AZƏRBAYCAN RESPUBLİKASI TƏHSİL NAZİRLİYİ

BAKI DÖVLƏT UNİVERSİTETİ

TƏTBİQİ RİYAZİYYAT VƏ KİBERNETİKA FAKÜLTƏSİ

RİYAZİ KİBERNETİKA

Ⅳ kurs KE-017S qrup tələbəsi

Yusifova Sevgi İlqar qızının

”Kompüter elmləri” ixtisası üzrə

bakalavr dərəcəsi almaq üçün

“CI/CD”

mövzusunda

BURAXILIŞ İŞİ

Kafedra müdiri: prof. Ə.Ə.Əliyev

Elmi rəhbər: E.E.Şükürov

Bakı – 2020

Mündəricat

Problem......................................................................................................................4

2Problemin həlli........................................................................................................5

2.1Proqram təminatında dəyişiklik olduqda versiyaların kontrolu sisteminin olmaması....................................................................................................................5

2.2Developerin production environment-ə qoşulma imkanının olması və

developerin üzərinə əlavə işin düşməsi.....................................................................6

“Continuous” (daimi) nə deməkdir?..........................................................................6

Continuous delivery pipeline nədir?..........................................................................7

Continuous delivery pipeline necə işləyir?................................................................8

"Fail fast" nədir?........................................................................................................8

Continuous Delivery Pipeline-ın bütün hissələri avtomatlaşdırılmalıdırmı? ...........9

Continuous Integration nədir?...................................................................................9

Continuous Integration necə işləyir?.........................................................................9

Continuous Integration dəyişiklikləri necə görür?....................................................9

“Pre-cheks” nədir?...................................................................................................10

“Unit test” nədir?....................................................................................................11

Continuous Testing nədir?.......................................................................................11

Pipeline-da testing-dən başqa hansı validasiyalar etmək olar?...............................12

Continuous delivery nədir?......................................................................................12

Pipeline-da versiyalanma necə baş verir?................................................................13

Artifaktlar necə “promote” olurlar?.........................................................................14

Müxtəlif versiya artefaktlar yaddaşda necə saxlanılır və onları necə ordan oxumaq olur...........................................................................................................................14

Continuous deployment nədir?................................................................................14

İstifadəçilərə çatdırılacaq tam deploying-dən qabaq deploymentin hansı test üsulları var?.............................................................................................................15

Blue/Green Testing/Deployments...........................................................................15

Canary testing/deployment......................................................................................16

Feature toggles.........................................................................................................16

Dark launch.............................................................................................................17

DevOps nədir?.........................................................................................................17

CD DevOps ilə necə kəsişir?...................................................................................18

“Pipeline-as-code” nədir?........................................................................................18

Proqram təminatının yaradılması infrastrukturuna DevOps necə təsir edir?..........18

Jenkins.....................................................................................................................19

Birdən çox addımın işlədilməsi...............................................................................22

Timeoutlar, təkrarlanmalar və s...............................................................................24

Tamamlanma...........................................................................................................26

Environment dəyişənlərinin istifadəsi.....................................................................29

Testlərin və artefaktların qeydi................................................................................30

Cleaning up və ismarıclar........................................................................................33

Deployment.............................................................................................................36

Addımlar Deployment Environmentlər kimi...........................................................37

Davam etmək üçün insan müdaxiləsinin tələb olunması........................................38

Nəticə.......................................................................................................................40

CI/CD

**Problem**:

Bildiyimiz kimi hər hansı produktun hazırlanması bir neçə etapdan ibarətdir. IT produktlar da istisna deyil. Bu etaplardan biri də “Development” prosesidir. Proqramçı produkt\* hazırlayarkən öz maşınında onu test edir və hazır olduğu təqdirdə onu “production” environment (“əhatə”) yerləşdirməlidir ki, istifadəçilərin ondan istifadə etmə imkanı olsun. Bunu etmək üçün bir neçə yol var və ən sadəsi production server-ə (host-a) build edilmiş proqramı yerləşdirib run etməkdir. Bu halda bizim proqram təminatına host olduğumuz server-də quraşdırılmış Apache, IIS və s. kimi serverlər qoşulma imkanı verəcək. Əlbəttə ki, developer production serverə müdaxilə etməməlidir. Bu düzgün praktika deyil. Bundan əlavə proqram təminatında dəyişikliklər baş verdikdə developer ya köhnə versiyanı silməli ya da backup götürməli olur. Bu da öz növbəsində serverdə artıq faylların yaranmasına və serverin fayl sisteminin artıq fayllarla dolmasına gətirib çıxarır. Hələ bu problemlərdən başqa developerin bu prosesə vaxt sərf etməsi də development prosesində gecikmələrə və iş şəraitinin pisləşməsinə gətirib çıxarır. Beləliklə bu prosesin 3 problemi var:

1. Proqram təminatında dəyişiklik olduqda versiyaların kontrolu sisteminin olmaması
2. Developerin production environment-ə qoşulma imkanının olması
3. Developerin üzərinə əlavə işin düşməsi

**Problemlərin həlli**

**Proqram təminatında dəyişiklik olduqda versiyaların kontrolu sisteminin olmaması**

Proqramda dəyişiklik olduqda köhnə kodun saxlanılması və lazım olduqda geri qaytarılması, development prosesinin branch-lara (budaq) ayrılması (master, test və s.) üçün versiyaların kontrolu sisteminə ehtiyac var və bu sistemlərdən ən populyarı git-dir. Bu sistem bizə hər hansı vaxt əvvəl edilmiş dəyişiklikləri geri qaytarmağa imkan verir. Bunun üçün developer proyekt-ə dəyişiklik etdiyi halda bu dəyişikliyi commit etməlidir. Bu commit-lər özündə yalnız edilən dəyişiklikləri saxlayır, onları git-in təqdim etdiyi xüsusi fayla yazır və proyektdə versiyalar əmələ gəlir. Developer-ə köhnə versiyaya qayıtmaq üçün terminalda sadəcə bir komanda yazmaq kifayət edir.

Bundan başqa git proyekti branch-lara ayırmağa da imkan verir. Bu nəyə lazımdır? Adətən production environment-dən əlavə test environment də olur ki, burada hələ production environment-ə yerləşdirilmək üçün hazır olmayan, test edilən yeniliklər yerləşdirilir. Testing prosesi üçün tester(lər) (QA) olur ki, onlar proqram təminatını production environment-lə identik test environment-də müxtəlif ssenarilərdə yoxlayır. Testdə edilən dəyişikliklər stabil olduğu və testerlər tərəfindən təsdiqləndiyi halda production environment-ə yerləşdirilir. Git sistemində dəyişikliklərin bir birinə qarışmaması üçün test branch-ı yaradılır və dəyişikliklərin testdə qəbul olunduğu halda test ilə master branch-ları merge olunur (birləşdirilir). Bu sistem təkcə test və master-dən yox, başqa spesifik branch-lardan da ibarət ola bilər. Misal üçün hansısa yeni bir funksiya (feature) yaradılırsa, onun üçün yeni branch yaradıla bilər. Bu branch-da development olduqdan sonra ilk növbədə test branch-ı ilə merge olunur, sonra dəyişikliklər tesdiqləndikdən sonra master branch-ı ilə merge olunur. Beləliklə git vasitəsi ilə versiyalanma problemini aradan qaldırmış oluruq.

**Developerin production environment-ə qoşulma imkanının olması və**

**developerin üzərinə əlavə işin düşməsi**

Bu iki problemin həlli üçün isə bizim mövzumuz olan CI/CD köməyimizə gəlir. CI/CD nədir? CI/CD (Continuous integration (CI) and continuous delivery (CD)) proqram təminatının istehsalı prosesi haqqında danışarkən çox ümumi işlənən termindir. Bəs əslində bu nə deməkdir?

Fabriklərdəki konveyerlər xammaldan hər hansı bir produkt düzətdiyi kimi proqram təminatının konveyeri də proqramçı tərəfindən yazılan kodu tez, avtomatlaşdırılmış və yenilənə bilən yol ilə release (hazır produkt) halına çatdırır. Bu proses “continuos delivery” (daimi çatdırılma) adlanır. Konveyeri başladan proses “continuous integration”, produktun keyfiyyətini yoxlayan hissə “continuous testing”, bu produktu istifadəçilər üçün əlçatan edən proses isə “continuous deployment” və bu işlə məşğul olan spesialist “DevOps” adlanır.

**“Continuous” (daimi) nə deməkdir?**

Continuous burada sadalanan proseslərə aiddir və bu heç də daim işlək demək deyil. Əslində bu həmişə işləməyə hazır deməkdir. Proqram tətbiqinin kontekstində bu həm də aşağıdakı praktikaları özündə cəmləşdirir:

* **Daim olan relizlər (reliz – hazır produktun təqdimi)**“Continuous” proseslərin məqsədi keyfiyyətli produktun qısa müddət intervallarında çatdırılmasıdır. Burdakı zaman intervalları produkt üzərində işləyən komandadan və ya kompaniyanın qoyduğu vaxt məhdudiyyətlərindən asılıdır. Bu müddət yarım il, 1 ay, 1 həftə, 1 gün və hətta daha az müddət ola bilər. Bəzən reliz günləri konkret vaxt intervallarından yox, ehtiyac baxımından tənzimlənə bilər. Ama məqsəd eynidir: Keyfiyyətli produktun istifadəçiyə tez, keyfiyyətli və təkrarlanabilən yolla çatdırılması. Bəzən bu proses heç istifadəçinin xəbəri olmadan belə baş verə bilər, misal üçün mobil tətbiqlərin yenilənməsi.
* **Prosesin avtomatlaşdırılması**Daim olan relizlərin bu qədər sürətli olması üçün ən əsas şərtlərdən biri produktun istehsalının demək olar bütün aspektlərinin emalının avtomatik olmasıdır. Bu aspektlərə produktun building (yığılması), testinqi, analizi, versiyalanması və bəzi hallarda “deployment”-i (serverə yerləşdirilməsi) daxildir.
* **Təkrarlanabilmə**Əgər bizim işlətdiyimiz avtomatlaşdırılmış proseslər daim eyni giriş parametrləri alıb və eyni nəticə verirsə, bu o deməkdir ki, bizim proses təkrarlanabiləndir. Yəni biz eyni versiyada olan proqram təminatını daxil etsək eyni nəticəni alırıq. Bu həm də o deməkdir ki, proqramın xarici mənbələrdən asılılığı olsa da nəticə eyni qalır. İdeal halda biz konveyerin hər hansı bir hissəsinə dəyişiklik etdikdə belə nəticə eyni qalmalıdır.
* **Sürətli emal**Sürətli əlbəttə ki nisbi anlayışdır, amma bu prinsip proqram kodunun dəyişildiyi andan relizə qədər olan prosesin effektiv və optimal olmasını nəzərdə tutur. Avtomatizasiya bu işi öz üzərinə götürur, amma avtomatlaşdırılmış proseslərin özü də gec emal oluna bilər. Misal üçün, produktun bütün aspektlərinin testinqi çox vaxt aparır, bəzən günün bir hissəsini təşkil edir, bu isə bir gün ərzində bir neçə dəfə yenilənən proqram təminatı üçün heç də yaxşı hal deyil.

**Continuous delivery pipeline nədir?**

Proqram kodunu işlədilmək üçün hazır olan hala gətirmək üçün istifadə olunan proseslər adətən bir konveyer şəklində düzülür, bu o deməkdir ki, bir proses işini bitirdikdən sonra o biri prosesə başlanğıc verir. Bu konveyerlərin müxtəkif adları var: continuous delivery pipeline, deployment pipeline və software development pipeline (proqram təminatının hazırlanması konveyeri). Pipeline-nın iş müddəti ərzində ümumi idarəedici proqram (supervisor application) pipeline-ın müxtəlif hissələrinin işləməsini, monitorinqini və menecmentini təmin edir.

**Continuous delivery pipeline necə işləyir?**

Proqram təminatının çatdırılması konveyerinin (software delivery pipeline) realizasiyası müxtəlif cür ola bilir. Burada çoxsaylı tətbiqlər var ki, hansılar konveyerin müxtəlif aspektlərində: kod trekinq, building, testing, göstərişlərin götürülməsi, versiyaların menecmentində və s. istifadə oluna bilər. Amma ümumi workflow əksər hallarda eynidir. Əsas proqram bütün workflow-u idarə edir və burada olan hər bir proses ayrıca run olur. Adətən hər bir proses orchestrator (idarəedici proqram) tərəfindən başa düşülən, idarə edilə bilən dildə və ya strukturda təyin olunur.

Proseslər bir və ya bir neçə funksiyanı (building, testing, deploying və s) icra etməyi nəzəri ilə qurulur. Hər bir proses müxtəlif və ya bir neçə texnologiyaları işlədə bilər. Əsas odur ki, hər bir proses işi avtomatlaşdırsın, effektiv və təkrarlanabilən olsun. Əgər hər bir proses öz işini düzgün yerinə yetirirsə prosesləri idarə edən tətbiq sonrakı prosesi başladır. Əks halda, proses öz işini düzgün yerinə yetirməsə orchestrator bu xəta haqqında developer, tester və digərlərinə məlumat verir ki, onlar bu nasazlığı aradan qaldırsınlar. Avtomatizasiya sayəsində bu tip xətalar daha tez tapılıb aradan qaldırılır nəinki bu proseslər manual olaraq işə salınsaydı. Xətalar haqqında olan bu tip bildirişlər “fail fast” adlanır və bizə uyğun nəticənin alınmasında mühüm rol oynayır.

**"Fail fast" nədir?**

Pipeline-ın gördüyü işlərdən biri dəyişikliklərin tez emalı, digəri isə relizi hazır edən proseslərin monitorinqidir. Yazılan kod compile olmur və ya testləri keçmirsə, istifadəçilər üçün vacibdir ki, bu xəta barədə mümkün qədər tez məlumat çatdırılsın. Fail fast pipeline-da olan problemin mümkün qədər tez tapılıb, bu məsələdə kompetent olan şəxslər tərəfindən “fix” olunub pipeline-nın yenidən start verilməsi ideyasına əsaslanır. Adətən, pipeline prosesi dəyişikliklərin kim tərəfindən edildiyini bilir və həmin şəxsə və onun komandasına bu xəta haqqında xəbər verir.

**Continuous Delivery Pipeline-ın bütün hissələri avtomatlaşdırılmalıdırmı?**

Demək olar ki bütün proseslər avtomatlaşdırılmalıdır. Amma, bəzi hallarda insanın prosesə müdaxilə etməsi lazım ola bilər. Buna misal olaraq istifadəçilərin məmnunluq testi ola bilər (İstifadəçilərin produktdan olan gözləntilərinin ödənilməsi üçün). Başqa bir misal proqram təminatının production environment-ə qoyulduğu zaman insan kontrolunun arzuolunan olduğu hallardır. Və əlbəttə ki, kodun düz işləmədiyi halda insan müdaxiləsi tələb olunur.

Pipeline haqqında ümumi anlayışları aldıqdan sonra müxtəlif növ continuous processing-ə və onların təhlilinə başlamaq olar.

**Continuous Integration nədir?**

Produkt kodunun dəyişdiyi halda dəyişikliyin müəyyən olunması, kodun mənbədən yüklənməsi (pulling), building və əksər hallarda Unit testlərin keçirilməsi prosesinə Continuous Integration deyilir. CI – Pipeline-ı başladan prosesdir (Bəzən pre-validations – bəzən uçuşqabağı yoxlanış (pre-flight checks) adlanan proseslər CI-dan öncə başladılır).

CI-ın məqsədi developerin yazdığı kodun işlək olub olmamasını və gələcək versiyalarda kod bazası kimi işlənə bilinməsini yoxlamaqdır.

**Continuous Integration necə işləyir?**

Əsas ideya bir və ya bir neçə kod repository-lərinin güdən avtomatlaşdırılmış “watching” prosesidir. Kod dəyişildiyi və repository-ə push olunduğu zaman “watching” prosesi yeni kodu yükləyir, build edir və uyğun unit testləri run edir.

**Continuous Integration dəyişiklikləri necə görür?**

Bugünki gündə “watching” prosesini adətən Jenkins kimi tətbiq üzərinə götürür, hansı ki həmçinin pipeline-da olan bütün (və ya əksər) prosesləri orchestrasiya edir və hər bir prosesdə olan dəyişiklikləri görə bilir. “Watching” prosesi dəyişiklikləri bir neçə yolla təxmin edə bilir:

* **Polling:** Monitorinq proqramı periodik olaraq repository-ə müraciət edir və yeni dəyişikliyin olub olmadığını yoxlayır. Əgər repository-də hər hansı dəyişikliklər varsa, monitorinq proqramı “oyanır” kodu pull, build, test edir.
* **Periodic:** Monitorinq proqramı dəyişikliklərin olub olmamasından asılı olmuyaraq periodik olaraq kodu build edir. İdealda dəyişiklik yoxdursa əlavə heçnə build olmur və uyğun olaraq əlavə resurs işlənmir.
* **Push:** Bu yol əksinə işləyir. Bu halda source management system kodda dəyişiklik olub commit olunduğu zaman monitoring tətbiqinə “push notification” atır. Əksər hallarda bu yanaşma “Web hook”-lar vasitəsi ilə realizasiya olunur, harada ki, xüsusi proqram kodun push olunduğunu hook vasitəsi ilə tutur və internet vasitəsi ilə monitoring tətbiqinə bildiriş göndərir. Bunun işləməsi üçün monitoring tətbiqinin açıq portu olmalıdır ki, hansına webhook vasitəsilə müraciət olunur.

**“Pre-cheks” nədir?**

Kodun əlavə validasiyalarıdır ki, kodun push olub pipeline-ı start verməyindən öncə baş verə bilir. Bu test build-lər və code-review kimi best-practice-ləri özündə cəmləşdirir. Bu proses adətən kodun pipeline-a daxil olmasından öncə baş verir. Amma bəzi pipeline-lar bu prosesi özündə realizasiya edə bilər. Misal kimi Gerrit kimi tool-u göstərmək olar ki, hansı kodun push olunub, amma repository tərəfindən hələ qəbul olunmadığı zaman ərzində işə düşür və formal code-review, validasiylar və test build-lər edir. Gerrit developerin iş yeri və Git remote repository arasında yerləşir. O, push olunan kodu tutur, validasiyaları edir, pass/fail cavabları vasitəsi ilə kodun repository tərəfindən qəbul edilib edilməyəcəyini təyin edir. Bu prosesə həm də kodun dəyişikliklərinin müəyyən olunması və test build-lərin hazırlanması da daxil ola bilər. Bu həm də hər hansı qrupların formal review etməsinə imkan verir. Bu yolla biz əmin ola bilərik ki, yeni yaranmış dəyişikliklər heçbir çətinliklər yaratmıyacağ

**“Unit test” nədir?**

Unit testlər developerlər tərəfindən yazılmış kiçik testlərdir ki, dəyişikliklərin avtonom işləmək qabiliyyətini yoxlayır. Avtonom dedikdə burada kodun başqa kodlarla əlaqədə olmadığı, başqa modulların ona təsiri olmadığı, heçbir asılılığı olmadığı vəziyyət deməkdir. Əgər kodun işləməsinə hansısa məlumat bazası və ya hansısa asılılıq lazımdırsa, onlar mock lar vasitəsi ilə əvəz olunur. Mock dedikdə resursu imitasiya edib müraciət vaxtı cavab verən, amma heçbir funksionalı olmuyan kod imitasiyasına deyilir.

Əksər orqanizasiyalarda developerlər kodla yanaşı öz kodlarına test də yazmalıdırlar ki kodun işləkliyini təsdiqləyə bilsinlər. Faktiki olaraq bir model (Test driver development kimi tanınan(TDD)) kodun yazılmasından öncə testlərin yazılmasını tələb edir ki, kod yazılanda kodun nə iş görəcəyi tam aydın olsun. Çünki belə kod dəyişikliklərinin sütətli və çoxsaylı olduğu kimi, onların yerinə yetirilmə sürəti də yüksək olmalıdır.

Testlər CI ilə sıx əlaqədə olduğuna görə developer öz local maşınında bu testləri run edib kodun işləkliyini yoxluya bilər. Adətən bu testlər ifadə formasını alır ki, funksiyaya verilən bir sıra giriş parametrləri, uyğun çıxış nəticələri verməlidir. Bir qayda olaraq onlar xəta emalının düzgünlüyünü yoxlayır. Bir sıra unit-testing framework-lar developerlərə bu işdə kömək edir.

**Continuous Testing nədir?**

Continuous testing CD pipeline-da kodun geniş spektrda testlərinin avtomatik start olunması prinsipinə əsaslanır. Unit testlər adətən CI-ın bir hissəsi olaraq inteqrasiya olunur və kodun avtonom vəziyyətdə işləmə qabilliyətini yoxlayır.

Bundan başqa testing-in başqa formaları da var və bu formalara aşağıdakıları misal göstərmək olar:

* **Integration testing** bütün servis və komponentlərin birlikdə işləmə qabiliyyətini yoxlayır.
* **Functional testing** funksiyaları onlardan olan gözləntilərə uyğun yoxlayır.
* **Acceptence testing** sistemi müəyyən parametrlərinə görə yoxlayır. Bunlara misal sistemin iş tezliyi, genişlənmə imkanı, həcmi və s. göstərmək olar.

Bunlardan bəziləri, yaxud heçbiri avtomatlaşdırılmış pipeline-da olmuya bilər və bu anlayışlar arasında sərhədlər qeyri dəqiq realizasiya oluna bilər. Amma, buna baxmıyaraq continuous testing-in məqsədi eynidir: ardıcıl testing-lər vasitəsi ilə produktun keyfiyyətini və reliz üçün hazır olduğunu təsdiqləmək. Continuous prinsipinin sürətinə əsaslanaraq ikinci məqsəd isə problemlər haqqında komandaya tez bir şəkildə xəbər verməkdir. Bunu fail fast adlandırırlar.

**Pipeline-da testing-dən başqa hansı validasiyalar etmək olar?**

Testlərin pass/fail nəticələrindən əlavə tətbiqlər var ki, hansılar testlər tərəfindən neçə sətr kodun yerinə yetirildiyini bizə deyə bilər. Bu koda əsasən götürülə bilən metrikalara aiddir. Bu metrika code-coverage adlanır və xüsusi tool-lar (Java üçün JaCoCo) vasitəsi ilə ölçülə bilər.

Bundan başqa bir çox növ metrikalar da var ki, hansılar kodun sətrlərinin sayını, çətinliyini, müəyyən olunmuş patternlərə uyğunluğunu yoxlayır. SonarQube kimi tool-lar kodu analiz edib və onun metrikalarını götürə bilər. Bundan əlavə istifadəçilər bu metrikalara qəbul oluna biləcək sərhədlər qoya bilərlər. Sonradan pipeline-ı elə tənzimləmək olar ki, hesablanan qiymətlər sərhədlərlə müqayisə olunsun və qiymətlət qəbulolunmaz olduqda prosesin emalı dayandırılsın. SonarQube çox yaxşı tənzimlənilə bilir və bu da imkan verir ki, validasiyaları komandaya uyğun qurulsun.

**Continuous delivery nədir?**

Continuous Delivery (CD) bir qayda olaraq bütün proseslər sırasına aiddir ki, hansı ki, avtomatik kod dəyişikləri haqqında məlumat alır və kodu building, testing, packaging və bu kimi proseslərdən keçirib, insan müdaxiləsi olmadan reliz üçün hazır olan produkt hazırlayır.

CD-nin proqram təminatının hazırlanması prosesindəki məqsədi avtomatizasiya, effektivlik, etibarlılıq, təkrarlanabilmə və keyfiyyətin yoxlanılmasının (continuous testing vasitəsi ilə) təminatıdır.

CI (kodun dəyişməsini güdülməsi, dəyişikliklərin building prosesini yerinə yetirilməsi və validasiya üçün unit testləri run edilməsi), continuous testing (kodun keyfiyyətinin yoxlanılması üçün müxtəlif növ testlərin ardıcıllıqla yerinə yetirilməsi) və continuous deployment (reliz versiya proqram təminatına istifadəçilər üçün qoşulma imkanının yaradılması) CD-ə daxildir.

**Pipeline-da versiyalanma necə baş verir?**

Versiyalanma CD pipeline-la iş zamanı əsas aspektlərdəndir. Continuous dedikdə daim yenilənən kod bazası və relizlərin olması nəzərdə tutulur. Amma bu o demək deyil ki, hər dəfə kod yeniləndikdə bu kod ən son və ən yaxşı koddur. Bu əsasən axırıncı stabil versiyanı development və ya build etmək istəyən daxili qruplar üçün aktual ola bilər. Buna görə də vacibdir ki, pipeline versiyalar yaratsın və onları rahatlıqla yaddaşda saxlayıb onlara müraciət edə bilsin.

Pipeline-ın iş vaxtı kod əsasında yaranan bütün obyektlərə artifact (artefakt) deyilir. Artefaktlar build olunduğu zaman onlara versiya verilməlidir. Rekomendasiya olunan versiyalanma sxemi “Semantik versiyalanma”-dır. (Bu həm də xaricdən gələn asılılıq artefaktlarının versiyalarına da aiddir.)

Semantik versiya nömrələri 3 hissədən ibarətdir: major, minor və patch. (1.4.3 versiya nömrəsi üçün 1 major, 4 minor, 3 isə patch versiyaları ilə uyğunlaşır). İdeya ondadır ki, bu versiya nömrəsində olan dəyişiklik, koddakı dəyişikliyin səviyyəsini göstərir. Major versiyanın dəyişməsi köhnə versiya ilə uyğunlaşa bilməyən kodla bağlıdır. Minor versiyadakı dəyişiklik yalnız keçmiş versiya ilə uyğunlaşa bilən funksional dəyişikliklərinə, Patch versiyasındakı dəyişiklik isə keçmiş versiya ilə uyğunlaşa bilən bug-fix (xətaların aradan qaldırılması) lərə aiddir. Bu rekomendasiya olunan yanaşmadır, əlbəttəki hər bir kompaniya öz versiyalanma sxemini işlədə bilər, əsas odur ki, hamı bu sxemi başa düşüb eyni qayda ilə onu istifadə etsin. Misal üçün hər dəfə build uğurlu keçəndə və reliz versiya alınanda bir vahid artırılan versiya path ola bilər.

**Artifaktlar necə “promote” olurlar?**

Komandalar artefatın testing, production və s. uyğun olub olmadığını yoxlamaq üçün onlara promotion level verə bilir. Burada müxtəlif yanaşmalar var. Jenkins və Artifactory kimi tətbiqlər promotionlar üçün işlənə bilər. Və ya daha sadə yol, versiyanın sonuna uyğun postfix əlavə etmək olar. Misal üçün, -snapshot kodun axırıncı versiyasını (snapshot) qeyd edir. Müxtəlif promotion strategiyaları və ya tool-ları artefaktı digər səviyyələrə (-milestone, -production) promote edə və onların stabillik və hazırlıq səviyyələrini bunla qeyd edə bilər.

**Müxtəlif versiya artefaktlar yaddaşda necə saxlanılır və onları necə ordan oxumaq olur**

Versiyalanmış artefaktlardan source koddan düzəlir və “artifact repositories” idarə edən proqram təminatları vasitəsilə yaddaşa yazılır. Artefakt repozitoriləri source management sistemləri kimidir, amma burada obyekt artefaktdır. Artifactory və ya Nexus kimi proqram təminatı artifaktların alınmasına, xaric olunmasına, onların menecmentinə və s. vasitəçilik edir.

Pipeline istifadəçiləri hansı versiya artefaktı istifadə etmək istədiyini göstərərək pipeline-ı həmin artefaktla işə sala bilər.

**Continuous deployment nədir?**

Continuous deployment (CD) hazır reliz versiyanı CD pipeline-dan götürüb istifadəçilər üçün produktu əlçatan edir. İnstallment-in keçirilməsi yolundan asılı olaraq CD produktu cloud-a deploy edə, proqram təminatının yeniliklərini istifadəçilərə təqdim edə (əsasən mobil tətbiqlərində), səhifəni yeniləyə və ya sadəcə mümkün yeniliklər siyahısının artıra bilər.

CD-nin pipeline-dan çıxan reliz versiyanı deploy etməsi heç də pipeline-dan çıxan bütün nəticələrin deploy olması demək deyil. Bu o deməkdir ki, pipeline-dan çıxan nəticələr “deployable” statusu alır. Bu əsasən ardıcıl pilləli continuous testing vasitəsi ilə nail olunur.

Pipeline-dan çıxan nəticənin deploy olub olmamasında asılı olmuyaraq tam deployment-dən qabaq insan və ya testing metodları tərəfindən son yoxlanışlar keçirilə bilər.

**İstifadəçilərə çatdırılacaq tam deploying-dən qabaq deploymentin hansı test üsulları var?**

End-user-lərin istifadəsi üçün deploy olunmuş proqram təminatının geri qaytarılması neqativ hadisə olduğundan (Həm texniki cəhətdən, həm də istifadəçilərin bu hadisəni qarşılaması cəhətindən) çoxsaylı metodologiyalar ixtira olunub ki, hansılar edilən dəyişiklikləri “yoxlamağa” və neqativ hallar yarandığı halda dəyişikliklərin tez və rahat şəkildə geri qaytarılmasına imkan verir. Bu metodologiyalara daxildir:

**Blue/Green Testing/Deployments**

Proqram təminatının deploymentinə olan bu yanaşmada iki identik environment olur – blue və green (Burdakı rənglər heçbir məna daşımır və yalnız identifikasiya üçün istifadə olunurlar). İstənilən halda bu iki environmentdən biri *production* deploymentini, digəri isə *candidate* deploymentini özündə saxlayır.

Bu iki instansdan öncə “gateway” dayanır və klient üçün produkta və ya tətbiqə olan şlüz rolunu oynayır. Marşrutizatoru blue və ya green environment-ə yönəltməklə istifadəçi trafiki seçilmiş deployment-ə yönləndirilə bilər. Bu halda deployment instance tez, asan və istifadəçi üçün şəffaf şəkildə dəyişdirilə bilər.

Yeni reliz testing üçün hazır olanda, deployment non-production environmentə yerləşdirilə bilər. Bu deployment test olunub təsdiqləndikdən sonra əsas trafik artıq bu environmentə yönləndirilə bilər (Həmin environment artıq production environment sayılır). İndi isə bundan öncə production environment olan hosting yeni candidate üçün hazır sayılır.

Həmçinin əgər sonuncu deployment-də problem tapılarsa və köhnə versiya hələ də digər environment-də qalıbsa, sadəcə istifadəçilərin trafiki köhnə production environment-ə yönləndirilir. Bununla effektiv şəkildə keçmiş production versiyasına qayıdılmış olur. Problemli yeni deployment daha sonra başqa environment-də review oluna və olan xətalar düzəldilə bilər.

**Canary testing/deployment**

Bəzi hallarda relizin blue/green prinsipi ile deploymenti işlək və ya arzuolunan olmuya bilər. Başqa bir yanaşma *canary* testing/deployment kimi tanınır. Bu modeldə istifadəçi trafikinin bir hissəsi produktun yenilənmiş hissələrinə yönləndirilir. Misal üçün, saytdakı axtarışın yeni versiyası production deploymentlə birlikdə deploy olunur. Və 10% axtarış müraciətləri yenilənmiş axtarış servisinə production environment-də testing etmək üçün yönləndirilir.

Əgər kiçik miqdarda trafiki problemsiz emal edirsə, bu servisə daha çox trafik yönləndirilir. Əgər trafikin miqdarının artımı ilə heç bir problem yanamırsa, yönləndirilən istifadəçi trafikinin miqdarı 100%-ə kimi artırıla bilər. Bu köhnə versiya deployment-i effektiv şəkildə “silir” və onun yerinə userlərin istifadəsi üçün yeni versiyanı gətirir.

**Feature toggles**

Asanlıqla geri qaytarıla biləcək (problem çıxdığı halda) yeni funksionallığın yeridilməsi üçün yeridilməsi üçün developerlər *feature toggle* əlavə edə bilərlər. Bu proqram kodunda olan if-then keçididir ki, uyğun qiymətin olduğu halda lazım olan kod parçasını işə salır. Danışılan qiymət kodun bu qiyməti tapıb ala biləcəyi yerdə yerləşməlidir ki, production-da olan kod, həmin qiymətə əsasən yeni funksionalı işlədə bilsin. Bu qiymət təyin olunubsa, yenilik işlənilir, əks halda isə production öz stabil funksionallığını davam etdirir.

Bu yanaşma developerlərə yeni funksionallığda problem çıxdığı halda həmin funksionallığı “kill switch” etməyə imkan verir.

**Dark launch**

Bu praktikada yeni funksionallığın kodu hissə-hissə production-da yoxlanılır, amma istifadəçi olunan dəyişiklikləri görmür. Misal üçün, production relizdə bir hissə müraciətlər yeni funksionallıq olan servisə yönləndirilir. İstifadəçilərə interfeys, tranzaksiyalar və cavablar qaytarmadan bütün informasiya developerlərin analizi üçün yığılır.

Burada ideya yeni funksionallığı production environmentin yükü altında candidate deploymentin yoxlanılmasıdır. Vaxt keçdikcə tətbiqin üzərinə düşən trafiki problem çıxana kimi və ya production-a tam hazır olana kimi artırmaq olar.

**DevOps nədir?**

DevOps development və operational komandaların development və releasing işini asanlaşdırmağa xidmət göstərən ideyalar və rekomendasiya olunan praktikalar toplusudur. Tarixən bütün development komandaları produktu yaradıb, amma onları reqular, təkrarlanabilən yolla install/deploy etmirdilər. İnstall/deploy öhdəlikləri (həmçinin digər support işləri) dövrün axırında operation team-in üzərinə düşürdü. Bu adətən böyük qarışıqlıqlara və problemlərə gətirib çıxarırdı. Çünki, operation team dövrün axırında başqa bir dövrə qatılırdılar və qısa müddətdə üzərinə düşən öhdəliyi yerinə yetirməli idilər. Bundan başqa developerlər komandası pis vəziyyətdə qalırdı, çünki produktu/deployment-i yetərincə test etmirdilər və proqram təminatında problem çıxdıqda bu gözlənilməz olurdu.

Adətən belə hallar developer və operation team-lər arasında əlaqəni pozurdu. DevOps idealları işlərin həm developer, həm də operations team-lərinin iştirakı ilə pipeline-ın əvvəlindən sonuna kimi birlikdə görülməsi ideyasını irəli sürür.

**CD DevOps ilə necə kəsişir?**

CD bir sıra DevOps ideallarının implementasiyasıdır. Produktun daha sonrakı packaging və deployment etapları pipeline-ın hər dövründə baş verə bilir və heç bir product development dövrünü gözləmir. Həmçinin development-dən developmentə development və operation team-lər şəffaf şəkildə produktun işləyib və ya işləmədiyini görə bilir. Developmentin uğurlu olması üçün CD pipeline nəinki development ilə bağlı olan hissəni keçməli, həm də operations ilə bağlı hissəni də uğurlu olaraq keçməlidir.

Növbəti səviyyəyə qalxdıqda DevOps pipeline-ı implementasiya edən infrastrukturun da kod kimi baxılmasını rekomendasiya edir. Yəni o avtomatik olaraq hazırlanmalı, güdülməli, asanlıqla dəyişilməli və pipeline dəyişildikdə yeni pipeline-ın işə başlamasını təmin etməlidir. Buna pipeline-ın kod kimi realizasiya olunması ilə nail olunur.

**“Pipeline-as-code” nədir?**

*Pipeline-as-code* developerlərin proqram kodunu yaratdığı kimi pipeline job/task-ların kod vasitəsi ilə yaradılmasına aid ümumi termindir. Məqsəd pipeline implementasiyasının kod kimi digər kodlarla birgə saxlanılmasıdır ki, bu da review, track, asanlıqla restart, problemli hallarda pipeline-ı saxlamaq kimi imkanlar açır. Jenkins daxil olmaqla bir sıra tool-lar buna imkan verir.

**Proqram təminatının yaradılması infrastrukturuna DevOps necə təsir edir?**

Adətən pipeline-da istifadə olunan individual hardware sistemlər proqram təminatı (OS, tətbiqlər, development tools və s.) vasitəsi ilə konfiqurasiya olunmuş olurdu. Bəzən hər sistem custom olub və hər birinin öz custom setup-ı var idi. Bu da o demək idi ki, burada hər hansı problem olduqda və problemin həlli tələb olunduqda yeni tasklar custom tasklar olurdu. Bu yanaşma fundamental CD prinsiplərinə ziddir, hansı ki, sadə yenilənəbilən və güdüləbilən environment tələb edir.

Keçən illər ərzində installing və configuring proseslərini standartlaşdıran tətbiqlər yaradıldı. Bundan başqa Virtual maşınlar yaradıldı ki, hansılar proqram vasitəsi ilə bir əməliyyat sisteminin üzərində digər əməliyyat sisteminin işini emulasiya edir. Bu Virtual Maşınlara (VM) supervisory program lazımdır ki, host sistemin üzərindən qalxsınlar. Və onlara öz əməliyyat sistemləri lazımdır ki, işləsinlər.

Növbəti gələn container-lərdir. Container-lərin prinsipi eyni olsa da fərqli işləyirlər. Onların ayrı tətbiq və əməliyyat sisteminin kopyasını tələb etməkdənsə sadəcə olan əməliyyat sistemində olan konstruksiyalardan istifadə edib əməliyyat sistemində özlərinə izolasiya olunmuş yer yaradırlar. Beləliklə onlar özlərini Virtual maşın kimi aparır ama əlavə resurs tələb etmirlər.

Virtual maşınlar və container-lər hazır parametrlər əsasında yarandığından işlədiyi host sistemə zərəri olmadan silib yenisini yaratmaq olur. Bu yenidən yaradıla bilən sistemə pipeline-ı başlatmağa imkan verir. Bundan başqa biz container-lərin yarandığı fayllardan yeni container-lər yarada bilirik, source kod-dakı kimi.

Bu yolla, əgər biz virtual maşında və ya container-də hər hansı bir problemlə qarşılaşırıqsa onu debug edib problemi axtarmaqdansa, bəzən container-i məhv edib yenisini yaratmaq daha tez və asan olur.

Bu həm də o deməkdir ki, source koddakı kimi pipeline üçün olan kodda hər hansı dəyişiklik olduqda pipeline yenidən işə başlıya bilər (CI vasitəsilə). Bu DevOps-un infrastruktur baxımından əsas ideologiyalarından biridir.

**Jenkins**

Jenkins-in işləməsi üçün qoyulan minimal tələblər:

* Minimum 256 Mb RAM, tövsiyyə olunan həcm – 512 Mb
* Yaddaşda Jenkins və Docker üçün 10Gb boş yer
* Java 8 və ya 11
* Docker

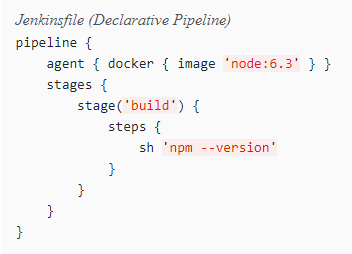
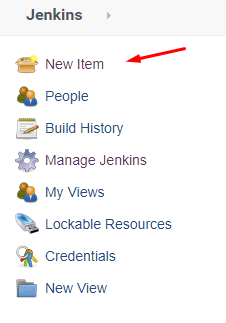
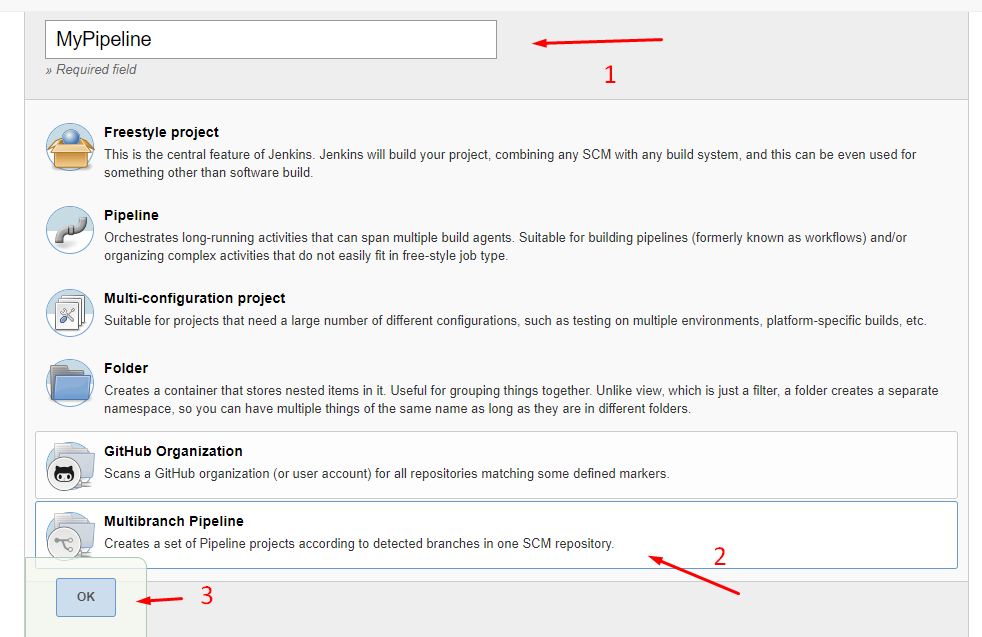
Jenkinsin environment-ə quraşdırılması:

* Jenkins-i yükləyirik
* Yüklədiyimiz qovluqda terminalı açırıq
* “java –jar jenkins.war --httpPort=8080” komandasını yerinə yetiririk
* <http://localhost:8080> adresində olan instruksiyanı yerinə yetiririk

Jenkins Pipeline (və ya sadəcə “Pipeline”) Jenkins dəki continuous delivery pipelines-ı dəstəkləyən plugin-lər yığımıdır.

Continuous delivery pipeline avtomatlaşdırılmış ifadədir ki, hansı ki, sizin source version control sisteminizdən produktu son istifadəçiyə çatdırır. Jenkins bizə genişləndiriləbilən tool komplekti verir ki, onun vasitəsilə pipeline yarada bilirik. Jenkins haqqında informasiya adətən JenkinsFile adlanan faylda qeyd olunur və öz növbəsində repozitoridə yerləşdirilir.

Pipeline ilə işə başlayaq:

1. JenkinsFile yaradırıq, bunun üçün isə repositoridə növbəti nümunəni kopyalayıb yeni fayl yaradıb, adını JenkinsFile qoymaq lazımdır.  
   
2. Jenkins-də “New Item” menüsuna klikləyirik  
   
3. Pipeline-ı adlandırıb MultiBranchPipeline seçirik  
   
4. . Add Source düyməsini klikləyirik. İstifadə etməyi istədiyimiz repository tipini seçirik və detalları qeyd edirik.

5. Save düyməsini klikləyək və ilk pipeline’ın işləməyini izləyirik!

Bizim proyektimizlə uyğunlaşmağı üçün nümunə Jenkinsfile’lardan birini dəyişdirmək lazım ola bilər. Sh komandasını öz local maşınınızda işlətdiyiniz eyni komanda ilə əvəz etməyi sınayın.

Pipeline’ı quraşırdırdıqdan sonra Jenkins avtomatik olaraq bütün yeni Branch’ları və Pull Request’ləri aşkar edəcək, bunlar sizin repository’də yaradılacaq və pipeline’ları onlar üçün işlədəcək.

**Birdən çox addımın işlədilməsi**

Proqram təminatını qurmaq, test etmək və tətbiq etmək üçün pipeline’lar birdən çox addımdan ibarətdir. Jenkins pipeline bizə birdən çox addımı asan bir şəkildə tətbiq etməyə icazə verir ki, bu da istənilən tip avtomatik prosesi modelləşdirməyimizə kömək edir.

Bir “addımı” bir əmri yerinə yetirən bir komanda olaraq təsəvvür etsək, hər bir addım uğurlu şəkildə yerinə yetirildikdən sonra növbəti addıma keçir. Əgər hansısa addım uğursuz olarsa, pipeline da fail olar.

Bütün addımlar uğurla tamamlandıqda, belə qəbul olunur ki, pipeline işini uğurla bitirmişdir.

**Linux, BSD, və Mac OS**

Linux, BSD və Mac OS (Unix tipli) sistemlərdə sh addımı Pipeline’da shell komandalarının yerinə yetirilməsi üçün istifadə olunur.

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Build') {

steps {

sh 'echo "Hello World"'

sh '''

echo "Multiline shell steps works too"

ls -lah

'''

}

}

}

}

**Windows**

Windows əsaslı sistemlər batch komandalarını yerinə yetirmək üçün bat addımlarından istifadə edir.

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Build') {

steps {

bat 'set'

}

}

}

}

**Timeoutlar, təkrarlanmalar və s.**

Burada bir sıra başqa addımları özündə saxlayan güclü addımlar vardır ki, həmin addımlar problemləri asanlıqla həll edir(həll olunana kimi addımlar yenidən təkrarlanır) yaxud da əgər addım çox uzun zaman alırsa çıxış edir.

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Deploy') {

steps {

retry(3) {

sh '. /flakey-deploy. sh'

}

timeout(time: 3, unit: 'MINUTES') {

sh '. /health-check. sh'

}

}

}

}

}

“Deploy” addımı flakey-deploy. sh skriptini 3 dəfə təkrarlayır və sonra 3 dəqiqə health-check. sh skriptinin yerinə yetirilməsini gözləyir. Əgər 3 dəqiqə içində health-check scripti işini bitirməsə onda pipeline “deploy” addımında fail marker ilə qeyd olunacaq.

Timeout və retry kimi wrapper addımlar başqa addımları özündə saxlaya bilir, həmçinin timeout ve retry da bunlara daxildir.

Biz bütün bu addımlar bir yerdə yarada bilərik. Məsələn əgər biz deployment prosesini 5 dəfə təkrarlamaq istəyiriksə ve fail olmaqdan qabaq deploymentə 3 dəqiqədən artıq vaxt sərf etmək istəmiriksə bu konfiqurasiyanı veririk.

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Deploy') {

steps {

timeout(time: 3, unit: 'MINUTES') {

retry(5) {

sh '. /flakey-deploy. sh'

}

}

}

}

}

}

**Tamamlanma**

Pipeline işini bitirdikdən sonra sizə hər hansı təmizlik addımları və ya pipeline’ın çıxış parametrlərinə əsaslanan əməliyyatlar etmək lazım ola bilər. Bu əməliyyatlar post bölməsində qeyd olunur.

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Test') {

steps {

sh 'echo "Fail!"; exit 1'

}

}

}

post {

always {

echo 'This will always run'

}

success {

echo 'This will run only if successful'

}

failure {

echo 'This will run only if failed'

}

unstable {

echo 'This will run only if the run was marked as unstable'

}

changed {

echo 'This will run only if the state of the Pipeline has changed'

echo 'For example, if the Pipeline was previously failing but is now successful'

}

}

}

Qabaqkı bölümlərdə agent direktivinin hər nümunədə qeyd olunduğuna fikir verə bilərsiniz. Agent direktivi jenkins-ə pipeline-ın hansı environmentdə və necə işləyəcəyini göstərir. Gözləndiyi kimi, agent bütün pipeline-lar üçün mütləqdir.

Agent bəzi proseslərin baş verməsinə səbəb olur:

* Jenkinsin blokda olan bütün addımları yerinə yetirməsi üçün onlar sıraya düzülür və “executor” hazır olan kimi bütün addımlar yerinə yetirilməyə başlayır.
* Versiya kontrol sistemindən götürülmüş fayllar və pipeline’ın əlavə fayllarının saxlanılması üçün workspace ayrılır.

Pipeline’ın agentlərinin bir neçə təyin olunma yolu var. Biz bu yanaşma ərzində docker containerlərə baxacağıq.

Pipeline docker imagelerinin ve konteynerlerin asanlıqla işlədilməsi ideası ilə yaradılıb. Bu pipeline-a müxtəlif sistem instrumentlərini və asılılıqlarını konfiqurasiya etmədən lazım olan environment və tool-ları təyin etməyə imkan verir.

Bu yanaşma bizə Docker containerində paketlənə bilən istənilən bir toolu praktik olaraq istifadə etməyə icazə verir.

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent {

docker { image 'node:7-alpine' }

}

stages {

stage('Test') {

steps {

sh 'node --version'

}

}

}

}

Pipeline işlədikdə Jenkins avtomatik olaraq lazımi containerləri işə salıb, qeyd olunmuş addımları yerinə yetirəcək:

[Pipeline] stage

[Pipeline] { (Test)

[Pipeline] sh

[guided-tour] Running shell script

+ node --version

v7.4.0

[Pipeline] }

[Pipeline] // stage

[Pipeline] }

**Environment dəyişənlərinin istifadəsi**

Environment dəyişənləri qlobal olaraq (aşağıdakı nümunədə göstərildiyi kimi) yaxud da hər bir addımda təyin oluna bilər. Gözlənildiyi kimi, environment dəyişənlərini hər bir addımda təyin edilməsi o deməkdir ki, onlar təkcə müəyyən edildiyi mərhələdə yerinə yetirilir.

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent {

label '!windows'

}

environment {

DISABLE\_AUTH = 'true'

DB\_ENGINE = 'sqlite'

}

stages {

stage('Build') {

steps {

echo "Database engine is **${**DB\_ENGINE**}**"

echo "DISABLE\_AUTH is **${**DISABLE\_AUTH**}**"

sh 'printenv'

}

}

}

}

Environment dəyişənlərinin JenkinsFile-dan təyin olunması yanaşması - build və testi Jenkins’in içində islətmək üçün müxtəlif yollarla konfiqurasiya etmək,

Makefile kimi instructing skripləri üçün çox əlverişli ola bilər.

Environment dəyişənləri Jenkins pluginləri tərəfindən də təyin oluna bilər. Müəyyən pluginlərin environment dəyişənlərinin adını və bu pluginlərin izahını öyrənmək üçün dokumentasiyaya müraciət edin.

Environment dəyişənlərinin digər istifadə halı “dummy”crediential’ların build və test skriptdə təyin olunmasıdır. Çünki aydındır ki credentialları Jenkinsfile’ın içinə qoymaq yaxşı ideya deyil, Jenkins pipeline istifadəçilərə tez və təhlükəsiz bir şəkildə Jenkinsfileda əvvəlcədən təyin olunmuş credientialları onların dəyərlərini bilmək ehtiyacı olmadan götürməyə imkan verir.

**Testlərin və artefaktların qeydi**

Testing-in düzgün Delivery Pipeline’ın kritik bir hissəsi olduğu zaman, bir çox insan fail olmuş testlərə aid informasiyanı tapmaq üçün minlərlə sətr log’ları oxuyub axtarmaq istəmir. Bu işi asanlaşdırmaq üçün, Jenkins test runnerin işlədiyi və testlərin cavabları olan faylları qaytardığı müddətdə testlərin cavablarını yazır və özündə saxlayır. Jenkins adətən junit addımı ilə birlikdə qurulur lakin əgər bizim test runnerimiz JUNİT tipli XML reportları çıxara bilmirsə, elə əlvə pluginlər vardır ki, praktiki olaraq istənilən növ işlədilən test report formatlarını dəstəkləyir.

Bütün test nəticələri və artefaktları saxlamaq üçün post bölməsindən istifadə edəcəyik.

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Test') {

steps {

sh './gradlew check'

}

}

}

post {

always {

junit 'build/reports/\*\*/\*.xml'

}

}

}

Bu həmişə test nəticələrini qeydə alır və imkan verir ki, Jenkins onları izləsin, tendensiyaları hesablasın və onlar haqqında olan informasiyanı çatdırsın. Testlərdə fail olan pipeline “UNSTABLE” olaraq nişanlanır, web UI’da sarı rəngdə qeyd olunur. Bu qırmızı ilə göstərilmiş “FAİLED” halından fərqlidir.

Susmaya görə build unstable olduqda belə pipeline öz işini davam edir.Testdən keçmədikdə deployment prosesini yerinə yetirməmək üçün deklarativ sintaksisdə olan skipStagesAfterUnstable parametrindən istifadə edə bilərik. Skriptlənmiş sintaksisdə siz currentBuild.currentResult == 'SUCCESS' ifadəsini yoxlaya bilərsiniz.

Testlər uğursuz olduğu zaman local analiz və araşdırma üçün Jenkinsdən build artefaktları götürmək çox zaman əlverişlidir. Bu pipelinenin işlədiyi müddətdə yaranmış faylların, yəni artefaktların saxlanılması üçün jenkinsə quraşdırılmış artefakt saxlanılması sistemi sayəsində mümkün oldu.

Bu aşağıda göstərilən nümunədəki kimi archiveArtifacts addımı və file-globbing ifadəsi vasitəsi ilə yerinə yetirilə bilir.

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('Build') {

steps {

sh './gradlew build'

}

}

stage('Test') {

steps {

sh './gradlew check'

}

}

}

post {

always {

archiveArtifacts artifacts: 'build/libs/\*\*/\*.jar', fingerprint: true

junit 'build/reports/\*\*/\*.xml'

}

}

}

Əgər archiveArtifacts addımında birdən çox parameter qeyd edilibsə, bu halda hər bir parametrin adı birmənalı şəkildə kodda göstərilməlidir, missal üçün artifaktın yolunu və faylın adını göstərmək üçün  artifacts və fingerprint seçimini etmək üçün bunu işlədə bilərik.Əgər sadəcə artifaktın yolunu və adını təyin etmək istəyirsinizsə, siz artifact parameter adını yazmaya bilərsiniz. Məs: archiveArtifacts 'build/libs/\*\*/\*.jar'

Jenkinsdə testlərin və artifaktların yazılması komandanın müxtəlif üzvlərinə informasiyanın tez və asan formada çatdırılması üçün səmərəli yoldur.

**Cleaning up və ismarıclar**

Pipeline’ın iş müddətinin axırında Pipeline’ın post bölməsi qarantiyalı şəkildə işləsə, yekunlaşdırma, bildiriş və digər pipeline’ın son tasklarını yerinə yetirmək üçün bəzi bildiriş və digər mərhələləri əlavə edə bilərik.

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

stage('No-op') {

steps {

sh 'ls'

}

}

}

post {

always {

echo 'One way or another, I have finished'

deleteDir() /\* clean up our workspace \*/

}

success {

echo 'I succeeeded!'

}

unstable {

echo 'I am unstable :/'

}

failure {

echo 'I failed :('

}

changed {

echo 'Things were different before...'

}

}

}

Bildiriş göndərməyin çoxlu yolu vardır. Aşağıda bir emailə, Hipchat room’a, Slack kanalına necə bildiriş göndərildiyini göstərən bir neçə snippetlər vardır.

### Email

post {

failure {

mail to: 'team@example.com',

subject: "Failed Pipeline: **${**currentBuild.fullDisplayName**}**",

body: "Something is wrong with **${**env.BUILD\_URL**}**"

}

}

### Hipchat

post {

failure {

hipchatSend message: "Attention @here **${**env.JOB\_NAME**}** #**${**env.BUILD\_NUMBER**}** has failed.",

color: 'RED'

}

}

### Slack

post {

success {

slackSend channel: '#ops-room',

color: 'good',

message: "The pipeline **${**currentBuild.fullDisplayName**}** completed successfully."

}

}

İndi komanda pipeline’ın fail, unstable hətta uğurlu olduğu zaman xəbərdar olur və pipeline’ı shipping prosesi ilə bitirə bilir.

**Deployment**

Ən sadə CD pipeline ən azından Jenkinsfile’da qeyd olunmuş üç addımdan ibarət olmalıdır. Build, Test, Deploy. Bu bölümdə biz əsasən Doploy addımına fokuslanacağıq. Lakin qeyd etməliyik ki, Build və Test addımları Deployment prosesi üçün vacib mərhələlərdir.

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

options {

skipStagesAfterUnstable()

}

stages {

stage('Build') {

steps {

echo 'Building'

}

}

stage('Test') {

steps {

echo 'Testing'

}

}

stage('Deploy') {

steps {

echo 'Deploying'

}

}

}

}

**Addımlar Deployment Environmentlər kimi**

Geniş yayılmış patternlərdən biri aşağıda göstərildiyi kimi əlavə environmentlərin əlavə olunması üçün addımların artırılmasıdır.

stage('Deploy - Staging') {

steps {

sh './deploy staging'

sh './run-smoke-tests'

}

}

stage('Deploy - Production') {

steps {

sh './deploy production'

}

}

Bu nümunədə olan ./run-smoke-tests skripti vasitəsi ilə işlədilən smoke testlər production environment’ə gedəcək release’in kvalifikasiya ve validasiya üçün yetərlidir. Kodu avtomatik şəkildə productiona deployment edən bu tip pipeline continuous deployment’i implementasiya edir. Continious Deploymentin çox ideal yanaşma olduğuna baxmayaraq bir sıra insana bu yanaşma uyğun gəlməyə bilər, amma buna baxmayaraq onlar hələ də continuous delivery yanaşmasından faydalana bilərlər. Jenkins Pipeline hər ikisini dəstəkləyir.

**Davam etmək üçün insan müdaxiləsinin tələb olunması**

Çox vaxt mərhələlər arasında keçid etdikdə, əsasən də bu keçidlər enviromentlər arasında olduqda davam etməzdən əvvəl insan müdaxiləsi lazım ola bilər. Məsələn, qərar vermək lazımdır ki, proqram təminatı production environment’ə qoyulmaq üçün kifayət qədər yaxşıdır. Bunu input addımı vasitəsi ilə həyata keçirmək olar.Aşağıdakı misalda “Sanity check” addımı prosesi insan müdaxiləsi olmayana kimi bloklayır və davam etməyə qoymur.

*Jenkinsfile (Declarative Pipeline)*

pipeline {

agent any

stages {

/\* "Build" and "Test" stages omitted \*/

stage('Deploy - Staging') {

steps {

sh './deploy staging'

sh './run-smoke-tests'

}

}

stage('Sanity check') {

steps {

input "Does the staging environment look ok?"

}

}

stage('Deploy - Production') {

steps {

sh './deploy production'

}

}

}

}

**Nəticə**

Jenkinsin yuxarıda göstərilən imkanları əsas imkanlarıdır və pluginlər vasitəsi ilə Jenkins pipeline genişləndirilə, praktiki olaraq bütün istehsal prosesini avtomatlaşdıra bilər.

İşin əvvəlində qeyd olunan 3 problemi həll etmiş oluruq.

1. Versiyalanma problemi Git, versiyalanma sistemi vasitəsi ilə həll olundu və serverdə əlavə kopyaların saxlanılmasına ehtiyac qalmadı. İstənilən vaxt proqram kodlarını asanlıqla geri qaytarmaq, bir neçə developerin birlikdə bir proyekt üzərində işləmək və s. kimi imkanlar yaradıldı.
2. Developerin artıq production environmentə qoşulmağına ehtiyac qalmadı. Çünki, o kodu və yazılmış testləri Git vasitəsi ilə repositoriyaya push edir, orchestrator bu hadisəni emal edir və pipeline’ı işə salır. Öz növbəsində də pipeline’ın menecmenti ilə devops məşğul olur və developerin üzərində heç bir əlavə məsuliyyət qalmır.
3. Pipeline’ın bizim sistemimizi avtomatlaşdırması sayəsində developer heç bir serverə girib, əlavə rutin iş görüb, developmentin vaxtından itirmir. Proseslərin avtomatlaşdırıldığını nəzərə alaraq insan faktorunun təsirinin azaldığı qənaətinə gəlirik.

Məsuliyyətin bu yolla işçilər arasında bölüşdürülməsi işçilərin daha produktiv olması və işin daha tez getməsinə gətirib çıxarır.