့ရဲ့ သဘောတရားတွေကို အသေးစိတ် ရှင်းပြရရင် -

- v1.0.0 (Major) ဒါကတော့ ပထမဆုံး Stable Release ပါ။ Major Version ပြောင်းလဲတယ်ဆိုတာ အရင်ဗားရှင်းဟောင်းတွေနဲ့ တွဲဖက်အသုံးပြုလို့ မရတော့လောက်အောင် Breaking Changes တွေ ပြုလုပ်တဲ့အခါမျိုးမှာ သုံးပါတယ်။ API တွေ ပြောင်းသွားတာ၊ Architecture ကြီးတစ်ခုလုံး ပြောင်းသွားတာမျိုးပါ။
- v1.1.0 (Minor): ဒါကတော့ နဂိုရှိပြီးသား လုပ်ဆောင်ချက်တွေကို မထိခိုက်စေဘဲ Feature အသစ်တွေ (Backward Compatible New

Feature တွေ) ထပ်ထည့်လိုက်တဲ့အခါ သုံးပါတယ်။ ဥပမာ၊ Function အသစ်တစ်ခု ထပ်တိုးတာမျိုးပါ။

- v1.0.1 (Major): ဒါကတော့ Feature အသစ်ကြီးကြီးမားမား မပါဘဲ ကုဒ်ထဲက Bug အသေးစားလေးတွေကို ပြင်ဆင်လိုက်တဲ့အခါ၊ Bug Fix လုပ်လိုက်တဲ့အခါမျိုးမှာ သုံးပါတယ်။

ဒီလိုမျိုး Semantic Versioning စနစ်ကို အသုံးပြုပြီး စနစ်တကျ စနစ်တကျ Tag တပ်ခြင်းအားဖြင့် ကိုယ့် Application နဲ့ ကိုက်ညီမယ့် ဗားရှင်းကို ယုံကြည်စိတ်ချစွာ ရွေးချယ် အသုံးပြုနိုင်မှာ ဖြစ်ပါတယ်။ ဒါ့အပြင် အဖွဲ့လိုက် လုပ်ဆောင်တဲ့နေရာမှာလည်း ဗားရှင်း အလွဲအမှားကြောင့် ဖြစ်ပေါ်လာမယ့် Communication Error တွေကို အများကြီး လျှော့ချပေးနိုင်မှာ ဖြစ်ပါတယ်။

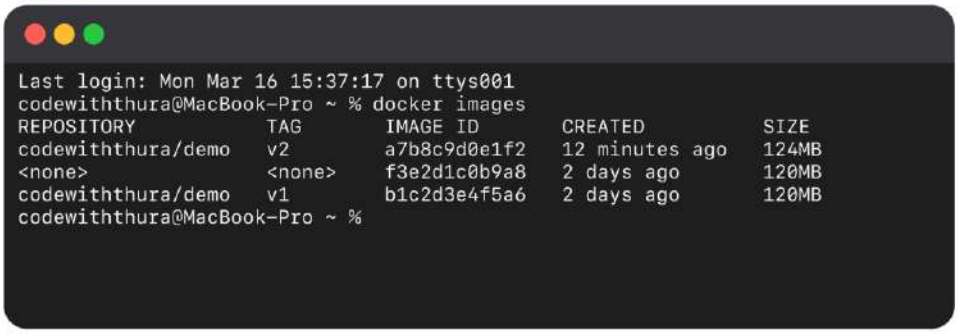
၄.၇ The "Dangling" Images များ

ဒီနေရာမှာ စိတ်ဝင်စားစရာကောင်းပြီး သတိပြုရမယ့် အခြေအနေတစ်ခု ရှိလာပါတယ်။ အဲဒါကတော့ Dangling Images လို့ခေါ်တဲ့ အခြေအနေပါ။ ဖြစ်လေ့ဖြစ်ထရှိတဲ့ ဥပမာတစ်ခုနဲ့ ပြောပြချင်ပါတယ်။

စက်ထဲမှာ ubuntu:latest ဆိုတဲ့ Image ရှိနေတယ်လို့ ဆိုကြပါစို့။ နောက်တစ်ပတ်လောက် ကြာတဲ့အခါ docker pull ubuntu:latest ဆိုတဲ့ command ကိုပဲ ထပ်ပြီး ရိုက်လိုက်ပါတယ်။ Docker Hub ဘက်မှာ Ubuntu

Version အသစ်ထွက်နေပြီဆိုရင် စက်ထဲကို Image အသစ်တစ်ခု ရောက်ရှိလာပါလိမ့်မယ်။ ဒီအခါမှာ Docker က ubuntu:latest ဆိုတဲ့ Name Tag ကို ခုနက အသစ်ရောက်လာတဲ့ Image ဆီကို လွှဲပြောင်းတပ်ပေးလိုက်ပါတယ်။

ဒါဆိုရင် မေးစရာရှိတာက ယခင်တုန်းက ubuntu:latest လို့ အမည်ပေးခဲ့တဲ့ Image အဟောင်းက ဘာဖြစ်သွားမလဲ ဆိုတာပါ။ အဖြေကတော့ အဲဒီ Image ဟာ ဖျက်ခံလိုက်ရတာ မဟုတ်ပါဘူး။ သူ့ရဲ့ ကိုယ်ပိုင် Image ID နဲ့ ဆက်လက်တည်ရှိနေဆဲပါပဲ။ ဒါပေမဲ့ သူ့ကို ညွှန်းထားတဲ့ Tag နာမည်က Image အသစ်ဆီကို ရောက်သွားတဲ့အတွက် သူဟာ "အမည်မရှိ၊ Tag မရှိတဲ့" (Untagged) အခြေအနေကို ရောက်သွားပါတယ်။ အောက်က ပုံကို ကြည့်ပါ။



စာအုပ်အပြည့်အစုံကို ဝယ်ယူဖတ်ရှုနိုင်ပါတယ်။

အခန်း (၆) - Dockerfile ဖြင့် Image များကို အဆင့်ဆင့် တည်ဆောက်ခြင်း

ပြီးခဲ့တဲ့ အခန်းတွေမှာ Docker Image ရဲ့ အခြေခံ သဘောတရားတွေကို အကျယ်တဝံ့ လေ့လာခဲ့ကြပါတယ်။ အထူးသဖြင့် အခန်း (၄) မှာတုန်းက Image ဆိုတာ ပြုပြင်ပြောင်းလဲလို့မရတဲ့ Read-only Template တွေဖြစ်ကြောင်း ဆွေးနွေးခဲ့သလို၊ Docker Hub လိုမျိုး Public Registry တွေကနေ Image တွေကို ဘယ်လို Pull လုပ်ပြီး ယူသုံးရမလဲဆိုတာကိုလည်း လက်တွေ့ လေ့လာခဲ့ကြပါတယ်။

ဒါပေမဲ့ ဒီနေရာမှာ စဉ်းစားစရာ တစ်ခု ရှိလာပါတယ်။ Docker Hub မှာရှိတဲ့ Image တွေဟာ လူတိုင်း ဆွဲယူ အသုံးပြုနိုင်အောင် ဖန်တီးထားတာ ဖြစ်တဲ့အတွက် ကိုယ့်ရဲ့ ပရောဂျက်မှာ လိုအပ်တဲ့ သီးသန့် Setting တွေ ဒါမှမဟုတ် Configuration တွေနဲ့ အမြဲတမ်း တစ်ထပ်တည်း ကျချင်မှ ကျပါလိမ့်မယ်။ အဲဒီအခါမှာ ကိုယ့်ရဲ့ လိုအပ်ချက်နဲ့ အတိအကျ ကိုက်ညီမယ့် ကိုယ်ပိုင် Custom Image တွေကို ဖန်တီးဖို့ လိုအပ်လာပါတယ်။

အခု ဒီအခန်းမှာတော့ ကျွန်တော်တို့ ရေးထားတဲ့ Source Code တွေကို Container တွေအဖြစ် ပြောင်းလဲပေးနိုင်ဖို့ အခြေခံ အကျဆုံးဖြစ်တဲ့ Dockerfile အကြောင်းကို စတင် လေ့လာသွားကြမှာ ဖြစ်ပါတယ်။ Dockerfile ဆိုတာ Image တစ်ခု တည်ဆောက်ဖို့အတွက် လိုအပ်တဲ့ ညွှန်ကြားချက်တွေကို စုစည်းထားတဲ့ စာသားဖိုင် တစ်ခုပါပဲ။ Dockerfile မှာ အသုံးပြုရတဲ့ Instruction တစ်ခုချင်းစီရဲ့ အဓိပ္ပာယ်တွေ၊ Image တစ်ခုကို

စာအုပ်အပြည့်အစုံကို ဝယ်ယူဖတ်ရှုနိုင်ပါတယ်။

အခန်း (၉) - Docker Compose ဖြင့် ရှုပ်ထွေးသော Multi-Container App များကို စီမံခြင်း

၉.၁ Docker Compose ကို ဘာကြောင့် အသုံးပြုဖို့ လိုအပ်လာတာလဲ။

ရှေ့အခန်းတွေတုန်းက ကျွန်တော်တို့ Container တစ်လုံးချင်းစီကို docker run သုံးပြီး ဘယ်လို Run ရမလဲဆိုတာ လေ့လာခဲ့ကြပါတယ်။ အဲဒီအပြင် Data တွေ မပျောက်ပျက်အောင် Volume တွေ ချိတ်ဆက်တာ၊ Container တွေ အချင်းချင်း ဆက်သွယ်နိုင်ဖို့ Network တွေ ဖန်တီးတာကိုပါ လက်တွေ့ လုပ်ဆောင်ခဲ့ကြပါတယ်။ ဒါတွေက Docker ရဲ့ အခြေခံအကျဆုံးနဲ့ အရေးအပါဆုံး အစိတ်အပိုင်းတွေ ဖြစ်ပါတယ်။

ဒါပေမဲ့ တကယ့်လက်တွေ့ လုပ်ငန်းခွင်မှာ Application တစ်ခုလုံးကို Container တစ်လုံးတည်းနဲ့ အစအဆုံး အလုပ်လုပ်နေတာမျိုးက အလွန်ရှားပါတယ်။ လက်တွေ့ Application တော်တော်များများမှာ Frontend ပိုင်း၊ Backend API ပိုင်းနဲ့ Database ပိုင်းဆိုပြီး အစိတ်အပိုင်းခွဲပြီး တည်ဆောက်လေ့ရှိပါတယ်။ ဒီလို အစိတ်အပိုင်းတွေကို Container တစ်လုံးစီ ခွဲပြီး Run မယ်ဆိုရင် အနည်းဆုံး Container သုံးလုံးလောက်ကို တစ်ပြိုင်နက်တည်း အလုပ်လုပ်ခိုင်းဖို့ လိုအပ်လာပါပြီ။ ဒီလို Multi-Container တွေ ပါဝင်လာတဲ့အခါ ရှေ့ပိုင်းမှာ ကျွန်တော်တို့ သုံးခဲ့တဲ့ နည်းလမ်းဟောင်းတွေနဲ့ စီမံခန့်ခွဲဖို့ဆိုတာ တဖြည်းဖြည်း ခက်ခဲလာပါလိမ့်မယ်။

စာအုပ်အပြည့်အစုံကို ဝယ်ယူဖတ်ရှုနိုင်ပါတယ်။

၉.၅ လက်တွေ့ စမ်းသပ်ခြင်း (၄)

Todo Application အတွက် Docker Compose ဖိုင် ရေးသားခြင်း

အခန်း (၈) တုန်းက ကျွန်တော်တို့ Python Flask ကို အသုံးပြုပြီး Todo Application တစ်ခုကို လက်တွေ့ တည်ဆောက်ခဲ့ကြပါတယ်။ အဲဒီတုန်းကတော့ Application တစ်ခုလုံးကို Dockerfile တစ်ခုတည်းနဲ့ Container တစ်လုံးတည်းမှာပဲ Run ခဲ့တာ ဖြစ်ပါတယ်။ အခုအခန်းမှာတော့ အဲဒီ Application လေးကိုပဲ တကယ့် လက်တွေ့ လုပ်ငန်းခွင်သုံး Production-ready ပုံစံမျိုးဖြစ်အောင် Container အများကြီးနဲ့ စနစ်တကျ ပြန်လည် တည်ဆောက်သွားမှာပါ။

ဒီလို တည်ဆောက်တဲ့နေရာမှာ နားလည်ရ ခက်ခဲမသွားအောင် Frontend နဲ့ Backend ကို သီးခြားစီ ခွဲမထုတ်တော့ဘဲ မူလ Flask Application ကိုပဲ ဆက်ပြီး အသုံးပြုသွားပါမယ်။ ဒါပေမဲ့ အဓိက ပြောင်းလဲသွားမယ့် အချက်ကတော့ အပြင်ကလာတဲ့ Request တွေကို စနစ်တကျ လက်ခံပေးဖို့ Reverse Proxy အဖြစ် Nginx ကို ထပ်မံ ထည့်သွင်းသွားပါမယ်။ ဒါကြောင့် အခု ကျွန်တော်တို့ တည်ဆောက်မယ့် စနစ်သစ်မှာ Nginx ၊ Flask Application နဲ့ PostgreSQL ဆိုပြီး အဓိက Container သုံးလုံး ပါဝင်လာမှာ ဖြစ်ပါတယ်။

အဆင့် (၁) ဖိုဒါဖွဲ့စည်းပုံ စတင် ပြင်ဆင်ခြင်း

ဒီလို Container တွေ များလာတဲ့အခါမှာ ဖိုင်တွေကို ရှုပ်ရှုပ်ထွေးထွေး မဖြစ်စေဖို့အတွက် Project Folder ကို အောက်ကအတိုင်း စနစ်တကျ အရင်ဆုံး ဖွဲ့စည်းပေးရပါမယ်။

```
todo-app/
├── backend/
│   ├── Dockerfile
│   ├── app.py
│   ├── requirements.txt
│   └── ...
├── nginx/
│   ├── nginx.conf
│   └── Dockerfile
└── docker-compose.yml
```

အပေါ်မှာ ဖွဲ့စည်းပုံကို ကြည့်လိုက်ရင် todo-app ဆိုတဲ့ Folder အောက်မှာ backend နဲ့ nginx ဆိုပြီး Folder နှစ်ခု သီးခြား ခွဲထုတ်ထားတာကို တွေ့ရမှာပါ။ backend Folder ထဲမှာတော့ Flask Application အတွက်

လိုအပ်မယ့် Code တွေနဲ့ သူ့အတွက် သီးသန့် Dockerfile တို့ ပါဝင်မှာ ဖြစ်ပါတယ်။ အလားတူပဲ nginx Folder ထဲမှာလည်း Nginx အတွက် လိုအပ်တဲ့ Configuration ဖိုင်နဲ့ Dockerfile တို့ကို ထည့်ထားရပါမယ်။

နောက်ဆုံး အနေနဲ့ ဒီ Container တွေ အားလုံးကို တစ်နေရာတည်းကနေ စုစည်း ထိန်းချုပ်ပေးမယ့် docker-compose.yml ဖိုင်ကိုတော့ Project ရဲ့ Root Directory မှာ ထားရှိပေးရမှာ ဖြစ်ပါတယ်။ ဒီလို ဖိုင်တွေကို သူ့နေရာနဲ့သူ စနစ်တကျ ခွဲခြားထားခြင်းအားဖြင့် Application ကြီးလာတဲ့အခါမှာ ပြုပြင်ထိန်းသိမ်းရတာ အလွန် လွယ်ကူသွားစေမှာပါ။

အဆင့် (၂) Nginx ကို Reverse Proxy အဖြစ် အသုံးပြုခြင်း

အခန်း ၈ မှာတုန်းက Backend Application နဲ့ Database အတွက် လိုအပ်တဲ့ ဖိုင်တွေ အားလုံးကို ကျွန်တော်တို့ အပြည့်အစုံ ပြင်ဆင် ဖန်တီးခဲ့ပြီး ဖြစ်ပါတယ်။ ဒါကြောင့် အဲဒီဖိုင်တွေကို အခု ဒီနေရာမှာ အသင့် ပြန်လည် ယူသုံးလိုက်ရုံပါပဲ၊ အသစ်ထပ်ပြီး ရေးသားနေစရာ မလိုတော့ပါဘူး။ ဒါကြောင့် ဒီအပိုင်းမှာ အပြင်ကနေ ဝင်လာမယ့် Request တွေကို Backend ဆီ စနစ်တကျ လွှဲပြောင်းပေးနိုင်ဖို့ Nginx ကို Reverse Proxy အနေနဲ့ အသုံးပြုဖို့သာ အာရုံစိုက်ပါမယ်။

Reverse Proxy ဆိုတာကို အနည်းငယ် ရှင်းပြရရင် အသုံးပြုသူတွေဆီက ဝင်လာတဲ့ Request တွေကို ကြားခံအနေနဲ့ စစ်ဆေးပေးပြီးမှ သက်ဆိုင်ရာ Backend တွေဆီကို လမ်းကြောင်းလွှဲ ပို့ဆောင်ပေးတဲ့ စနစ်တစ်ခု

ဖြစ်ပါတယ်။ ဒီလို ကြားခံ စနစ်တစ်ခုကို အသုံးပြုလိုက်တဲ့အတွက် လုံခြုံရေး ပိုကောင်းလာသလို၊ အလုပ်လုပ်တဲ့ နေရာမှာလည်း ပိုပြီး စနစ်ကျသွား စေပါတယ်။

Nginx ကို အသုံးပြုဖို့အတွက် ပထမဆုံး nginx ဆိုတဲ့ Folder အသစ်တစ်ခု တည်ဆောက်လိုက်ပါ။ အဲဒီ Folder ထဲမှာ nginx.conf ဆိုတဲ့ ဖိုင်တစ်ခုကို ဖန်တီးပြီး အောက်က Configuration တွေကို ရေးသားပေးရပါမယ်။

```
server {
    listen 80;

    location / {
        proxy_pass http://backend:5000;
        proxy_set_header Host $host;
        proxy_set_header X-Real-IP $remote_addr;
    }
}
```

ဒီ Configuration မှာ အရေးအကြီးဆုံးက proxy_pass http://backend:5000; ဆိုတဲ့ စာကြောင်းဖြစ်ပါတယ်။ ဒီနေရာမှာ Hostname အနေနဲ့ backend ဆိုတဲ့ နာမည်ကို သုံးထားတာကို သတိပြုပါ။ ဒါဟာ နောက်ပိုင်းမှာ ကျွန်တော်တို့ ရေးသားမယ့် Docker Compose ဖိုင်ထဲက Backend Service ရဲ့ အမည်ဖြစ်ပါတယ်။

ဆက်လက်ပြီး Nginx Container ကို အလုပ်လုပ်စေဖို့ nginx Folder ထဲမှာပဲ အောက်က Dockerfile ကို ထပ်မံ ဖန်တီးပေးပါ။

```
FROM nginx:alpine
RUN rm /etc/nginx/conf.d/default.conf
COPY nginx.conf /etc/nginx/conf.d/
```

ဒီ Dockerfile မှာဆိုရင် Base Image အနေနဲ့ အသုံးပြုထားပါတယ်။ အဲဒီနောက် Nginx ရဲ့ မူလ Default Configuration ကို ဖျက်ထုတ်လိုက်ပြီး ကျွန်တော်တို့ ရေးသားခဲ့တဲ့ nginx.conf ဖိုင်နဲ့ အစားထိုး ထည့်သွင်းလိုက်ပါတယ်။ ဒီအတွက်ကြောင့် Nginx Container စ Run တာနဲ့ ကျွန်တော်တို့ ညွှန်းကြားထားတဲ့အတိုင်း ဝင်လာသမျှ Request တွေကို Backend ဆီ တိုက်ရိုက် လွှဲပြောင်းပေးသွားမှာ ဖြစ်ပါတယ်။

အဆင့် (၃) Docker Compose ဖိုင် တည်ဆောက်ခြင်း

အခုဆိုရင် Backend အတွက် လိုအပ်တဲ့ကုဒ်တွေနဲ့ Nginx အတွက် Configuration တွေ အားလုံး အဆင်သင့် ဖြစ်နေပါပြီ။ ဒါကြောင့် ဒီအစိတ်အပိုင်း အားလုံးကို တစ်နေရာတည်းကနေ စုစည်း ထိန်းချုပ်မယ့် docker-compose.yml ဖိုင်ကို စတင် ရေးသားကြပါမယ်။ ပရောဂျက်ရဲ့

စာအုပ်အပြည့်အစုံကို ဝယ်ယူဖတ်ရှုနိုင်ပါတယ်။

အခန်း (၁၂) - Orchestration နိဒါန်းနှင့် Docker Swarm ဖြင့် စနစ်များကို ထိန်းချုပ်ခြင်း

ရှေ့ပိုင်း အခန်းတွေမှာ Container တစ်လုံးချင်းစီကို တည်ဆောက်တာ၊ Docker Compose ကို အသုံးပြုပြီး Service တွေ ချိတ်ဆက်တာတွေကို လက်တွေ့ လုပ်ဆောင်ခဲ့ကြပါတယ်။ ဒါ့အပြင် Configuration တွေနဲ့ Secret တွေကို လုံခြုံအောင် ဘယ်လို သိမ်းဆည်းရမလဲ၊ Container တွေရဲ့ အခြေအနေကို ဘယ်လို စောင့်ကြည့်ရမလဲ ဆိုတဲ့ အရေးကြီးတဲ့ အပိုင်းတွေကိုလည်း လေ့လာခဲ့ကြပါတယ်။

ဒါပေမဲ့ ဒီနေရာမှာ သတိပြုရမယ့် အချက်တစ်ချက် ရှိပါတယ်။ အခုချိန်အထိ ကျွန်တော်တို့ လေ့လာခဲ့သမျှ အရာအားလုံးဟာ Server တစ်လုံးတည်း (Single Machine) ပေါ်မှာပဲ အခြေခံပြီး လုပ်ဆောင်ခဲ့တာ ဖြစ်ပါတယ်။ တကယ့် လက်တွေ့ Production Environment ကြီးတွေမှာ Application တစ်ခုကို Server တစ်လုံးတည်းပေါ်မှာပဲ တင်ပြီး Run နေရုံနဲ့ လုံးဝ မလုံလောက်ပါဘူး။

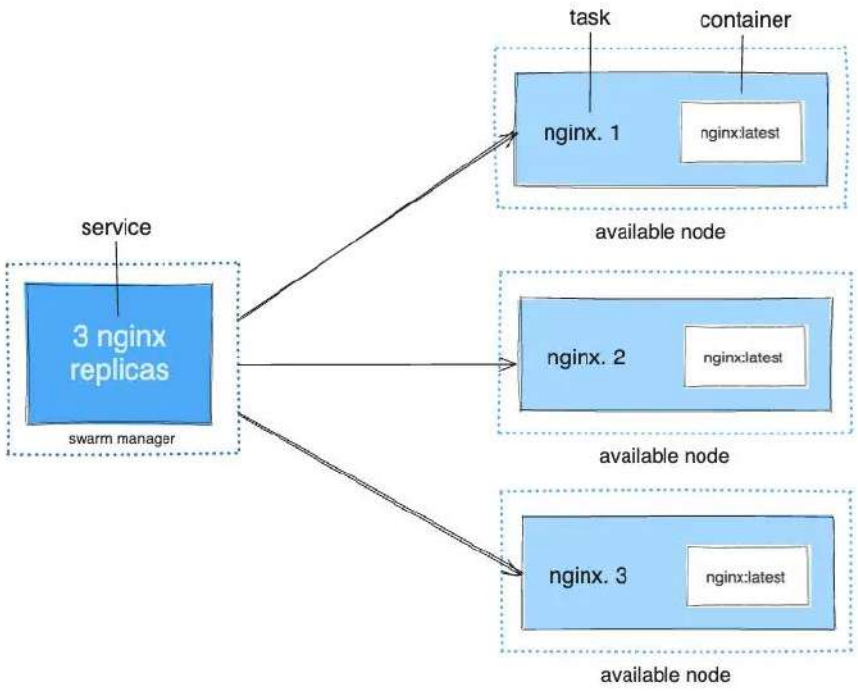
ဥပမာအားဖြင့် အသုံးပြုသူ သန်းနဲ့ချီပြီး တစ်ပြိုင်နက် ဝင်ရောက်လာမယ့် Application တစ်ခုရဲ့ Traffic ကို Server တစ်လုံးတည်းက ခံနိုင်ရည်ရှိဖို့ ဆိုတာ လုံးဝ မဖြစ်နိုင်ပါဘူး။ တကယ်လို့ Server တစ်ခုခုဖြစ်ပြီး Crash ဖြစ်သွားခဲ့ရင် Application တစ်ခုလုံး ရပ်တန့်သွားမယ့် အန္တရာယ်ရှိပါတယ်။ ဒီလို အခြေအနေမျိုးတွေကို ကျော်လွှားနိုင်ဖို့အတွက် စက်တစ်လုံးတည်း မဟုတ်ဘဲ စက်အများကြီးကို ပေါင်းစပ် အသုံးပြုဖို့ လိုအပ်လာပါတယ်။

စာအုပ်အပြည့်အစုံကို ဝယ်ယူဖတ်ရှုနိုင်ပါတယ်။

၁၂.၄ Docker Swarm မစတင်မီ သိထားသင့်သည့် အခြေခံ သဘောတရားများ

Docker Swarm ကို လက်တွေ့ စတင် အသုံးမပြုခင်မှာ သူ့ရဲ့ အခြေခံ အသုံးအနှုန်း Terminology တွေကို အရင်ဆုံး သေချာ နားလည်ထားဖို့ လိုအပ်ပါတယ်။ Swarm ကို "စက်အဖွဲ့အစည်း" တစ်ခုလို မြင်ကြည့်ရင် ပိုလွယ်ပါတယ်။ ဒီ Hierarchy ကို အရင်မှတ်ထားပေးပါ။

Stack → Service → Task → Container



စာအုပ်အပြည့်အစုံကို ဝယ်ယူဖတ်ရှုနိုင်ပါတယ်။

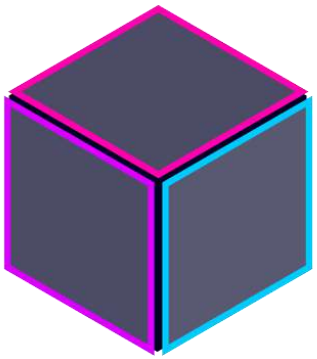
၁၂.၁၂ နိဂုံးချုပ်

ဒီအခန်း (၁၂) မှာဆိုရင် Container Orchestration ရဲ့ အရေးကြီးတဲ့ အခန်းကဏ္ဍကနေ စတင်ပြီး၊ Docker Swarm နဲ့ Cluster တစ်ခုကို လက်တွေ့ တည်ဆောက်ခဲ့ကြပါတယ်။ အဲဒီနောက်မှာ Service တွေ၊ Stack တွေကို ဘယ်လို စနစ်တကျ စီမံခန့်ခွဲရမလဲ ဆိုတာကို လေ့လာခဲ့ကြသလို၊ လက်တွေ့ လုပ်ငန်းခွင်မှာ အလွန် အသုံးဝင်တဲ့ Scaling နဲ့ Rolling Updates လုပ်ဆောင်ချက်တွေကိုပါ လေ့လာခဲ့ကြပါတယ်။ အခန်းရဲ့ နောက်ဆုံး အပိုင်းမှာတော့ လက်ရှိ ဈေးကွက်ထဲက နာမည်ကြီး Orchestration Tool တွေ ဖြစ်ကြတဲ့ Swarm နဲ့ Kubernetes တို့ရဲ့ အဓိက ကွာခြားချက်တွေကိုပါ လေ့လာခဲ့ကြပါတယ်။

ဒီစာအုပ်ရဲ့ အစပိုင်း Docker ရဲ့ အခြေခံ သဘောတရားတွေ၊ Container တွေ တည်ဆောက်တဲ့ အဆင့်တွေကနေ စတင်ခဲ့ပြီး အခုလို နောက်ဆုံး Container တွေကို စနစ်တကျ ထိန်းချုပ် စီမံတဲ့ Orchestration အဆင့်အထိ တစ်ဆင့်ချင်း စိတ်ရှည်လက်ရှည် လိုက်ပါ လေ့လာခဲ့တဲ့အတွက် စာဖတ်သူအနေနဲ့ Production အဆင့် Container Management ရဲ့ အခြေခံ သဘောတရားတွေကို ပိုင်ပိုင်နိုင်နိုင် နားလည် သဘောပေါက်သွားမယ်လို့ ကျွန်တော် အပြည့်အဝ ယုံကြည်ပါတယ်။ ဒီစာအုပ်ကနေ ရရှိလိုက်တဲ့ အခြေခံ သဘောတရားတွေဟာ နောင်တစ်ချိန်မှာ Kubernetes လိုမျိုး ပိုမို ကြီးမားတဲ့ အဆင့်မြင့် နည်းပညာတွေကို ဆက်လက် လေ့လာတဲ့အခါ သေချာပေါက် အထောက်အကူ ဖြစ်စေပါလိမ့်မယ်။

နည်းပညာ ဆိုတာ အမြဲတမ်း ပြောင်းလဲ တိုးတက်နေတာ ဖြစ်တဲ့အတွက် ဒီစာအုပ်ကနေ ရရှိလိုက်တဲ့ အသိပညာတွေပေါ်မှာပဲ ရပ်တန့်မနေဘဲ၊ ရှေ့ဆက် ပိုပြီး မြင့်မားတဲ့ အဆင့်တွေကို ဆက်လက် တက်လှမ်းနိုင်ပါစေလို့ ဆန္ဒပြုပါတယ်။ ဒီစာအုပ်ကို အစကနေ အဆုံးအထိ ယုံကြည်စွာနဲ့ လိုက်ပါ ဖတ်ရှု လေ့လာခဲ့ကြတဲ့ တစ်ဦးချင်းစီတိုင်းကို အထူးပဲ ကျေးဇူးတင် ရှိပါတယ်။

လေ့လာသူအားလုံး အဆင်ပြေကြပါစေ။



CODE

with Thura