The Twelve-Factor App

Bevezetés

A modern korban a szoftvert általában szolgáltatásként szállítják: a neve webes alkalmazás vagy szoftver-mint-szolgáltatás. A tizenkét tényezős alkalmazás egy módszertan ilyen szoftver-mint-szolgáltatás létrehozására, ami:

- A telepítés automatikussá tételéhez deklaratív formátumot használ, így a legkisebbre csökkenti az idő és költség-szükségletet, ami a projekthez új fejlesztő csatlakozása esetén felmerül;
- Az üzemeltetési környezetek közötti legnagyobb hordozhatóság érdekében egyértelmű és tiszta szerződést használ a futtató operációs rendszerrel;
- Alkalmas a modern felhős környezetbe történő telepítésre, elkerülve a szerver- és rendszer-adminisztráció szükségességét;
- A fejlesztés és az üzemeltetés között a legkisebb eltérés, valamint a folyamatos üzembehelyezés gyakorlatával elérhetővé teszi az optimális hatékonyságot;
- És skálázható, az eszközök, a környezet vagy a fejlesztési gyakorlat jelentős változtatása nélkül.

A tizenkét tényezős módszertan bármilyen programnyelven írt alkalmazáshoz felhasználható, és ez a háttérszolgáltatások (adatbázis, várakozósor, memória gyorsítótárazás, stb.) tetszőleges kombinációját használhatja.

Háttér

A dokumentum készítői közvetlenül több száz alkalmazás fejlesztésében és telepítésében vettek részt, munkájuk révén közvetett módon pedig több százezer alkalmazás fejlesztését, üzemeltetését és skálázását figyelték meg a Heroku platformon.

Ez a dokumentum szintetizálja minden tapasztalatunkat és megfigyelésünket a szoftver-mint-szolgáltatás alkalmazások széles körének dzsungelében. Ez hármas egyensúlyt biztosító ideális megoldás egy alkalmazás fejlesztésének és üzemeltetésének teljes életciklusa alatt. Különös figyelmet tudunk fordítani az alkalmazás időben történő természetes növekedésére és fejlődésére, az alkalmazás kódbázisán dolgozó fejlesztők együttműködésének a dinamikájára és egyben el tudjuk kerülni a költséges szoftverpusztulást

Célunk, hogy felhívjuk a figyelmet néhány olyan rendszerszintű problémára, amelyet láttunk a modern alkalmazásfejlesztés során, hogy közös szókincset biztosítsunk ezeknek a problémáknak a megvitatásához és kísérő terminológia segítségével széles körű fogalmi megoldásokat kínáljunk ezekhez a problémákhoz. A formátumot Martin Fowler könyvei inspirálják: Enterprise Application Architecture és a

Kinek érdemes ezt a dokumentumot elolvasni?

Minden fejlesztőnek, aki szolgáltatásként üzemelő alkalmazáson dolgozik. Üzemeltető mérnököknek, aki telepíti vagy kezeli ezeket az alkalmazásokat.

The Twelve Factors

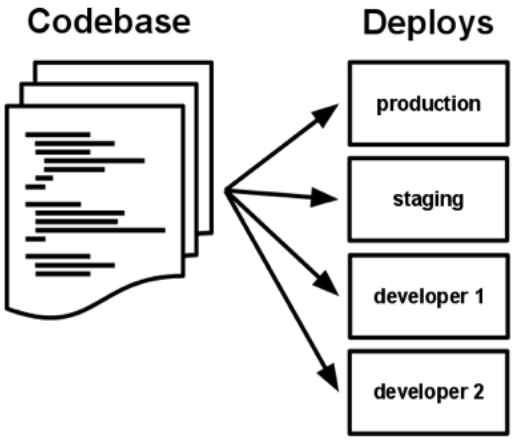
I. Kódbázis: egy kódbázis kódtárban kezelve, több telepítés

A tizenkét tényezős alkalmazást minden esetben verziókövető rendszerrel kezelünk, mint a Git, Mercurial, vagy Subversion. A verziókövető adatbázist és másolatát *kódtárnak* hívjuk.

A *kódbázi*s egy tetszőleges kódtár (központosított verziókövetési rendszernél, mint a Subversion), vagy bármilyen csoportja a kódtáraknak, amik a kezdő commit-et megosztják (decentralizált verziókövető rendszerek esetén, mint a Git)

Egy alkalmazáshoz mindig pontosan egy kódbázis tartozik:

- Ha több kódbázis van, akkor az nem lehet egy alkalmazás. Ebben az esetben elosztott rendszerről beszélünk több összetevővel. Ilyenkor tekintsünk minden összetevőt egy-egy alkalmazásnak, és ezek az alkalmazások külön-külön már alkalmasak arra, hogy a 12 tényezős alkalmazásfejlesztés követelményeinek megfeleljenek.
- Ha több alkalmazás ugyanazt a kódot megosztja (például ugyanazt a függvényt használják, ugyanazt a konstans/modell kódot tartalmazzák stb.), akkor ezek az alkalmazások megsértik a 12 tényezős alkalmazásfejlesztés előírásait. Ebben az esetben a megoldás az, hogy a megosztott kódot például egy könyvtárba ki kell szervezni, és az egyes alkalmazások közötti függőségek feloldásának mechanizmusával felhasználni, ahol szükség van rá.



II. Függőségek: Mindig explicit módon deklaráljuk és különítsünk el a függőségeket

A legtöbb programozási nyelvben rendelkezésre áll már csomagkezelési megoldás, ami támogatja könyvtár modulok becsomagolását és csomagban terjesztését, mint a CPAN a Perl fejlesztőknek vagy a Rubygems Ruby-hoz. A csomagkezelő megoldásunk segítségével egyrészt telepíthetjük a könyvtárainkat rendszer szinten, ekkor minden telepített alkalmazás használhatja őket (ezt hívják “site package”-nek) vagy telepíthetjük az alkalmazásunk könyvtárába (ezt pedig “vendoring” vagy “bundling” néven ismerhetjük).

A tizenkét tényezős alkalmazásfejlesztés soha nem feltételez rendszerszinten rendelkezésre álló könyvtárakat. Az összes függőségét pontosan deklarálja a *függőségi deklarációs jegyzékében*. Ezen

túlmenően az alkalmazás a függőségeket elkülönítő megoldást használ a futtatás során, ami biztosítja, hogy nincs implicit (nem pontosan definiált, hanem csak valamilyen egyéb körülményből következő) függőség, ami a környezetből "beszivárog". A teljes és explicit függőségi követelményeket egységesen alkalmazza az üzemeltetési és a fejlesztési környezetben is.

Például a Bundler parancs szolgál Ruby környezetben Gemfile formátumban a függőség meghatározásához és a bundle exec parancs a függőségek leválasztásához. Pythonban két külön eszköz létezik ezekhez a lépésekhez – Pip a deklarációs jegyzékhez és a Virtualenv az elkülönítéshez. Még C nyelven is megoldható az Autoconf segítségével a jegyzék legyártása, a statikus linkelés pedig az elkülönítést biztosítja. Nem számít, hogy milyen kombinációt alkalmazunk, a függőségek jegyzékbe vétele és a környezet leválasztása mindig együtt kell, hogy jelen legyen az alkalmazásfejlesztés során – ha csak az egyiket használjuk, nem teljesítjük a 12 tényezős alkalmazásfejlesztés követelményeit.

Az egyik nagy előnye a függőségek megfelelő kezelésének az, hogy ha új fejlesztő csatlakozik, akkor a környezet telepítése egyszerűen a kódbázis letöltését jelenti a gépére, ezen túlmenően csak a programozási nyelvre és a csomagkezelőre van még szükség semmi egyébre. Ezzel a fejlesztő egy meghatározott *build paranccsa* mindent be tud állítani. Például a build parancs a Ruby/Bundler környezetben bundle install, míg a Clojure/Leiningen fejlesztők a lein deps parancsot használják.

III. Konfiguráció: a konfigurációs beállításokat tároljuk a környezetben

Az alkalmazás *konfigurációja* minden olyan, ami valószínűleg változik az egyes telepítések között (tesztelés, üzemeltetés, fejlesztői környezetek stb.). Ideértve:

- Az adatbázis elérések kezelése, Memcached elérés és további háttérszolgáltatások elérése
- Hozzáférési információk külső szervízekhez mint az Amazon S3 vagy a Twitter
- Telepítésenként különböző értékek, mint például a gép azonosítója, amire telepítünk

Az alkalmazások néha konstansként tartalmaznak beállítási információkat. Ez megsérti a 12 tényezős alkalmazásfejlesztési alapelveket, aminek teljesítéséhez szigorúan szét kell választani a konfigurációs értékeket a forráskódtól. A konfiguráció telepítésenként jelentősen különbözik, a forráskód nem.

Ennek a szigorú szétválasztásnak egy jó lakmusz-próbáját teljesíti az alkalmazásunk, ha a kódbázis bármelyik pillanatban nyílt forráskódúvá tehető anélkül, hogy érzékeny információ kompromittálódna.

Figyelem, a konfigurációnak ez a definíciója nem terjed ki az alkalmazások belső konfigurációjára, mint a Rails esetében a 'config/routes.rb', vagy a Spring esetében ahogy az egyes modulok kapcsolódnak, mert ezek az információk a különböző telepítések esetében nem különböznek.

A konfiguráció egy másik megközelítése, ha a konfigurációs állományainkat, mint a Railsnél a 'config/database.yml' nem adjuk hozzá, így nem is tároljuk a kódtárunkban. Ez a konstansok használatához képest egy nagy előrelépés, mivel azokat kódtárban tároljuk, azonban továbbra is gyenge pont: nagyon könnyű véletlenül vagy figyelmetlenségből felvenni a kódtárunkba az egyik checkin-nél. És könnyen vezethez ahhoz az állapothoz, amikor a konfigurációs információink szétszórva, különböző helyeken és különböző formátumokban szaporodnak, egyre nehezebben áttekinthetővé téve a beállításokat. Továbbá, ezek a formátumok általában nyelvhez és keretrendszerhez igazodó formát vesznek fel.

A 12 tényezős alkalmazás a konfigurációs információkat *környezeti változókban tárolja*. Ezeket gyakran rövidítjük angolul az Environment Variables után env vars-nak vagy env-nek. A környezeti változókat - ellentétben a konfigurációs állományokkal- igazán könnyű telepítésenként különböző értékkel beállítani, kevés esélyünk van a kódtárba véletlenül felvenni, és a speciális konfigurációs állományokkal (vagy más konfigurációs módszerrel, mint például a Java System Properties) ellentétben, nyelv és operációs rendszertől független megoldás.

Még egy érdekes nézőpontja a konfigurációkezelésnek, a csoportosítás. Néha az alkalmazások a konfigurációt nevesített csoportokba szervezik (gyakran hívják az ilyet "környezetnek"), az egyes telepítések után elnevezve, mint a 'development', 'test' vagy a 'production' környezet a Rails-nél. Ez a megoldás nem skálázható tisztán: amikor az alkalmazásból új telepítés készül, az új környezetnek új név kell, mint például: 'staging' vagy 'qa'. És ahogy a projekt bővül, a fejlesztők saját változókat hoznak létre, mint például 'tamas-staging', 'peter-staging', stb, ez kombinatorikus robbanáshoz vezethet, ami nagyon törékennyé teszi az alkalmazás konfigurációinak az egységes kezelését.

A 12 tényezős alkalmazásfejlesztésben a környezeti változók a beállítás egységei, mindegyik teljesen független a többi a többi környezeti változótól. Soha nincsenek "környezetenként" csoportosítva, helyette mindegyiket telepítésenként külön kezeljük. Ez a megoldás az alkalmazás az élettartama alatt a telepítések számával együtt könnyen skálázható.

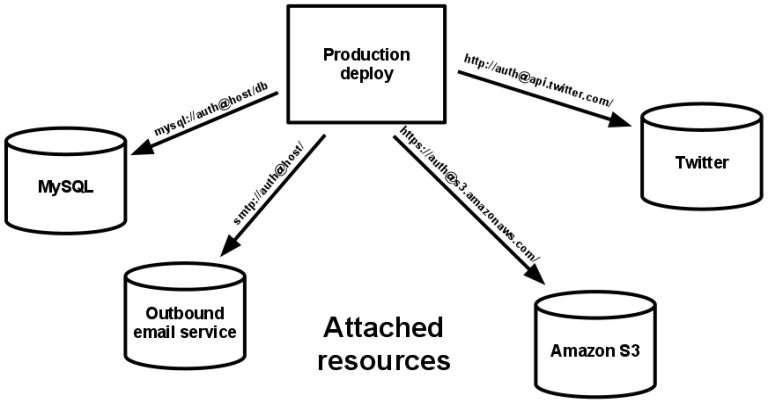
IV. Háttérszolgáltatások: kezeljük a háttérszolgáltatásokat csatolt erőforrásokként

Háttérszolgáltatás minden, amit az alkalmazás a normál működése közben hálózaton keresztül használ. Például idetartoznak az adatokat tároló rendszerek (mint a MySQL vagy CouchDB), az üzeneteket kezelő/sorbaállító rendszerek (mint a RabbitMQ vagy Beanstalkd), SMTP szolgáltatások a kimenő email kezeléséhez (mint a Postfix), és a gyorsítótárak (mint a Memcached).

A háttérszolgáltatásokat mint például az adatbázisokat általában ugyanazok az adminisztrátorok üzemeltetik, akik az alkalmazás telepített változatát. Az ilyen helyben kezelt és üzemeltetett szolgáltatásokon túlmenően az alkalmazás használhat harmadik fél által üzemeltetett szolgáltatásokat. Ilyenek például az SMTP (mint a Postmark), a telemetriát gyűjtő (mint a New Relic vagy Loggly), bináris csomagokat megosztó (amint az Amazon S3), vagy akár API-n keresztül elérhető fogyasztói (mint a Twitter, Google Maps, vagy Last.fm) szolgáltatók.

A 12 tényezős alkalmazásfejlesztési alapelveknek megfelelő alkalmazás nem tesz semmilyen különbséget saját, helyi vagy 3 személy által üzemeltetett szolgáltatások között Az alkalmazás számára mindegyik csatolt erőforrás, amit a konfigurációban tárolunk, legyen az URL-en vagy másmilyen elérésen keresztül elérhető. A 12 tényezős alkalmazás telepítése képes a helyi MySQL adatbázis helyett egy külső szolgáltató által üzemeltetettre (mint az Amazon RDS) váltani az alkalmazásunk kódbázisának a változtatása nélkül. Hasonlóan, a saját SMTP szervert le kell tudnunk cserélni külső szolgáltatóra (mint például a Postmark) forráskód változtatás nélkül. Mindkét esetben egyedül konfigurációban az erőforrás kezelésének a beállításán kell változtatni.

Minden egyes háttérszolgáltatás egy erőforrás. Például, egy MySQL adatbázis az egy erőforrás, így két MySQL adatbázis (az alkalmazás rétegben horizontális particionáláshoz felhasználva) két különböző erőforrásnak minősül. A 12 tényezős alkalmazás ezeket csatolt erőforrásként használja, ez gyenge csatolást jelent az erőforráshoz csatlakozó alkalmazás telepítései felől.



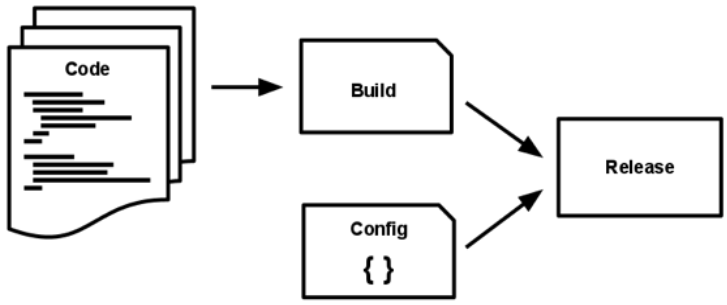
Az erőforrásokat az alkalmazás telepített példányától később csatolhatjuk és eltávolíthatjuk, ha akarjuk. Például, ha az adatbázis működésképtelenné válik fizikai meghibásodás miatt, akkor az alkalmazás rendszergazdája fel tud húzni egy új adatbázist a korábbi mentésből. Az aktuális éles adatbázist le tudja választani, és az új adatbázist csatlakoztatni tudja, anélkül, hogy a kódbázison változtatni kéne.

V. Előállítás, telepítés, üzemeltetés: egyértelműen válasszuk külön a forráskód futtathatóvá alakítását (BUILD), telepítését (RELEASE) és telepített alkalmazásunk üzemeltetését (RUN)

A kódbázis három szakaszban formálódik át (nem fejlesztői környezetbe) telepített alkalmazássá:

- Az *előállítási szakasz* az a transzformáció, amikor a forráskód kódtára átalakul futtatható alkalmazássá, amit *build*-nek hívunk. A forráskódnak vesszük azt a változatát, amit a fejlesztési eljárás elhelyezett a kódtárban, hozzávesszük a függőségeket és elkészítjük futtatható alkalmazást, valamint a szükséges hozzávalókat (képek, egyéb bináris állományok, stb.)
- A *telepítési szakaszban* vesszük az előállítási szakaszban előállított futtatható alkalmazást, és kombináljuk az aktuális telepítési konfigurációval. Az eredményül kapott kiadás, amit hívunk *release*-nek is, tartalmazza a futtatható állományokat plusz a megfelelő konfigurációt, és azonnal alkalmas arra, hogy a futtatókörnyezetben futtassuk.
- Az *üzemeltetési szakaszban* (nevezzük "runtime"-nak is) a választott kiadást használva az alkalmazás megfelelő folyamataiból indítva futtatjuk az alkalmazást a végrehajtási környezetben.

A 12 tényezős alkalmazásfejlesztés élesen elkülöníti tehát az előállítási, telepítési és az üzemeltetési szakaszokat. Például kizárja, hogy valaki a végrehajtásban lévő kódon változtasson, mivel ezt követően lehetetlen ezeket a változtatásokat a fordítási szakaszba visszaküldeni.



Általában a telepítési segédeszközök kínálnak kiadási verziókat kezelő eszközöket is, legalábbis lehetőséget adnak arra, hogy egy telepített változatról visszaálljunk egy korábban telepítettre. Például a Capistrano a kiadási változatokat a releases nevű mappában tárolja, ahol az éppen aktuális változat egy szimbolikus link a megfelelő könyvtárra. A rollback parancs így egyszerűen vissza tud állni az előző kiadási változatra.

Minden kiadási változatnak egyedi azonosítóval (ID) kell rendelkeznie, mint például a készítésének időpontja (vagyis 2011-04-06-20:32:17) vagy egy folyamatosan növekvő szám (mint a v100). A kiadási változatok összesége tulajdonképpen egy folyamatosan bővülő főkönyv. A kiadási változatok létrejöttük után már nem változhatnak. Bármilyen változáshoz új kiadást (release) kell készíteni.

A fordítási szakaszt az alkalmazás fejlesztői kezdeményezik, amikor új kódot tárolnak be a kódtárba. A végrehajtás pont fordítva: automatikus indítható, ha a szerver újraindul vagy az összeomlott folyamatot a folyamatkezelő újraindította. Ezért aztán a végrehajtásnak olyan kevés elemből kell állnia, amennyire csak lehetséges, hiszen az éjszaka közepén kialakuló probléma esetén a fejlesztők nem állnak mindig rendelkezésre. A fordítási folyamat már lehet összetett, mivel a hiba visszaszáll a fejlesztőre, aki az adott változatért felel.

VI. Folyamatok: az alkalmazást egy vagy több állapot nélküli folyamatként futtassuk

Az alkalmazás a futtatókörnyezetben egy vagy több *folyamatként* fut.

A legegyszerűbb esetben maga a kód egy különálló script, a futtatókörnyezet pedig a fejlesztő helyi laptopja rajta a nyelvi környezettel és a folyamatot parancssorból indítjuk (például így: Python my_script.py). A skála másik végén egy kifinomult rendszer éles telepítése több folyamattípust használhat, létrehozva nulla vagy több folyamatot.

A 12 tényezős alkalmazásfejlesztés folyamatai állapotmentesek és nem osztanak meg semmit. Minden nem átmeneti adatnak állapotnyilvántartó háttérszolgáltatásban a helye, tipikusan valamilyen adatbázisban.

A memóriaterületet vagy az állományrendszert rövid egy-tranzakciós átmeneti gyorsítótárként (cache) használhatjuk. Például, letöltünk egy nagy állományt, feldolgozzuk és az eredményt tároljuk az adatbázisban. A tizenkét tényezős alkalmazás sosem tételezi fel, hogy akár gyorsítás céljából valami a memóriában vagy lemezre mentve elérhető a jövőben egy másik kérés kiszolgálásakor vagy egy jövőbeni feladat végrehajtásakor - mivel minden folyamattípusból több létezhet, és nagy a valószínűsége, hogy a jövőbeni kérést már egy másik folyamat fogja kiszolgálni. Még akkor is, ha csak egy folyamatunk van, az újraindítás (ezt kiválthatja új kód telepítés, konfiguráció változás, vagy éppen a futtatókörnyezet áthelyezheti a folyamatot egy másik fizikai elérésre) általában mindent helyi állapotot (mint például a memória és az állományrendszer) eltöröl.

Az Asset csomagolók, mint a django-assetpackager az állományrendszert használják a lefordított elemek gyorsítótárazásához. A tizenkét tényezős alkalmazásfejlesztés jobban szereti, ha ezeket a fordításokat az előállítási szakasz során tesszük meg. Vannak olyan Asset csomagolók, mint a Jammit és a Rails asset

pipeline amiket konfigurálhatunk annak érdekében, hogy az Assetek csomagolását az előállítási szakasz során végezzék el.

Egyes webes rendszerek “post-it munkamenetek”-re támaszkodnak – vagyis gyorsítótárazzák a felhasználó munkamenetének az adatait az alkalmazás folyamatához rendelt memóriában, és elvárják, hogy az azonos látogató jövőbeni kéréseit ugyanehhez a folyamathoz irányítsuk. Ez ellenkezik a tizenkét-tényezős alkalmazásfejlesztés elveivel, ilyet sosem szabad használni vagy sosem szabad erre a működésre támaszkodni. A munkamenet állapotának adataihoz a jó jelölt egy olyan adatbázis, ami lejárati időt is kínál az adatokhoz, ilyen például a Memcached vagy a Redis.

VII. Hálózati port hozzárendelés: tegyük a szolgáltatásainkat port hozzárendeléssel elérhetővé

A webes alkalmazások néha webszerver konténerekben futnak. Például, a PHP alkalmazások egy Apache HTTPD vagy a Java alkalmazások futhatnak egy Tomcat belsejében.

A 12 tényezős alkalmazásfejlesztés követelményeinek megfelelő alkalmazás teljes egészében önálló nem támaszkodik egy további webszerver szolgáltatásaira ahhoz, hogy webes felületet hozzon létre. A webes alkalmazás a HTTP szolgáltatását egy porthoz rendelve kínálja és ezen a porton figyelve várja a hozzá intézett kéréseket.

A fejlesztő lokális gépén, a fejlesztési környezetben ez megoldható egy szolgáltatási URL-en keresztül. Például a *http://localhost:5000/* felkeresésével férünk hozzá az alkalmazásunk szolgáltatásaihoz. Ha telepítjük az alkalmazást, akkor pedig az útválasztó réteg kezeli a nyilvános címre érkező kéréseket és továbbítja a porthoz rendelt webes folyamathoz.

Ezt általában úgy oldjuk meg, hogy webszerver könyvtárat adunk az alkalmazásunkhoz függőségi definíció segítségével, mint például a Tornado Python környezetben, Thin a Ruby esetén, vagy Jetty a Javá-hoz és az egyéb JVM alapú nyelvekhez. Ez az *alkalmazás keretein belül* történik. Így az alkalmazásunknak csak a port hozzárendelésben kell megállapodnia -amin a kéréseket kiszolgálja- a futtatási környezettel.

A HTTP nem az egyetlen szolgáltatás, amit porthoz rendelve lehet szolgáltatni. Majdnem mindenfajta szerver szolgáltatást nyújtó alkalmazás képes a bejövő kéréseket kiszolgálni, ha a folyamatát porthoz rendeljük. Ideértve akár az ejabberd (XMPP-ről beszélünk) és a Redis (a Redis protocol esetén).

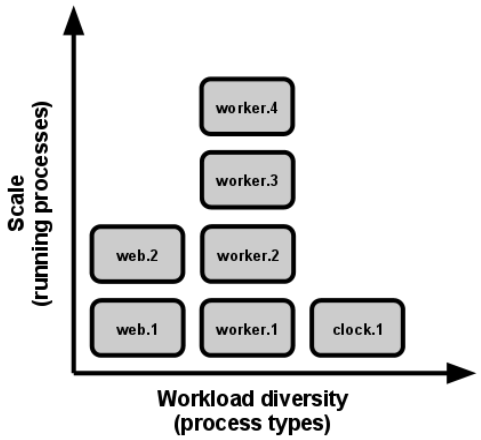
Vegyük észre, hogy a port-hozzárendeléses megközelítés eredményeként bármelyik alkalmazás képes háttérszolgáltatása lenni egy másik alkalmazásnak, ha a szolgáltatási URL-jét erőforrás szolgáltatásként az őt használó alkalmazás konfigurációjában megadjuk.

VIII. Párhuzamos folyamatok: szolgáltatásunkat a nagyobb terheléshez -az állapot nélküli-folyamat modellünknek köszönhetően méretezzük át menet közben

Minden számítógépes program futtatása egy vagy több folyamatként (process) jeleníthető meg. A webes alkalmazások sokféle folyamat végrehajtási formát ölthetnek. Például a PHP folyamatok az Apache által a háttérben indított alfolyamatokként (child process) futnak, a kérések mennyiségéhez igazodva. A Java folyamatok épp az ellenkezőleg, a JVM egy méretes főfolyamatot (uberprocess) szolgáltat, ami induláskor a rendszer (CPU és memória) erőforrásainak egy jelentős részét lefoglalja, és a párhuzamosságot belső szálakkal kezeli. Mindkét esetben a futó folyamat(ok) csak minimális szinten láthatók az alkalmazás fejlesztője szemszögéből.

A tizenkét tényezős alkalmazásokban a folyamatok elsőrendű állampolgárok. A folyamatok a tizenkét tényezős alkalmazásban nagyon erősen másolják a unix világ folyamat modelljét ami szolgáltatás daemon-okat futtat.Ezt a modellt felhasználva a fejlesztő a különböző terhelésekhez különböző *típusú folyamatokat* rendelve építheti fel az alkalmazását. Például a HTTP kéréseket kiszolgálhatják web folyamatok, a hosszú ideig futó háttérfeladatokat pedig kiszolgálhatják munkavégző (worker) folyamatok.

Ez nem zárja ki az egyes folyamatok saját belső párhuzamos tevékenységét (multiplexing) több szálon a futó virtuális gép (VM) belsejében, vagy az EventMachine, Twisted, vagy Node.js és hasonló környezetekben található az aszinkron/eseményvezérelt felépítést. De a virtuális gép (VM) alapú rendszer maga csak egyféle nagy lehet (függőleges skálázással a gépet tudjuk alatta megnövelni), így az alkalmazásnak képesnek kell lennie a különböző folyamatokat több gépre leosztva futni, amennyiben párhuzamos (vízszintes) skálázódásra van szükség.



A folyamatmodell kiválósága akkor ragyog fel igazán, ha a megnövekedett terhelésre válaszul az egyszerre kiszolgáló folyamatok számát meg kell növelni (vízszintesen kell skálázni). A tizenkét tényezős alkalmazás semmit sem megosztó, így párhuzamosan (vízszintesen) építhető természetének köszönhetően a párhuzamos végrehajtási kapacitás növelése egyszerű és megbízható művelet. A különböző folyamattípusok együttese és ezen belül az egyes folyamattípusokból futó folyamatok számát szokás *folyamatkialakításnak* (process formation) hívni.

A tizenkét tényezős alkalmazás sosem démonizál vagy ír PID állományt. Helyette az operációs rendszer folyamatvezérlőjére támaszkodik (mint a systemd, vagy felhős környezetben az elosztott folyamatok vezérlője, vagy a Foreman-hoz hasonló eszközök fejlesztéskor) a kimeneti adatfolyam kezeléséhez, a bedőlt folyamatokra adott válaszhoz és a felhasználó által kezdeményezett újraindítások és leállások kezeléséhez.

IX. Eldobhatóság: az elérhető legnagyobb hibatűrés gyors üzembehelyezéssel és egyszerű, gördülékeny leállítással

A tizenkét tényezős alkalmazás folyamatai *eldobhatóak*, ami azt jelenti, hogy nagyjából a parancs észlelésének pillanatában képes elindulni vagy leállni. Ez megkönnyíti a rugalmas méretezést, a kód vagy konfiguráció változásának gyors telepítését, és az üzemeltetési telepítés hibatűrését.

A folyamatoknak törekedniük kell az üzembehelyezésükhöz szükséges idő minimalizálására. Ideális esetben egy folyamatnak néhány másodpercig tart az indulási parancstól eljutni egészen addig, hogy a folyamat fut és kész a kérések vagy a feladatok fogadására. A gyors üzembehelyezési idő több teret biztosít a folyamat üzembehelyezéséhez és felskálázódásához; segíti a hibatűrést, ugyanis ha indokolt, a folyamatkezelő könnyebben át tudja mozgatni a folyamatot egy új fizikai eszközre.

A folyamat álljon le egyszerűen és gördülékenyen, ha megkapja a SIGTERM leállító parancsot a folyamatkezelőtől. Webes folyamatoknál ez elérhető, ha megszüntetjük a szolgáltatás portján a rendelkezésre állást (listening) - ezzel minden további kérést visszautasítva, engedélyezzük viszont a folyamatban lévő kérések teljesítését és csak ezután állítjuk le a folyamatot. Ebből következik, hogy ebben a modellben a HTTP kérések kiszolgálása rövid (nem több néhány másodpercnél), a hosszú ideig tartó hívásokkal történő folyamatos lekérdezés (long polling) esetén pedig a kliens zökkenőmentesen újrakapcsolódik, ha a kapcsolat megszakad.

Munkavégző folyamatok esetén az egyszerű és gördülékeny leállás elérhető az aktuális végrehajtás alatt lévő feladat visszahelyezésével a várakozósorba. Például a RabbitMQ üzenetsor esetén a munkavégző folyamat küldhet egy NACK jelet; míg Beanstalkd esetén a feladat automatikusan visszakerül a várakozósorba, ha a munkavégző felé megszakad a kapcsolat. A Delayed Job-hoz hasonló zárolás alapú rendszereknél fontos, hogy a zárolást ilyenkor a feladat adatrekordján feloldjuk. A modellünknek a következménye, hogy minden feladat ismételhető kell legyen, ami tipikusan a feladatok tranzakcióba burkolásával, vagy a műveletek idempotenssé tételével (az első feladatvégzés után a további ismétlések nem változtatják az eredményt) érhető el.

A folyamatoknak ezen kívül jól kell tűrniük a hirtelen halált is, a futtató hardver eszköz meghibásodásának az esetét is. Bár ez a leállási parancs (SIGTERM), hatására történő egyszerű és gördülékeny leállásnál sokkal ritkábban előforduló lehetőség, azért meg szokott történni. Az javasolt megközelítés egy hibatűrő üzenetsor használata, mint például a Beanstalkd, ami egyszerűen visszahelyezi a feladatot a várakozósorba, ha kapcsolat megszakad az ügyfélprogram felé vagy a kliens időtúllépésbe kerül. Akárhogy is, a tizenkét tényezős alkalmazás a nem várt és/vagy nem gördülékeny leállások kezelésére megfelelő felépítést alkalmaz. A kizárólag összeomlásra (crash-only) tervezési mód ezt a koncepciót jelenti logikus végkövetkeztetésként.

X. Egyensúly a fejlesztés és az üzemeltetés között: a fejlesztési és az üzemeltetési folyamatok legyenek annyira hasonlóak amennyire csak ez lehetséges

Történeti okokból jelentős különbségek vannak a fejlesztés (a fejlesztő az alkalmazás helyi telepítését élőben módosítja) és az üzemeltetés (az alkalmazás végfelhasználók által elérhető, éppen futó telepítése) között. Ezek a különbségek és az ezekből adódó problémák három területen jelentkeznek:

- Az idő különbség: A fejlesztő a kódon napokig, hetekig vagy akár hónapokig dolgozhat, mire a munkája az üzemeltetett alkalmazásba kerülne.
- A személyi különbség: A fejlesztő írja a kódot, az üzemeltető mérnök telepíti azt.
- Az eszköz különbség: a fejlesztő dolgozhat olyan csomaggal, amiben Nginx, SQLite és OS X van, az üzemeltetés viszont Apache, MySQL és Linux környezetre telepíti az alkalmazást.

A tizenkét tényezős alkalmazást eleve folyamatos telepítéshez tervezzük, hogy ez a különbség a fejlesztés és az üzemeltetés között kicsi legyen. Nézzük a három különbséget egyenként:

- Az időbeli különséget tegyük kicsivé: a fejlesztő megírhatja a kódot, és az órák, de akár percek múlva is az üzemeltetésbe kerülhet.
- A személyi különbséget tegyük kicsivé: a kódot író fejlesztőt szorosan bevonjuk a telepítésbe így közelről figyeli az üzemeltetésben az alkalmazás viselkedését.
- Az eszközökben megjelenő különbséget tegyük kicsivé: legyen a fejlesztési és az üzemeltetési környezet olyan hasonló, amennyire csak lehet.

Összegezve ebben a táblázatban:

	Hagyományos alkalmazás	Tizenkét tényezős alkalmazás
Telepítések között eltelik	Hetek	Órák
Kódot létrehozó kontra telepítő	Különböző emberek	Ugyanazon emberek
Fejlesztői kontra üzemeltetési környezet	Különböző	Hasonló amennyire csak lehetséges

A háttérszolgáltatások, mint az alkalmazás adatbázisa, az üzenetsor vagy a gyorsítótár olyan terület, ahol a fejlesztés és az üzemeltetés egyensúlya fontos. Sok nyelv kínál könyvtárakat, amik egyszerűsítik a hozzáférést a háttérszolgáltatáshoz, ideértve adaptereket a különböző típusú szolgáltatásokhoz. Néhány példa táblázatban.

Típus	Nyelv	Könyvtár	Adapter
Adatbázis	Ruby/Rails	ActiveRecord	MySQL, PostgreSQL, SQLite
Üzenet várakozósor	Python/Django	Celery	RabbitMQ, Beanstalkd, Redis

Gyorsítótár	Ruby/Rails	ActiveSupport::Cache	Memory, filesystem, Memcached

A fejlesztők néha nagyon vonzónak találják, ha fejlesztés közben pehelysúlyú háttérszolgáltatást használhatnak a saját fejlesztési környezetükben, míg erős háttérszolgáltatások kerülnek az üzemeltetési környezetbe. Például fejlesztéshez SQLite-ot üzemeltetéshez viszont PostgreSQL-t használni; vagy helyben a folyamat memóriáját, üzemeltetéskor pedig Memcached-et használni gyorsítótárazáshoz.

A tizenkét tényezős fejlesztő ellenál annak a kísértésnek, hogy más háttérszolgáltatást használjon a fejlesztési és az üzemeltetési környezetben, még akkor is, ha az adapterek elméletileg bármilyen háttérszolgáltatások közötti különbséget eltüntetnek. A háttérszolgáltatások közötti különbségek azt jelenti, hogy apró inkompatibilitási problémák merülhetnek fel azt eredményezve, hogy a kód, ami működött és a teszteken megfelelően teljesített fejlesztési és tesztkörnyezetben, üzemeltetés közben hibára fut. A hibáknak ez a típusa súrlódást hoz létre, ami a folyamatos telepítést akadályozza. Ennek a súrlódásnak a negatív hatása a folyamatos telepítésre és az alkalmazás életciklusára összegzett költsége rendkívül magas.

A pehelysúlyú háttérszolgáltatások ma már kevésbé vonzóak, mint korábban voltak. A modern háttérszolgáltatásokat, mint a Memcached, a PostgreSQL és a RabbitMQ a modern csomagkezelési rendszereknek köszönhetően - mint a Homebrew és az apt-get - már egyáltalán nem körülményes telepíteni és futtatni. Alternatív megoldásként a deklaratív telepítő eszközök, mint a Chef és a Puppet kombinálva a vékony virtuális rendszerekkel, mint a Docker és a Vagrant lehetővé teszi a fejlesztők számára, hogy az üzemeltetési környezethez nagyon hasonló fejlesztési környezetben dolgozhassanak. Ezen rendszerek telepítésének és használatának a költsége alacsony - összehasonlítva a fejlesztési és üzemeltetési egyensúllyal, valamint a folyamatos telepítés előnyeivel.

A háttérszolgáltatások eléréséhez adaptereket használni továbbra is hasznos, mivel az újabb háttérszolgáltatásokra való áttérést relatív fájdalommentessé teszik. De az alkalmazás valamennyi telepítésének (fejlesztési környezet, tesztelés és üzemeltetés) azonos típusú és verziójú háttérszolgáltatásokat kell használnia.

XI. Naplók: kezeljük a naplókat esemény-folyamatként

A *Naplók* a futó alkalmazás működésébe nyújtanak betekintést. Szerver alapú környezetben gyakran a lemezen egy állományba írják őket (“naplóállomány”); de ez csak egy kimeneti formátum.

A naplók valamennyi futó folyamat és háttérszolgáltatás kimeneti adataiból összegyűjtött és idő szerint rendezett stream. A naplók nyers állapota tipikusan szöveges formátum ahol minden esemény külön sorban szerepel (akkor is, ha a kivételekből származó veremtartalom több sorba is kerülhet). A naplóknak nincs meghatározott elejük vagy végük hanem folyamatosan keletkeznek egészen addig, amíg az alkalmazás üzemel.

A tizenkét tényezős alkalmazás saját maga sosem foglalkozik a saját kimeneti adatfolyamának irányításával vagy tárolásával. Nem szabad naplófájlokat írnia vagy kezelnie. Helyette minden futó

folyamat a saját eseményeit gyorsítótár nélkül kiírja a sztenderd kimenetre stdout. A fejlesztés alatt a fejlesztő a saját képernyője előtt ülve fogja nézni, hogy megfigyelje az alkalmazás viselkedését.

A tesztelési vagy az üzemeltetési telepítéseknél minden folyamat adatfolyamát a végrehajtási környezet rögzít, összevonva az alkalmazás összes többi adatfolyamával, megtekintésre és hosszútávú archiválásra egy vagy több végcél felé irányítva.Ezek az archívumok nem láthatók és konfigurálhatók az alkalmazásból, helyette teljes egészében a futtatókörnyezet kezeli őket. Ehhez nyílt forráskódú napló forgalom irányító eszközök állnak rendelkezésre (mint a Logplex és a Fluentd).

Az alkalmazás eseményeinek adatfolyamát irányíthatjuk állományba, vagy valós időben figyelhetjük terminálablakban. A legfontosabb, hogy az adatfolyamot elküldhetjük egy naplóindexelő és elemző rendszerbe, mint a Splunk, vagy egy általános célú adattárház eszközbe, mint a Hadoop/Hive. Ezek a rendszerek elegendő teljesítményt és rugalmasságot biztosítanak az alkalmazások időbeli viselkedésének megismeréséhez, ideértve:

- Meghatározott múltbeli események keresése.
- A trendek tágabb nézőpontból történő ábrázolása (mint a percenkénti kérések száma).
- Aktív riasztás a felhasználó által létrehozott figyelésnek megfelelően (például egy riasztás jön létre, ha a percenkénti hibák száma meghalad egy bizonyos küszöbértéket).

XII. Adminisztratív folyamatok: futtassuk az adminisztrációs és felügyeleti feladatokat egyszer futó folyamatokként

A folyamat formátuma pedig legyen olyan folyamatok együttese, amit az alkalmazás rendszeres üzleti folyamatai (például webes kérések kiszolgálására) is használnak. Ezen kívül a fejlesztők gyakran egyszeri adminisztrációs vagy karbantartási feladatokat is szeretnének végrehajtani, mint például:

- Adatbázis migrációk végrehajtása (mint például manage.py migrate a Django-nál, rake db:migrate a Rails használatakor).
- Konzol használata (vagy másnéven REPL shell) tetszőleges kód futtatásához vagy akár az alkalmazás adatmodelljének használatához az éles adatbázis vizsgálata közben. A legtöbb nyelv biztosít REPL eszközt az értelmező alkalmazás parancssori paraméterek nélküli futtatásával (mint a python vagy perl) egyes esetekben pedig erre külön parancs van (az irb a Rubyhoz, rails console vagy a Rails-hez).
- Az egyszer használatos scripteket az alkalmazás kódtárában kell tárolni (egy php példa: php scripts/fix_bad_records.php).

Az egyszeri adminisztrációs folyamatokat ugyanolyan környezetben kell futtatni, mint az alkalmazás rendszeres, hosszan futó folyamatait. Ugyanazt a kódbázist és konfigurációt használó telepítésen futtatva, mint amit a telepítés egyéb folyamatai használnak. Az adminisztrációs scripteket/kódokat az alkalmazással együtt kell szállítani a szinkronizációs problémák elkerülése érdekében.

Ugyanazt az elkülönítési megoldást kell használni minden folyamattípusnál. Például, ha a Ruby webes folyamata a bundle exec thin start parancsot használja, akkor az adatbázis migrációnak bundle exec rake db:migrate-et kell használnia. Hasonlóan, a Virtualenv-et használó Python programnak a saját telepítésű

bin/python könyvtárát kall használnia a Tornado webkiszolgáló indításához ugyanúgy ahogy manage.py felügyeleti folyamathoz.

A tizenkét tényezős fejlesztés határozottan támogatja azokat a nyelveket, amik alapból kínálnak REPL környezetet és megkönnyítik az egyszeri adminisztrációs scriptek futtatását. A fejlesztők a saját gépükön az egyszeri admin scripteket egyből az alkalmazás munkakönyvtárából tudják parancssorban futtatni. Éles telepítés esetén a fejlesztők ilyen folyamatok futtatásához ssh-t vagy más, a telepítés futtatási környezete által biztosított távoli parancsvégrehajtási megoldást használhatnak.

Források

https://12factor.net/