Budapesti Corvinus Egyetem
Gazdálkodástudományi Kar
Számítástudomány Tanszék

VERZIÓVÁLTÁS SZOLGÁLTATÁSFOLYTONOS ALKALMAZÁSOKBAN

Készítette: Oroszi Róbert
Gazdaságinformatikus szak
E-Business szakirány
2013

Konzulens:
Balogh Zoltán

Tartalomjegyzék

Rövidítésjegyzék

IRC	IRC - Internet Relay Protocol
Jabber	TODO

1. fejezet

Bevezetés

Az interneten böngészve a felhasználó gyakran észre sem veszi, de egy weboldalon történő két kattintása között elképzelhető, hogy az adott rendszer kétszer is verziót váltott, azaz frissítette a rendszerét. De hogyan történik mindez? Hogyan lehetséges leállás nélkül naponta többször verziót váltani, akár földrajzilag elosztott rendszereket populálni? Ezekre a kérdésekre kíván a szakdolgozat válaszokat adni. Bemutatásra kerül a continuous integration folyamat, ezekhez használt rendszerek, eszközök.

A dolgozat témája alapvetően a webes alkalmazásokat hivatott bemutatni, azonban kitér az asztali alkalmazásokra is.

A verzióváltás téma azért rendkívül izgalmas, mert akkor működnek jól, ha a végfelhasználó nem veszi észre, azonban ezeknek a rendszerek megtervezése, kiépítése, működtetése néha komolyabb mérnöki probléma, mint azok az alkalmazások, amelyek számára készültek.

2. fejezet

Continuous Integration

> A Continuous Integration - azaz a folyamatos integráció - egy olyan szoftverfejlesztési módszertan, amelyben a fejlesztőcsapat tagjai által írt kód napi rendszerességgel kerül (vagy automatizálással minden funkció, javítás implementálása után) integrálásra a korábbi fejlesztések közé. Minden új kód integrálása során automatizált tesztek ellenőrzik, hogy a rendszerbe való illesztés során okozott-e valamilyen hibát az új kódrészlet, illetve megfelel-e az adott fejlesztőcsapat által meghatározott minőségi kritériumoknak és ennek eredményéről a lehető leghamarabb visszajelzést ad. *Continuous Integration (original version)* 2002

A szoftverfejlesztés során általában egy projekten több fejlesztő is dolgozik a kód különböző vagy akár egyazon részein is. A fejlesztők az alkalmazás másolatával dolgoznak a saját munkaállomásukon (nem pedig közvetlenül egy helyen). Ha egy feladat elkészül, akkor fel kell másolni a módosított fájlokat egy közös tárolóba/szerverre. Viszont a felmásolás előtt mindenképpen szükséges frissíteni a saját lokális példányukat, hogy elkerüljék a ütközéseket, illetve egymás munkájának felülírását, ezt a folyamatot nevezik integrációnak. Azonban előfordulhat, hogy a központi tároló és a saját lokális másolat között akkor különbség lehet, hogy nagy mértékben kénytelen a fejlesztő módosítani a saját fejlesztését, majd a javítás elvégzése után következhet ismét egy frissítés (integrálás), majd esetleg a megismételt javítás. Ez mint látható, egy ördögi kör. Ennek a megoldására jött létre a Continuous Integration, amelynek segítségével a fejlesztők adott időközönként megismétlik az integrációs lépést, hogy minél kevésbé térjen el a lokális verzió a központi tárolóban lévő verziótól.
Hiába tűnik ez egy triviális és egyszerű megoldásnak, ez a megközelítés csak a 2000-es évek elején született meg, viszont azóta töretlen népszerűségnek örvend. Mára a folyamatos integráció összekapcsolódott az automatikus fordítással (build automation), azonban alapvetően nem szükséges része. Tehát például egy céges előírás, amely megköveteli, hogy a fejlesztők kötelesek minden reggel a lokális másolatuk frissítésére, is tulajdonképpen folyamatos integrációnak tekinthető, mert ezáltal megvalósítják a rendszeres integrációt. Ezek alapján kijelenthető, hogy a Continuous Integration valójában csak egy verziókezelő (pl.: git, Subversion, Mercurial) használatát követeli meg.
Az automatizált fordítás habár nem alapvető része a folyamatos integrációnak mégis a fejlesztést és a fejlesztők munkáját nagyban megkönnyíti. Ez a művelet történhet bizonyos időközönként (munkaidő után éjszaka) vagy bizonyos eseményekre, mint

például ha fejlesztők feltöltik a közös tárolóba a fájljaikat (commit). Az integrációs lépések és az automatikus fordítás közé érdemes lehet beépíteni a tesztelést, amely garantálja, hogy az új funkciók nem törik el a régi funkciókat. Továbbá érdemes az integrációról, a tesztelésről, és a fordításról riportokat generálni, melyekkel az eredmények - egy nem fejlesztő számára is - érthetőbb formába kerülnek.
A folyamatos integrációra, tesztesetek futtatására, riportok értelmezésére már sok szoftver elérhető, többek között a Bamboo, amely a Jira és az Confluence mögött álló Atlassian terméke, vagy a Travic CI, amely ingyenes szolgáltatásként elérhető bármilyen nyílt forráskódú szoftver számára, illetve a mára de-facto rendszernek tekinthető Jenkins, amely a következő fejezetben kerül bemutatásra.

2.1. Jenkins

https://jenkins-ci.org/

A Jenkins egy ingyenesen elérhető, nyílt forráskódú folytonos integráció támogató eszköz. A legtöbb verziókezelő rendszert támogatja és több mint 300 kiegészítő érhető el hozzá, melyek segítségével könnyedén testreszabható. Emellett támogatja az xUnit teszt keresztrenderek riportjait.

EZ KEVÉS!

2.2. TDD

Test Driven Development

Manapság a folyamatos integráció és az automatikus fordítás egyik legfontosabb velejáró kulcsszava a TDD, azaz a teszt vezérelt fejlesztés.
A TDD használatához egy elég erős szemléletváltásra van szükség, ezért legalább annyi ellenzője van, mint támogatója. A teszt vezérelt vagy teszt irányított fejlesztés a nevével ellentétben nem egy tesztelési megoldás, hanem sokkal inkább tervezés. Johansen, 2011a
Egy alkalmazás jó és jól működéséhez könnyen bővíthetőnek kell lennie, hogy a termék, szolgáltatás meg tudjon újulni, a továbbfejlesztés zökkenőmentesen tudjon zajlani. De hogyan történik az új funkciók tervezése, beépíthetőségi megvalósításának tervezése?
A TDD ezt próbálja elősegíteni azáltal, hogy a teszteseteket még a tervezés fázisában kell megírni, így a problémák a munka korai fázisában kiderülhetnek. A TDD teszt keretrendszerek általában könnyen olvashatóak még a fejlesztésben

kevésbé járatos projekt résztvevők számára is, a 2.1 ábra ezt kívánja bemutatni. 2.1

```
1  suite( "This test covers the login window for the application.",
2    function(){
3
4    suiteSetup(function(){
5      this.application = new Application();
6    });
7
8    test( "after logging in, the username should be 'MyMscThesis'",
9      function(){
10       var user = this.application.login({
11             username: "MyMscThesis",
12             password: "MySecretPassword"
13           });
14       user.name.should.equal( "MyMscThesis" );
15   });
16
17 });
```

2.1. ábra. Egy TDD teszt mocha és should keretrendszerekkel

Mivel a TDD teszteseteket a fejlesztés alatt folyamatosan futtatják - ellenben az egységtesztekkel, melyek általában az integráció során futnak le -, ezért kimondottan gyorsnak, mindenhol, bármilyen sorrendben futtathatónak kell lenniük, mert egyébként a fejlesztők nem fogják felhasználni őket. Teszteseteknek csak abban az esetben kellene változniuk, ha a kód ezt megköveteli, illetve ha a specifikáció változik.
A fejlesztési folyamat négy lépésre bontható:

Feladatok meghatározása:

Ebben a lépésben az ügyfél igényeknek megfelelő funkcionalitást kell feladat meghatározássá alakítani tesztesetek formájában. Meg kell határozni, hogy mit kellene tennie a rendszernek. Fontos, hogy nem azt kell itt kitalálni, hogy az adott fejlesztő hogyan valósítsa meg az adott feladatot, hanem hogy mit kell majd megvalósítani. A mit meghatározása által a fejlesztő is jobban megérti a feladatot, kisebb a félreértés lehetősége. A teszteset megírása után

következik a megvalósítás.

Megvalósítás:

Ezután következik a hogyan valósítsuk meg az előre definiált feladatot. Azután meg is kell valósítani. A megvalósításnak kellene a legkönnyebb résznek lennie, ha mégsem könnyű akkor a következő problémák fordulhattak elő.

Túl nagy feladatot határoztunk meg az előző lépésben, így az a feladat, hogy kisebb feladatokra bontsuk. Felmerül a kérdés, hogyan lehet könnyű a megvalósítás, ha előtte még egy alkotóelemet is el kell előtte készíteni? Ilyenkor az adott alkotóelem feladatait kell meghatározni úgy, hogy a fejlesztők lesznek az ügyfelek és a ő igényeiket kell kielégíteni, és TDD alapján lefejleszteni.

Ha nem nagy lépésről van szó de mégis nehéz megvalósítani, akkor refaktorálni kell az adott részt. Ez lehet azért mert koszos a kód, vagy nincs jól megtervezve.

Ellenőrzés:

A megfelelő eszközzel le kell ellenőrizni, hogy sikeresek lettek-e a tesztek. Ennek gyorsnak és könnyűnek kell lennie. (A tesztek nem futhatnak néhány másodpercnél lassabban.) Ezáltal folyamatosan ellenőrzött, tervezett és fejlesztett lesz a kód.

Tisztítás:

Ha sikeresen le lehet futtatni a teszteket, akkor következik a kód tisztítása. A működő kódot át kell nézi, a duplikációkat eltüntetni.

Beszédesebb neveket választhatunk a változóinknak. Mivel az ügyfél számára fontos funkcionalitás már le van tesztelve, ezért ez a művelet már könnyedén elvégezhető, ugyanis ha például egy metódus neve elgépelésre kerül, akkor rögtön jelez a teszt, hogy hiba van. Ettől tisztább és karbantarthatóbb lesz a kód.

Érdemes megfigyelni, hogy az összes lépés megfeleltethető a tervezés egy-egy részének. A meghatározott feladatok automatizálásának köszönhetően lehetséges, hogy kis biztonságos lépésekben történjen a rendszer felépítése és tervezése. Segíti a

feladat megértését, rákényszerít a könnyű megvalósíthatóságra a folyamatos tisztítás és újratervezés segítségével.

TDD előnyei:

- Refaktorálást segíti.
- A kód módosítása könnyebb, hiszen hiba esetén a tesztek eltörnek, amely azonnali visszacsatolást a fejlesztő számára.
- Könnyebb egy tesztelhető kódot refaktorálni (a tervezés miatt).
- Az is segítség lehet, hogy hol nincs hiba.
- Segít tesztelhetővé tenni az alkalmazást.
- Rákényszerít, hogy ne legyen az alkalmazásban spagetti kód.
- Felesleges funkció nem kerül megvalósításra, csak az ami a teszthez szükséges.
- Előre kell tervezni.
- Gyors, folyamatos visszajelzés kapható a funkció állapotáról (nem csak az adott fejlesztőknek, de a csapat többi tagjának és a projektmenedzsernek is).
- Jobban ellenőrizhető a munka.
- Van, hogy egy funkciónak nincs látható eredménye egy hétig. Ezzel szemben a TDD-nél naponta meg lehet mondani, hogy mennyi sikeres tesztet sikerült írni.
- Segít megérteni a feladatot a példákon keresztül.
- Időcsökkentő tényező a hibajavításnál és a refaktorálásnál.
- Hibajavításnál segíthet pontosabban megjelölni a hiba helyét.
- Akár dokumentációként is szolgálhat a teszt. Példakódnak tekinthető.
- Biztosítja, hogy az új kód nem érint más tesztelt egységet.
- Ha nincs TDD, akkor gyorsabban készül a szoftver, de nehezebben módosítható később.

- A TDD tisztítás része akár kódfelülvizsgálatnak is tekinthető.
- Stabilitást elősegítheti.

TDD hátrányai:

- Nő a fejlesztési idő (refaktorálásnál csökken).
- Ha nem tiszta, hogy mit kell tenni az adott feladattal könnyen előfordulhat, hogy rossz teszt kerül megírásra, amit át kell majd írni. (Újabb időnövelő tényező.)
- Menet közben történt koncepcióváltásnál ki kell dobni a teszteket. (Újabb időnövelő tényező.)
- A program működése nem lesz hibamentes, ha tesztek sikeres lefutnak.
- Nem csodafegyver. A rendszer tesztelésének (minőségbiztosításának) csak egy kis részét kéne, hogy képezze. (Acceptance tesztelés, integrációs tesztelés mellett)
- Csak tapasztalt fejlesztőkkel érdemes használni.
- A tervezést nem mindig lehet úgy alakítani, hogy az megfeleljen a TDD-nek.
- Hálózattal, fájlrendszerrel kapcsolatos dolgokra nem használható.
- Páros programozásban a legjobb használni.
- A tesztek írása unalmas lehet egyesek számára. Nagy fegyelemre van szükség.
- Nehéz belerázódni, ezért arra a következtetésre lehet jutni, hogy semmi értelme.
- TDD-ben tapasztalt párral lenne a legideálisabb.
- Nehéz megmagyarázni a menedzsereknek/ügyfeleknek, hogy az elején miért tart ennyi ideig a fejlesztés.

TODO: http://bitroar.posterous.com/when-tdd-fails

3. fejezet

Continuous Notification

A folyamatos integrációnál mivel naponta több verzió is kikerülhet, sőt párhuzamosan akár több csapat vagy akár fejlesztő is élesíthet funkciókat, ezért nagyon fontos, hogy a rendszer folyamatosan tájékoztassa a csapatokat az élesített funkciók működéséről hibáiról illetve a többiek által fejlesztett funkciók állapotáról. Mindehhez két dologra van szükség, a kommunikáció és a rendszer minden apró részének monitorozására, mert ezáltal a hibák gyorsan kiszűrhetőek, a reakció idő csökkenthető és mindenki valós időben láthatja, hogy mi történik a rendszerben, azaz transzparens lesz, amelynek köszönhetően a szervezetnek könnyebb a részegységek megismerése, felelősségi köreik meghatározása, megismerése.

3.1. Chat rendszer

A fejlesztői csapatok illetve a csapattagok közti kommunikáció rendkívül fontos, mely persze történhet verbálisan, azonban ez megszakíthatja a munkát, illetve távoli munkavégzés esetén (főleg ha a résztvevők más országból végzik a munkát) akkor sokkal jobb megoldás egy chatrendszer bevezetése, ilyenek lehetnek például:

- Jabber
- IRC
- HipChat
- Campfire
- Grove.io
- FlowDock

De miért kell chat, miért nem elég az email alapú kommunikáció?

A legtöbb szervezetben sajnos még mindig az email a legfontosabb kommunikációs eszköz, azonban az email lassú, nem alkalmas instant üzenetek küldésére, illetve gyakran kimaradnak a levelezésből megfelelő emberek, ezzel ellentétben viszont a cseten gyorsan lehet üzenetet váltani. Természetesen a csettel nem lehet kiváltani az emailt, a két eszközt kombinálva, egymást segítve lehet jól használni. Továbbá a cset mellett érvelve elmondható, hogy az emberek könnyebben elviselik a zajt csetet használva, mint az emailezésben.

3.1. táblázat. Chat rendszerek összehasonlítása

-	Jabber	IRC	HipChat	Campfire	Grove.io	FlowDock
Saját szerver	✓	✓	✗	✗	✗	✗
Mobil alkalmazás	✓	✓	✓	✗	✓	✗
Asztali alkalmazás	✓	✓	✓	✗	✗	✗
[h] Egy-az-egy beszélgetés	✓	✓	✓	✗	✗	✗
Csoportbeszélgetés	✓	✓	✓	✗	✗	✗
Audió, Videó	✓	✓	✓	✗	✗	✗
Email értesítés	✓	✓	✓	✗	✗	✗

Forrás: *See how HipChat stacks up against the competition.*,,

3.2. Naplózás, naplógyűjtés

Az alkalmazások viselkedésének monitorozása rendkívül fontos, ugyanis amint kikerülnek a belső homokozóból, akkor megváltozhatnak a reakcióik a felhasználók különböző tevékenységeire, ezeknek a problémák a felderítése használják a naplózás és a naplógyűjtést. Érdemes kiemelni, hogy a fejlesztési (development), tesztelési (staging/user acceptence testing) illetve az előnézeti (preview) szerverek is mind valamilyen homokozónak számítanak, hiszen valós felhasználók nem használják, a felhasználók szimulálása - akár tesztelők, akár automatizált tesztek által - szinte sosem tökéletesek.

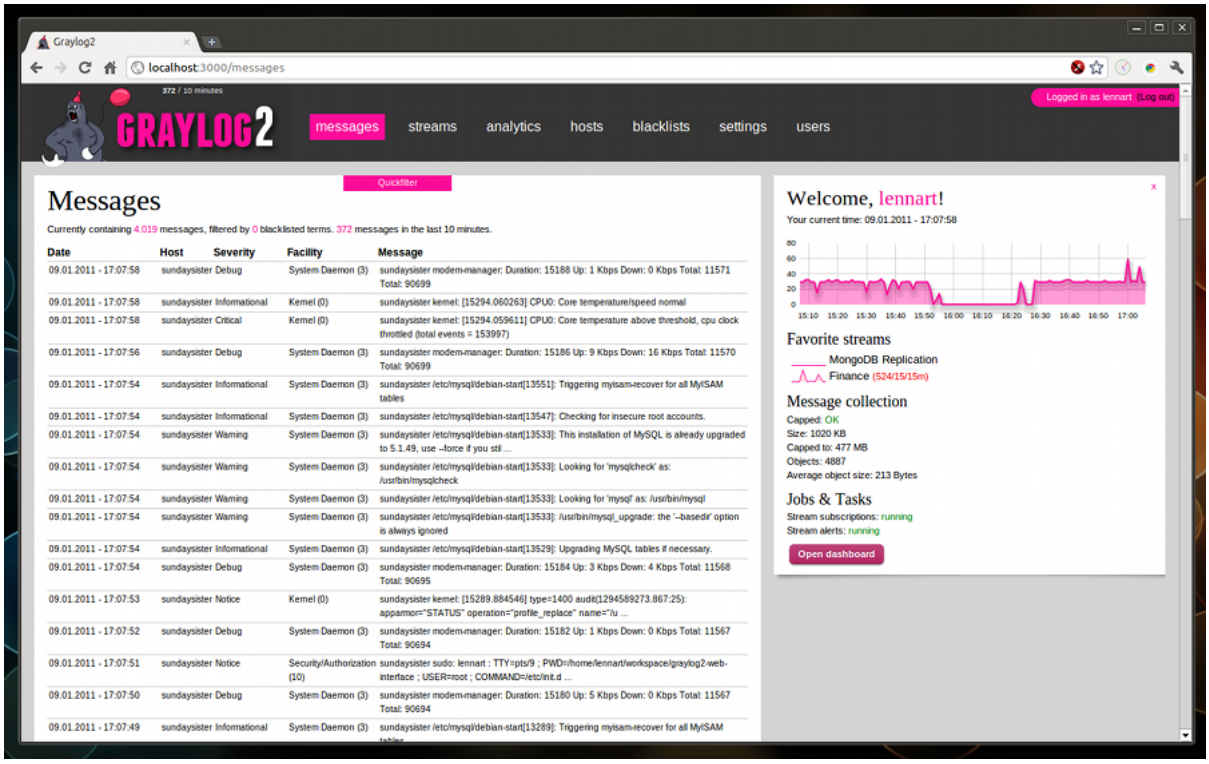
A jól használt naplózással a hibák hamarabb kiszűrhetőek, a hibák forrása könnyebben kiszűrhető, azonban minden rendszer naplójainak egyidejű figyelése egyszerűen lehetetlen. Ezért használnak a legtöbb rendszerben központi naplózást és naplógyűjtést, amelynek segítségével a hibák felfedezése, forrásának felderítése és a megoldása is könnyebben megoldható. Miért lehetne a központi naplózással könnyebben megoldani egy problémát? Amennyiben az alkalmazás példánya tegyük fel lassan válaszol, miközben a többi példány hiba nélkül működik (ezt egy könnyű ellenőrizni, hiszen egymás mellé helyezhető az alkalmazás példányok naplója), akkor egyértelmű, hogy a hiba a példányt futtató eszközön van csak jelen - a legtöbb esetben - amely megkönnyíti a szakembereknek a hibajelenség elhárítását.

Továbbá a központi naplózás könnyen implementálható a folyamatos értesítés munkafolyamataiba, amellyel egy-egy alkalmazás/termék felelős szakembere azonnal

értesülhet a fenálló hibáról, a javítást azonnal elkezdheti.
A piacon több naplógyűjtő alkalmazás is elérhető mind telepíthető, mind formában. A telepíthető és SaaS verzióban is elérhető többet a között a *GrayLog2*.

3.2.1. GrayLog2

A GrayLog2 segítségével könnyen monitorozható az összes szoftveres hiba, legyen szó akár adatbázisról, az operációs rendszer hibáiról vagy alkalmazáshibáról. Emelett lehetőséget nyújt nem csak hibajellegű üzenetek tárolására, hanem a hibakereséshez szükséges üzenetek megjelenítésére is.



3.1. ábra. A GrayLog2 interfésze

A **??** ábrán látható a GrayLog2 webes interfésze, amelynek segítségével a szakemberek könnyen észlelhetik valós időben a fennálló vagy éppen bekövetkező hibákat, továbbá kereshetnek a korábban előforduló hibákban (erre a GrayLog2 az szoftvert használja) illetve könnyen beállíthatnak értesítéseket, hogy akár az éjszaka felmerülő hibákról is értesülhessen a felelős.

3.3. Alkalmazás hiba monitorozás

A naplózással a hibák könnyen észrevehetővé és később visszakereshető válnak, azonban a következőkre nem nyújt megoldást:

- hibák rögzítése a jegykezelő (issue kezelő) rendszerben
- egy hiba csak egyszer kerüljön rögzítésre
- automatikus értesítés a hibáról
- a hibászázalék vizualizálása komponensenként

Természetesen alkalmazás hiba monitorozó rendszer nélkül is működhet rendszer jól, azonban előfordulhat, hogy nagy mennyiségű hibánál a naplógyűjtő rendszerben átsiklanak problémák felett vagy egyszerűen elfelejtésre kerülnek. A másik probléma lehet amikor a hibajelentést közvetlenül a jegykezelő rendszerbe kötik be. Ez miért lehet probléma? Egy nagy felhasználóbázissal rendelkező rendszernél, ha hiba csak a felhasználók néhány százalékánál fordul elő, akkor is szükségtelenül nagy mennyiségű jegy keletkezést fogja előidézni.

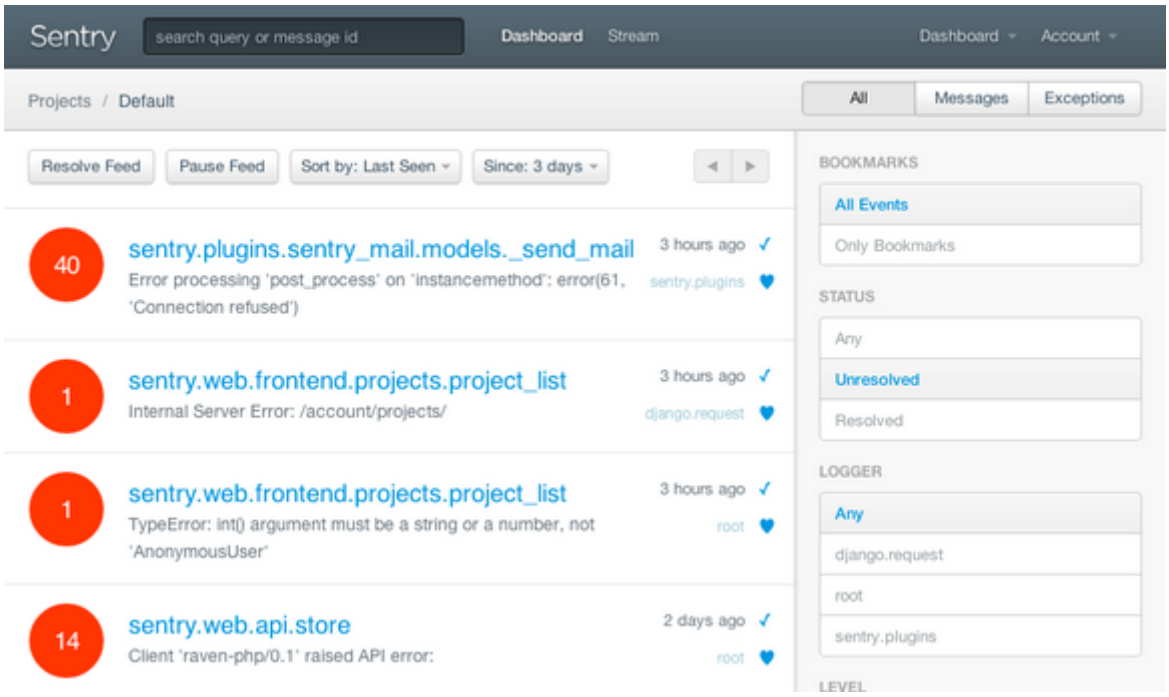
Az alkalmazás hiba monitorozás leginkább a nem webes alkalmazásoknál (mobil és asztali) terjedt el az utóbbi időben, ugyanis ezeknél a rendszereknél a naplók nem, vagy csak nehezen gyűjthetők valós időben.

Elérhető alkalmazások:

- Sentry - `http://getsentry.com`
- Exceptional - `http://www.exceptional.io/`
- Bugsense - `http://www.bugsense.com/`

3.4. Szolgáltatások integrálása

git commit hook jenkins log deploy hibák graylog nagios sentry



3.2. ábra. A Sentry interfésze

3.4.1. Hubot

GitHub, Inc., wrote the first version of Hubot to automate our company chat room. Hubot knew how to deploy the site, automate a lot of tasks, and be a source of fun in the company. Eventually he grew to become a formidable force in GitHub. But he led a private, messy life. So we rewrote him.

Today's version of Hubot is open source, written in CoffeeScript on Node.js, and easily deployed on platforms like Heroku. More importantly, Hubot is a standardized way to share scripts between everyone's robots.

3.5. Webes alkalmazások

A webes alkalmazásoknál a folyamatos integráció több feladatot is ellát:

- tesztek futtatása (unit, tdd, bdd, acceptance, integration)
- a fő verzióba való olvasztás
- adatbázis migrációs fájlok létrehozása
- kliensoldali statikus tartalmak konkatenálása, minimalizálása

A közösségi kódmegosztó github integrációs folyamata teljesen megfelel a felsorolásnak (Douglas, 2012). A különbség mindössze annyi, hogy miután sikeresek a continuous integration lépései, egyből élesítésre kerülnek a szerverekre, azaz a fejlesztők a felelősek, ha valami hibát vétenek egy-egy funkció implementálásában.
Viszont a Facebook a saját PHP-ban írt alkalmazásának performancia javításának céljából további feladatokat is végez a folyamatos integrálás során, ez pedig a buildelés, azaz a fordítás. A buildelés során a Facebook saját kódbázisát transzformálja, amelynek köszönhetően hatszoros gyorsulást sikerült elérniük (a PHP kódot optimalizált C kódra alakítják; az átalakítás során használt szoftver ingyenesen elérhető [`https://github.com/facebook/hiphop-php`]). Paul, 2012

Tesztek futtatása

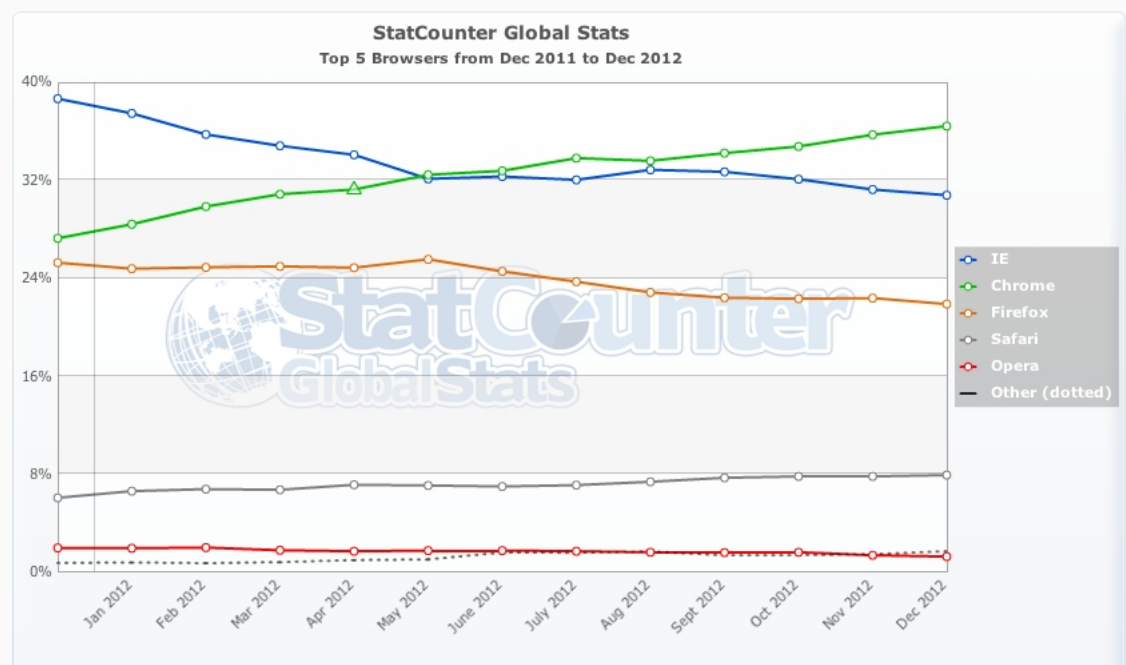
A tesztek futtatása a webes rendszerekben ugyanolyan fontos, mint bármelyik másik platformon. Viszont míg az asztali és mobil alkalmazásoknál, ismerhetőek a kliensek rendszer tulajdonságai (például ha egy alkalmazás csak Windows 7 operációs rendszeren működik, akkor ismerhetőek az azon elérhető futtathatósági lehetőségek, illetve fejlesztői eszköztárak), azonban a webes alkalmazásoknál nem csak különböző operációs rendszerekre kell optimalizálni, hanem az eltérő böngészőkre (melyek a lehető legkülönbözőbb módon implementálták az egyébként is laza HTML és ECMAScript szabványokat) és azok különböző verzióra. Ezeknek az okoknak köszönhetően a tesztelés, illetve a tesztautomatizálás rendkívül fontos a webes alkalmazásoknál (Johansen, 2011b).

Kliensoldali tartalmak

A kliensoldali tartalmak konkatenálása és minimalizálás kiemelt fontosságú a webes alkalmazásoknál, ugyanis ezek a statikus tartalmak - a HTML mellett - az alkalmazás minden betöltésekor letöltésre kerülnek, ami pedig sok különálló fájl esetén a felhasználó élmény rovására mehet.

3.6. Asztali alkalmazások

Az asztali alkalmazások esetén az alkalmazások frissen tartása, frissítése egy bonyolultabb folyamat, mert míg a webes alkalmazásoknál az új verzió elhelyezését, élesítését az alkalmazás fejlesztője - vagy PaaS esetén egy harmadik fél - végzi, addig az asztali alkalmazásoknál, a frissítés végbemenetele a végfelhasználó részéről interakciót igényel.
Az asztali alkalmazások frissítésére egy tökéletes példa a Google Chrome. Ahogyan a 3.3 ábrán látható, a keresőóriás Google böngészője már 2012. decemberében már legelterjedtebb böngésző volt. Azonban a Chrome verzióváltási politikája elég erőteljesen eltér az iparban megszokottól, mert minden hatodik héten új verziót adnak ki (Laforge, 2010) és emellett a szoftver automatikusan ellenőrzi, hogy van-e elérhető frissítés és amennyiben rendelkezésre áll új verzió, akkor automatikusan letöltésre kerül (Kuchhal, 2009).



3.3. ábra. Böngészők piaci részesedése 2011. december és 2012. decembere között (*Böngészők piaci részesedése* 2012)

Azonban más cégek, mint például a github, nem hat hetente ad ki új verziót, hanem a négy hónap alatt huszonöt új verziót tettek elérhetővé (Roben, 2012). Ilyen gyakori kiadási ciklus, csak úgy valósítható meg, ha a legtöbb folyamat automatizálásra kerül (mint ahogy a github automatizálja is), mert ezáltal szinte kizárják az emberi hanyagság okozta hibákat, illetve erőforrást képesek spórolni.

Felhasznált irodalom

Letöltve: 2013. március 2. URL: http://campfirenow.com/.

Letöltve: 2013. március 2. URL: https://grove.io/.

Letöltve: 2013. március 2. URL: https://www.flowdock.com/tour.

Böngészők piaci részesedése (dec. 2012).
Letöltve: 2012. január 2.
URL: http://gs.statcounter.com/#browser-ww-monthly-201112-201212.

Continuous Integration (original version) (2002).
Letöltve: 2012. december 25.
URL: http://martinfowler.com/articles/originalContinuousIntegration.html.

Douglas Jake (aug. 2012). *Deploying at GitHub*.
Letöltve: 2012. november 20.
URL: https://github.com/blog/1241-deploying-at-github.

Johansen Christian (2011a). *Test-Driven JavaScript Development*. Pearson Education, Inc, p. 12.

— (2011b). *Test-Driven JavaScript Development*. Pearson Education, Inc, p. 27.

Kuchhal Rahul (jan. 2009). *Google Chrome Installation and Updates*.
Letöltve: 2012. december 22.
URL: http://blog.chromium.org/2009/01/google-chrome-installation-and-updates.html.

Laforge Anthony (2010). *Release Early, Release Often*.
Letöltve: 2012. március 1.
URL: http://blog.chromium.org/2010/07/release-early-release-often.html.

Paul Ryan (árp. 2012). *Exclusive: a behind-the-scenes look at Facebook release engineering.*

Letöltve: 2012. december 27.

URL: `http://arstechnica.com/business/2012/04/exclusive-a-behind-the-scenes-look-at-facebook-release-engineering/3/`.

Roben Adam (szept. 2012). *How we ship GitHub for Windows.*

Letöltve: 2012. november 30.

URL: `https://github.com/blog/1271-how-we-ship-github-for-windows`.

See how HipChat stacks up against the competition.

Letöltve: 2013. március 2. URL: `https://www.hipchat.com/compare`.