AZ AUTOMATIZÁLT STRESSZTESZTELÉS – ESETTANULMÁNY

Béli Marcell – Hoffer Tibor – Tóth Fanni

Absztrakt: A tanulmányban azt vizsgáljuk, hogy egy automatizált szoftvertesztelő rendszer megoldást tud-e nyújtani a szoftver funkcionális tesztelése mellett a hatékony és pontos terheléses tesztelésre. A stressztesztet gyakorlati példán, valós üzleti problémán keresztül mutatjuk be: az FX Software Zrt. által fejlesztett InFoRex rendszert vetjük terheléses teszt alá. A stresszteszt futtatásának köszönhetően egy tisztább képet kaptunk a rendszer határaival kapcsolatban. A teszt eredményei szerint garantálható a rendszer megbízhatósága kritikus helyzetekben is, míg az eredmények tárolására készített szoftver segítségével korán észrevehetővé tehető, ha rossz irányba indul el egy fejlesztés. Rávilágítunk, hogy drasztikusan kisebb erőforrásigény mellett sokkal nagyobb terhelés alá tudunk vetni egy-egy szoftvert automatizáltan, de ehhez azonban továbbra is szükség van emberi kontrollra. Összességében a stresszteszt automatizálása hozzásegít egy céget egy minőségi és stabil szoftver fejlesztéséhez, mindezt alacsony erőforrás felhasználás mellett.

Abstract: The aim of our study is to investigate whether an automated software testing system can provide a solution for efficient and accurate stress testing in addition to functional software testing. The stress testing is presented through a real business problem in a case study: the InFoRex system developed by FX Software PLC. is stress tested. Due to running the test, a clearer picture of the system's limits is obtained. The test results show that the reliability of the system can be guaranteed even in critical situations. Software designed to store the results can give an early warning if a development is going in the wrong direction. We point out that we can put a much higher load on software automatically with drastically lower resource requirements, but this still requires human control. Overall, stress test automation helps a company to develop high quality and stable software with low resource consumption.

Kulcsszavak: szoftvertesztelés, automatizálás, esettanulmány

Keywords: software testing, automation, case study

1. Bevezetés

Mint manapság minden területen, a szoftverek minőségbiztosításában is egyre nagyobb teret kap a folyamatok automatizálása. A manuális tesztelés kiváltásával időt és költséget lehet megspórolni, mindamellett, hogy az emberi hibafaktort minimalizáljuk.

A szoftvereket fejlesztő cégek számára fontos, hogy az általuk kiadott termékek minősége a lehető legjobb legyen, viszont ez nem merülhet ki abban, hogy funkcionálisan hibátlan legyen. A felgyorsult világ hatására egyre nagyobb terhelés alá vannak véve a fejlesztett rendszerek, így a fejlesztő cégek számára fontossá vált, hogy ismerjék termékük terhelhetőségének határait. Ennek hatására egyre nagyobb szerepet kapnak a stressztesztek és performancia tesztelések. Felmerül a kérdés, hogy ezeket a teszteket lehetséges-e automatizálni?

Egy komplex rendszer teljesítményét és terhelhetőségének határait manuálisan nagyon körülményesen, pontatlanul és nagyon költségesen lehet megvalósítani. Először is elő kell készülni: meg kell találni az emberi erőforrásokat, akik a feladatot el tudják végezni. Biztosítani kell ezen felül, hogy hozzáférjenek azokhoz a funkciókhoz, amelyeket tesztelni szükséges és létre kell hozni a tesztkörnyezetet.

ISSN 1788-7593 (Nyomtatott) DOI: https://doi.org/10.14232/jtgf.2024.1-2.11-27
ISSN 2676-9867 (Online)

Második pontként össze kell hangolni a tesztelőket, hiszen fontos, hogy az akciók, amelyeket végrehajtanak, egy időben történjenek annak érdekében, hogy a rendszer a lehető legnagyobb terhelést kapja.

Béli et al. (2022) korábban megvizsgálta, hogy a funkcionális tesztelés milyen esetben automatizálható és milyen esetben célszerű a manuális tesztelést folytatni. Ebben a cikkben azt vizsgáljuk, hogy az automatizált szoftvertesztelő rendszer, a TestComplete megoldást tud-e nyújtani a szoftver funkcionális tesztelése mellett a hatékony és pontos terheléses tesztelésre.

A terheléses tesztet gyakorlati példán, valós üzleti problémán keresztül mutatjuk be: az FX Software Zrt. által fejlesztett InFoRex rendszert vetjük terheléses teszt alá és az esettanulmányon keresztül bemutatjuk eredményeink.

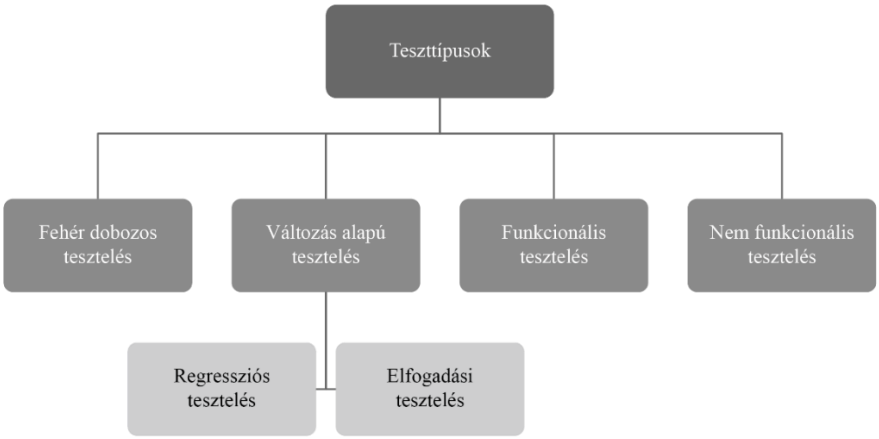
Az InFoRex integrált treasury rendszer a klasszikus front – mid – és backoffice feladatok összehangolt és automatizált kezelésére tervezett komplex szoftver, amely teljes ügyletkezelést biztosít banki és vállalati ügyfelei számára. A felhasználók számossága miatt releváns üzleti probléma a stressztesztelés kivitelezése és lényeges megállapításokat tehetünk, valamint következtetéseket vonhatunk le a téma körüljárását követően.

2. Áttekintés

A szoftvertesztelés normalizálásának céljából jött létre 2002-ben az International Software Testing Qualifications Board (ISTQB) nevű nonprofit szervezet, amely a tesztelők számára tesz elérhetővé szabványok alapján különböző minősítéseket. A testület kizárólagos magyarországi képviselete a Magyar Szoftvertesztelői Tanács (HTB). A szoftvertesztelés általuk hivatalosan használt definíciója: „Folyamat, mely az összes szoftverfejlesztési életciklushoz kapcsolódó akár statikus, akár dinamikus tevékenységekből áll, amelyek kapcsolatban vannak egy komponens vagy rendszer, illetve a hozzá kapcsolódó munkatermékek tervezésével, előkészítésével és kiértékelésével. Célja, hogy megállapítsa, a szoftvertermék teljesíti-e a meghatározott követelményeket, megfelel-e a célnak és megtalálja a hibákat.” (HTB, 2023)

A szoftvertesztelés napjainkban egyre nagyobb hangsúlyt kap. Catelani et al. (2011) szerint a tesztelés és a termékfejlesztés költsége csaknem egyenértékű. Diaz et al. (2003) azonban rávilágított, hogy a tesztelés költsége automatizálással csökkenthető. Ahhoz, hogy megértsük, hogy a stressztesztelés, amelynek automatizálása tanulmány célja, hol helyezkedik el a teszttípusok közt, bemutatjuk a szoftvertesztelés négy teszttípusát ISTQB (2021) alapján (*1. ábra*).

1. ábra: **Teszttípusok**



Forrás: saját szerkesztés.

A teszttípus olyan tesztevékenységek csoportja, amelyek meghatározott tesztcélokon alapulnak, és amelyek egy komponens vagy rendszer meghatározott jellemzőit célozzák meg.

Fontos kiemelni, hogy az egyes teszttípusok nem függetlenek egymástól. Lehetnek olyan helyzetek, amikor keverve, egyidejűleg alkalmazzuk a különböző megközelítéseket.

Ilyen eset lehet például, amikor úgy írunk meg funkcionális tesztesetet, hogy figyelembe vesszük a szoftver forráskódját, vagy amikor biztonsági hibát tesztelünk vissza.

A fehér dobozos tesztelés a belső struktúra alapján megy végbe. A különböző tesztelési szinteken különböző fajta struktúra alapján kerül levezetésre egy-egy teszteset. Az alsóbb szinteken, komponens és integrációs teszt esetén, a szoftver forráskódja és a szerkezete alapján, míg a felsőbb szinteken az üzleti logika vagy a menü rendszer felépítése alapján történik a tesztelés.

A változás alapú tesztelés, a nevéből is következtethetően, akkor megy végbe egy szoftver életciklusában amikor valamilyen módosítás történik. Ennek kiváltó okai alapján két csoportra lehet bontani. Az egyik csoport a megerősítő tesztelés. Ez a tesztelés a korábban feltárt és bejelentett hibának a javítás utáni vizsgálatát takarja. A tesztelés célja viszonylag egyszerű, kideríteni, hogy a hiba teljesen javult-e vagy a javítás nem volt megfelelő és még fennáll a probléma. A másik csoport a regressziós tesztelés. A tesztelés célja, hogy feltárja az új funkció implementálása vagy a hibajavítások után, a rendszer nem módosított moduljaiban keletkező új, vagy, a változás hatására előtérbe került, eddig rejtett hibákat.

Összességében a megerősítő tesztelés egy bizonyos hibára fókuszál, így a tesztelés végrehajtásához pontos lépéseket tartalmazó dokumentumokat nem minden esetben lehet találni. Ilyen esetben a legcélravezetőbb, hogy ha a hiba leírásában megfogalmazott lépéseket követjük, ami a hiba bekövetkezte előtt történt. Ezeket érdemes akár teszteset leírásként rögzíteni és a későbbi regressziós tesztelések során végrehajtani.

A funkcionális teszteléssel a rendszer viselkedését vesszük górcső alá. Ez a tesztelés biztosítja, hogy a fejlesztett szoftver moduljai megfelelően működnek.

A tesztelés segít megbizonyosodni arról, hogy a rendszer azt teszi, amit a fejlesztés elején a felhasználói követelményekben és a rendszer specifikációjában megfogalmaztunk. A manuális és az automatizált tesztelők számára készülnek teszteset leírások az egyes funkciók teszteléséhez. Fekete dobozos tesztelési technikákkal viszik végbe a tesztelési folyamatokat, mivel a funkciók teszteléséhez nem fontos ismerni a szoftver forráskódját. Ez a fajta tesztelés az összes tesztelési szinten meg kell történjen, annak érdekében, hogy a rendszer megfelelően funkcionáljon. Funkcionális tesztelés automatizálásának bemutatásával foglalkozik Béli et al. (2022). Kiderült, többszöri teszt esetén a tesztelés automatizálása hatékony, míg egyedi teszteknél érdemes továbbra is manuálisan tesztelni. Ebben a cikkben viszont a funkcionális teszt helyett a nem funkcionális tesztelési csoportra fókuszálunk.

A funkcionális teszteléssel ellentétben, a nem funkcionális tesztelés nem azt nézi, hogy a szoftvernek hogyan kellene működnie, hanem hogy egyáltalán működik-e különböző körülmények között.

Ezzel a teszteléssel azt vizsgáljuk, hogy a rendszer megfelel-e a nem funkcionális követelményeknek is. Az összes tesztelési szinten elvégezhetőek ezek a tesztek. A fekete dobozos tesztelési technikákat alkalmazzuk ehhez.

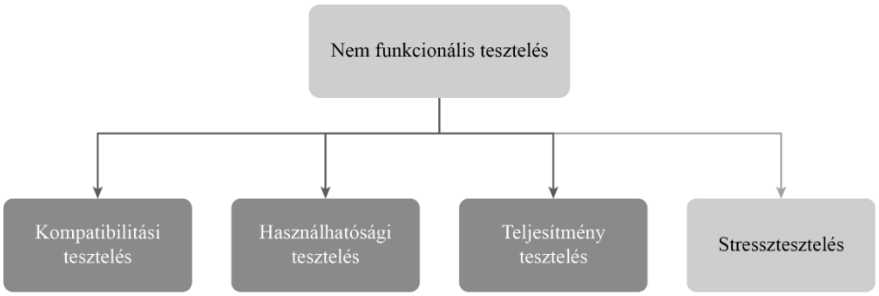
A nem funkcionális teszteléssel a szoftver olyan attribútumait teszteljük, amelyek nem tekinthetőek funkciónak, mint például:

- Teljesítmény
- Kompatibilitás
- Használhatóság
- Megbízhatóság
- Biztonság
- Karbantarthatóság
- Hordozhatóság

Összességében következtethetünk arra, hogy a funkcionalitás mellett sok olyan tulajdonsága van egy szoftvernek, ami befolyásolhatja a felhasználók által alkotott véleményt.

A sok nem funkcionális tulajdonság miatt rengeteg különféle ezekhez tartozó tesztelési típus létezik. Négy tesztelési típust mutatunk be részletesebben (*2. ábra*). A kompatibilitási, használhatósági és teljesítmény tesztelés mellett a stressztesztelést részletezzük, amely a gyakorlati rész megalapozására is szolgál.

2. ábra: **Nem funkcionális tesztelési típusok**



Forrás: saját szerkesztés.

A kompatibilitási tesztelés során a szoftvert egy másik környezetben teszteljük, ellenőrizzük a működését és viselkedését. A másik környezet takarhat hardverváltoztatást, hálózatváltozást, különféle kiszolgálókat és ezeknek kombinációját.

A hardveres változtatás első olvasásra ijesztőnek és költségesnek tűnhet, viszont az elmúlt pár évben már nagyon megbízható és hatékony virtualizációs programok kerültek a piacra, amelyekkel könnyedén lehet létrehozni különféle hardverrel környezetet a tesztelendő szoftver számára. Ennek köszönhetően vizsgálhatóvá válik, hogy a fejlesztett termék minden olyan hardveren képes ellátni funkcióját, amilyenekkel majd a későbbiekben esetleg használni fogják.

A hálózati változtatás alatt érthetjük az internetkapcsolat sebességét és a kapcsolat típusát is. ebben az esetben is megkönnyíti a tesztelést a különböző programok használata, hiszen képesek vagyunk használatukkal az internet sebességét korlátozni. Ezáltal pedig vizsgálhatóvá válik, hogy mi az a minimum hálózati kapcsolati sebesség, amely szükséges a szoftver használatához.

A használhatósági tesztelés során a tesztelők tanulmányozzák a végfelhasználók által végzett tesztelést és felhasználási módot és keresik azokat a pontokat, ahol javítani tudnának a felhasználói élményen a rendszer újabb verzióival. Ezek a tapasztalatok vezethetnek felületi átalakításokhoz, teljesítmény javításhoz, de akár teljesen új funkciók implementálásához is. Az ilyen tesztelés leghatékonyabb módja, ha manuálisan történik, mivel felmerülhetnek kérdések a tesztelés során.

Teljesítménytesztelés során általában normál, vagy kismértékben megnövelt terhelés mellett vizsgáljuk, hogy a rendszer megfelel-e az előre definiált teljesítménykövetelményeknek. Vizsgálandó többek között az indítástól a teljes betöltésig tartó idő és az egyes funkciók teljesítésére fordított idő. Ezeknek figyelése és tesztelése nem csak a felhasználói élmény miatt fontos, az értékek esetleges változása utalhat valamilyen rendszerhibára is, amely funkcionálisan még nem gátolja a működést, de a későbbiekben problémát okozhat.

A stressztesztelés során a rendszert leterhelik, hogy kiderüljön, mennyi terhelést vagy maximális forgalmat képes kezelni a teljesítmény romlása nélkül. (ProofIT, 2023). Egyszerre többször kerülhet ugyanaz a funkció használatba, amivel rengeteg olyan hiba jöhet elő, amely normál vagy kicsit megnövelt használat mellett még nem kerül a látótérbe. Ilyen lehet például lekérdezés vagy adatrögzítés adatbázisba.

Mivel ebben az esetben nagyon magas terhelésről beszélünk, így a végrehajtásához sok tesztelőre lenne szükség, ez pedig rengeteg költséggel jár. Az erőforrásigény mellett fontos a tesztelést végzők összehangoltsága, hogy az akciók lehetőleg egyidőben történjenek, ennek kivitelezése viszont nehézségekbe ütközhet. A stressztesztelés hatékony és pontos végrehajtása manuálisan nem kivitelezhető, ennek automatizálása kardinális fontosságú.

3. Esettanulmány – Automatizált stressztesz t

A felvezetést követően, tekintsük át, hogy hogyan lehet megvalósítani a stressztelt elést automatizáltan az FX Software Zrt. InFoRex rendszerén keresztül! Az InFoRex integrált treasury rendszer a klasszikus front – mid – és backoffice feladatok összehangolt és automatizált kezelésére tervezett komplex szoftver, amely teljes ügyletkezelést biztosít banki és vállalati ügyfelei számára. A felhasználók számossága miatt releváns üzleti probléma a stressztesztelés kivitelezése és lényeges megállapításokat tehetünk, valamint következtetéseket vonhatunk le a téma körüljárását követően. A stressztesztelés kivitelezéséhez a TestComplete automatizált tesztelési környezetet használjuk, amely egy olyan automatizált tesztelési környezet, amely lehetőséget nyújt az asztali, mobil és webes alkalmazások tesztelésére. A TestComplete-ről részletesen ír Béli et al. (2022).

A stressztesz t gyakorlatilag kivitelezhetetlen manuálisan, viszont még nem esett szó arról, hogy ez miért fontos a szoftverfejlesztő cégeknek és a cég ügyfeleinek.

A végfelhasználók számára nemcsak az fontos, hogy funkcionálisan hibátlan legyen a rendszer, hanem az is, hogy megbízható és stabil legyen. Rengeteg olyan helyzet történhet, ahol a rendszer megnövekedett terhelést kap és ilyenkor kritikus szempont, hogy ne omoljon össze a szoftver. Amennyiben egy ilyen helyzetben megállna a rendszer, az presztízs veszteséghez vezetne a fejlesztő cég számára, ami nemcsak a jelenlegi ügyfeleivel szemben lenne negatív hatással, hanem a jövőbeni potenciális megrendelőivel szemben is.

A szoftver határainak ismerete olyan szempontból is fontos, hogy olyan mértékben már ne értékesítse a szoftvert, amelyet már nem képes kiszolgálni, függetlenül attól, hogy milyen hardver van használatban a futtató környezetben. Ezért fontos, hogy a stressztesztelés pontos és hatékony megvalósítására keressen minden szoftverfejlesztő vállalat megoldást. A gyakorlati részben az eddig ismertetett tesztelési megoldásokat felhasználva bemutatjuk a stressztesztelés automatizálását.

3.1. Előkészületek

A stressztesztelés hatékony kivitelezéséhez fontos, hogy többen egyszerre végezzenek valamilyen akciót a tesztelt szoftveren, így az első lépés, hogy megkeressük azt a funkciót, ami erre alkalmas a TestComplete-ben elérhetőek közül.

3.1.1. Network Suit

A TestComplete erre alkalmas funkciója a Network Suit néven található meg. Ebben lehetőség nyílik több különböző gép (Hosts) összehangolására. A szinkronizációs

pontokat (Synchronization points) is itt lehet kezelni, újakat hozzáadni vagy a már nem kellőket törölni.

Továbbá lehetőség nyílik értékek átvitelére a gépek között, akár ellenőrzés céljából, akár annak érdekében, hogy azt az értéket, amit az egyik tesztre használt gépen kaptunk a szoftvertől, felhasználjuk a többi gépen más célból.

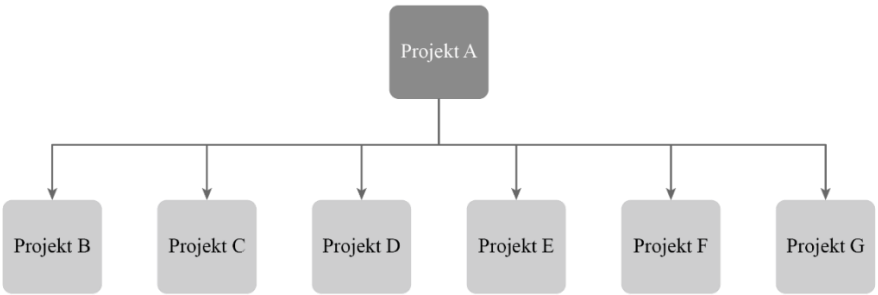
A munka (Jobs) menüpontban vannak összegyűjtve azok a feladatok (Tasks) amelyek a teszt futása alatt meghatározzák, hogy melyik projekt melyik gépen fusson.

3.1.2. Projektek előkészítése

Az összehangolt tesztelést a TestComplete úgy kezeli, hogy egy főprojekt összefogja a futásra használt alprojekteket. A projektek a már megírt funkcionális teszt szkripteket tartalmazzák.

Esetünkben ez hat különálló projektet fog jelenteni, mivel hat virtuális gép lesz alkalmazva a stressztesztelés végrehajtásához. Ennek vizualizálására szolgál a *3. ábra*. Ezzel lehetőség nyílik arra, hogy minden gépen más-más lépések menjenek végbe.

3. ábra: **Projekt ágazati ábra**



Forrás: saját szerkesztés.

3.1.3. Virtuális gépek előkészítése

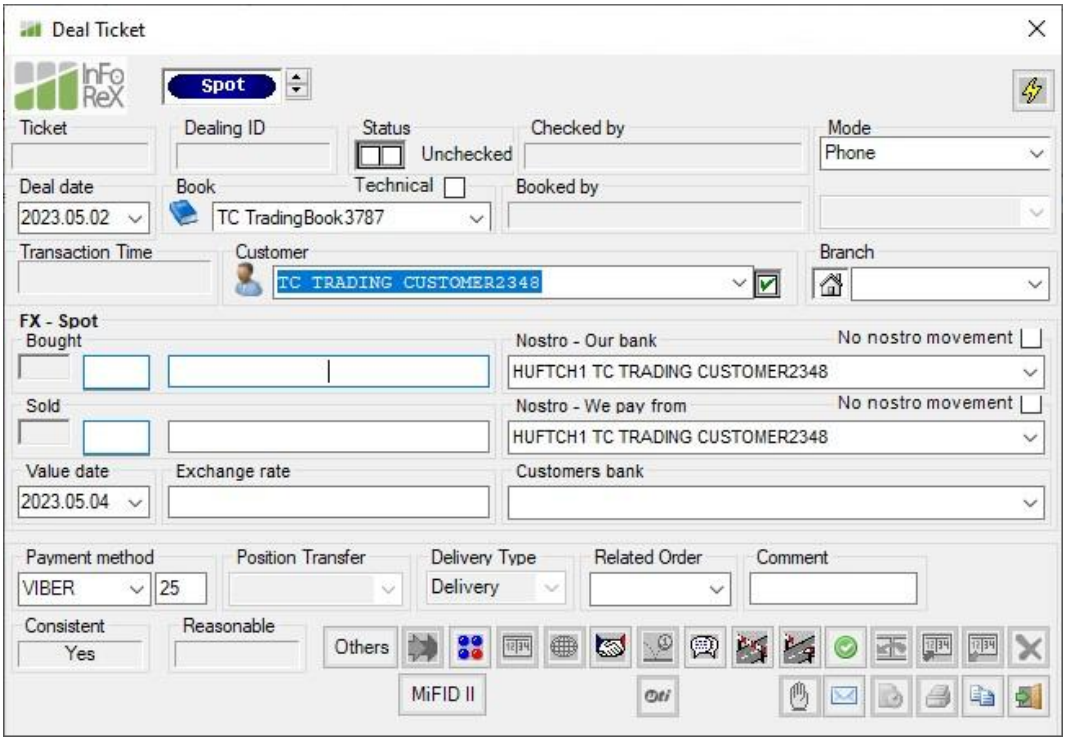
A virtuális gépeket a szoftver követelményei szerint kell létrehozni. Esetünkben ez Windows 10 operációs rendszert futtató gépekből fog állni. A gépeket különböző erősségű hardverrel lehet ellátni, de ezt jelen esetben nem alkalmazzuk, így a tesztfutásra ugyanolyan virtuális számítógépeket fogunk használni.

A tesztgépekre feltelepítjük a TestExecute programot, amely lokálisan felel a teszt futtatására az egyes gépeken. Ezután a TestComplete Hosts menüpontjában hozzáadjuk a létrehozott virtuális gépeket.

3.1.4. Tesztelt InFoRex funkció

Az InFoRex egyik fő funkcióját vetjük stressztesztalá: ez pedig az ügyletkötés (*4. ábra*). Ezen felül még teszteljük külön a szoftver azon funkcióját, amikor az adatbázisból adatot tölt be a felületre.

4. ábra: **InFoRex ügyletkötési ablak**



Forrás: saját szerkesztés.

3.1.5. Szükséges adatok rögzítése és tárolása

Az ügyletek sikeres megkötéséhez szükséges adatokat létre kell hozni a tesztelt verzióban mielőtt elkezdhetnénk a stressztesztelést. Ezeket az adatokat a legcélszerűbb, ha a főprojekt segítségével hozzuk létre.

A tesztesetek lefuttatása után a rögzített adatokat át kell adni a Network Suit projekt változóiba, annak érdekében, hogy az alprojektek használni tudják azokat a tesztelés során. Ezek után át lehet térni a megvalósítás folyamatára.

3.2. Stressztesztelés megvalósítása

Az első lépés a megvalósításban, hogy megtaláljuk azokat az akciókat a folyamatban, amelyek párhuzamos futását szükséges tesztelni. Ezek általában olyan szinkronizációs pontok, ahol adatot hív le az adatbázisból vagy rögzít az adatbázisba a rendszer. Ezeket a szinkronizációs pontokat a teszt szkriptekben a logikailag megfelelő helyeken kell elhelyezni, például az ügyletek elfogadásai teljesen egyszerre történjenek, úgy, hogy már minden kliens gépen megtörtént az ügyletkötési ablak mezőinek teljes körű kitöltése.

Ezután ezekből az építőkövekből felépítjük a teljes stressztesztünket.

3.2.1. InFoRex-ek indítása

Az InFoRex ügyletkötésekhez különböző (pénzügyi területen használatos) jogkörrel rendelkező felhasználókra van szükség. A funkcionális tesztelés során ezt úgy

kezeljük, hogy öt darab InFoRex-et indítunk, különböző felhasználókkal, a jogkörök alapján, amelyek az alábbiak:

- Dealer.
- Backoffice.
- Treasurer.
- Riskman.
- InfAdmin.

A stressztesztelés elvégzésekor ugyanezt a módszert alkalmazzuk. Az első szinkronizációs pont tehát: megvárjuk, amíg mindegyik gépen betölt az összes elindított verzió (*5. ábra*).

5. ábra: **InFoRex indítás szinkronizálás**

```
43      aqPerformance.Start("StartupTime", false);
44
45      TestedApps.Items(InFoRexDealer).RunAs("FXDOMAIN", "fxduser", "TestPass1234", "", "", "1", true);
46      WaitFor(1);
47
48      Delay(5000);
49      aqPerformance.Start("StartupTime", false);
50
51      TestedApps.Items(InFoRexBO).RunAs("FXDOMAIN", "fxbouser", "TestPass1234", "", "", "1", true);
52      WaitFor(2);
53
54      Delay(5000);
55      aqPerformance.Start("StartupTime", false);
56
57      TestedApps.Items(InFoRexTreasurer).RunAs("FXDOMAIN", "fxtruser", "TestPass1234", "", "", "1", true);
58      WaitFor(3);
59
60      Delay(5000);
61      aqPerformance.Start("StartupTime", false);
62
63      TestedApps.Items(InFoRexRiskman).RunAs("FXDOMAIN", "fxrmanuser", "TestPass1234", "", "", "1", true);
64      WaitFor(4);
65
66      Delay(5000);
67      aqPerformance.Start("StartupTime", false);
68
69      TestedApps.Items(InFoRexSA).RunAs("FXDOMAIN", "fxinfauser", "TestPass1234", "", "", "1", true);
70      WaitFor(5);
71
72      if (Project.Variables.OurDealingCode == "xxxx")
73      {
74        Delay(5000);
75        aqPerformance.Start("StartupTime", false);
76
77        TestedApps.Items(InFoRexMO).RunAs("FXDOMAIN", "fxmouser", "TestPass1234", "", "", "1", true);
78        WaitFor(6);
79      }
80    }
81    else
82    {
83      TestedApps.Items(InFoRexDealer).Run(5);
84    }
85
86     NetworkSuite.SynchPoints.InFoRexStartUp.WaitFor();
87
88    Delay(500);
89  }
```

Forrás: saját szerkesztés.

3.2.2. Adatbetöltés stressztesztelés

Az InFoRex-nek számos olyan pontja van, ahol nagy mennyiségű adatot kérdez le az adatbázisból. A szoftver használata közben kialakulhat olyan helyzet, hogy egyszerre többen kérnének le adatokat az adatbázisból, vagy akár olyan is, hogy egy felületre egyszerre többen frissítenek rá egy időben. Annak érdekében, hogy garantálni tudjuk, hogy ilyen esetben sem áll le a rendszer és a kért adatok betöltődnek, célszerű ezt a funkciót is nagy terhelés alá vetni.

A szinkronizációs pontokat a frissítés (Refresh) gomb megnyomása elé kell rakni. (*6. ábra*) Mivel ezeket a lépéseket a SmokeTest alatt menüpontonként csoportosítva hajtjuk végre a funkcionális tesztelés során, ezért ebben az esetben ajánlott új szkripteket írni a meglévők módosítása helyett.

6. ábra: **EMIR képernyő frissítése**

```
//USEUNIT FxCore


function EmirRiport()
{
  let inFoRex = Aliases.InFoRex;
  let frmMain = inFoRex.frmMain;
  frmMain.StripMainMenu.Click("Backoffice|EU Reporting|Emir|EMIR Reporting");
  inFoRex.frmEmirMain.grpDatePicker.dsReportDate.wDate = "2022-04-04";
  //creat Reported trade report
  inFoRex.frmEmirMain.cmbReportType.SelectorPanel.cmbInput.ClickItem("Reported trade");
  inFoRex.frmEmirMain.btnCreateReport.ClickButton();

  // Synchpont
  NetworkSuite.SynchPoints.Refresh.WaitFor();
  RefreshForm(inFoRex.frmEmirReportGenerator,inFoRex.frmEmirReportGenerator.myFlowLayoutPanel1.grpAutoLoad.btnAutoLoad);
  inFoRex.frmEmirReportGenerator.SetFocus();
  inFoRex.frmEmirReportGenerator.Close();

  //creat Valuation update report
  inFoRex.frmEmirMain.cmbReportType.SelectorPanel.cmbInput.ClickItem("Valuation update");
  inFoRex.frmEmirMain.btnCreateReport.ClickButton();

  // Synchpont
  NetworkSuite.SynchPoints.Refresh.WaitFor();
  RefreshGrid(inFoRex.frmEmirReportGenerator,inFoRex.frmEmirReportGenerator.splitContainer1.SplitterPanel.grdReportItems,inFoRex.frmEmirReportGenerator.myFlowLayoutPanel1.grpAutoLoad.btnAutoLoad);
  inFoRex.frmEmirReportGenerator.SetFocus();
  inFoRex.frmEmirReportGenerator.Close();
  inFoRex.frmEmirMain.Close();
  inFoRex.dlgInFoRex.btnNem.ClickButton();
}

module.exports.EmirRiport = EmirRiport;
```

Forrás: saját szerkesztés.

Amint a tesztelni kívánt menüpontokra elkészültek a tesztesetek, az utolsó lépés következik a teszt futtatása előtt: a teszteket be kell állítani az alprojektekben, hogy lefussanak (*7. ábra*). Több variáció is lehetséges: a hat virtuális gépen egyidőben ugyanarra a menüpontra történhet a frissítés, vagy mindegyik gépen különböző menüpontokra történhet az akció. Erre két megoldás lehetséges. Ebben az esetben külön tesztesetként fogjuk rögzíteni az egyes szkripteket. A tesztesetek sorrendje fogja megadni fentről lefelé, hogy milyen sorrendben fognak lefutni.

7. ábra: **Tesztesetek**

Stress_Refresh_Check						
EmirRiport	Script\EmirRiport - EmirRiport	[none]	1	0 Continue running	0 Stop current item
GlBookingBatches	Script\GlBookingBatches - GlBookingBatches	[none]	1	0 Continue running	0 Stop current item
MifidARM	Script\MifidARM - MifidARM	[none]	1	0 Continue running	0 Stop current item
MMPerformance	Script\MMPerformance - MMPerformance	[none]	1	0 Continue running	0 Stop current item
NostroBalance	Script\NostroBalance - NostroBalance	[none]	1	0 Continue running	0 Stop current item
Performancebybook	Script\Performancebybook - Performanc...	[none]	1	0 Continue running	0 Stop current item
PositionPortfolio	Script\PositionPortfolio - PositionPortfolio	[none]	1	0 Continue running	0 Stop current item

Forrás: saját szerkesztés.

3.2.3. Ügyletkötések stressztesztelése

Az előző megoldáshoz képest az ügyletkötés tesztelésére írt szkriptek sokkal összetettebbek a sok adatbevitel miatt. A funkcionális tesztelésben használt teszteseteket érdemes tehát módosítani és nem újraírni. Ezzel sok időt nyerhető, mivel a funkcionális tesztelés alatt használt adatok újrafelhasználhatók.

Az egyes ügyleteknek öt különböző státusza lehet. Ezek a tesztelés során fontos szerepet kapnak, mivel a státusz alapján különböző akciókat lehet végrehajtani az ügyleten. Ügylettípusonként ezek nem változnak. Az ellenőrizetlen (Unchecked) státusz azt jelzi, amikor elkezdjük rögzíteni az ügyletet. Ezután kerül ellenőrzött (Checked) státuszba. Ezt a tevékenységet a Dealer jogkörrel rendelkező hajthatja végre. A következő státusz a könyvelt (Booked), amelyet Backoffice jogkörrel rendelkező hajthat végre. A másik két státusz az elutasított (Refused) és törölve

(Deleted). Az elutasított státuszba ellenőrzött és könyvelt ügylet is kerülhet, míg törölni csak elutasított ügyletet lehet.

Mindegyik esetben történik kommunikáció az adatbázissal. Ennek ismeretében már látható, hogy sok kombinációt kell alkalmazni annak érdekében, hogy a lehető legnagyobb arányban kiszűrhetők legyenek a lehetséges hibák.

A szinkronizációs pontok itt is egyértelműen megtalálhatóak. Azok elé a gombnyomások elé kell ezeket a pontokat rakni, amelyek a státusz változáshoz vezetnek az ügyletben (*8. ábra*). A funkcionális tesztek átláthatósága miatt ez viszonylag egy könnyebb feladat, mert minden ilyen akció külön metódusban szerepel. A pontok sikeres rögzítése után következik a tesztlépések gépenkénti összeállítása.

8. ábra: **Ügylet szinkronizációs pont**

```
inFoRex.frmDealCheck.frmAction.frmComment.frmComment_B.Click();
//Save the current time in project variable
Project.Variables.CurrentDateAndTime= aqDateTime.Now();
var Comment = Project.Variables.CurrentDateAndTime + "PC5";
this.Comment = Project.Variables.CurrentDateAndTime + "PC5";

//Enter the date in comment
inFoRex.frmDealCheck.frmAction.frmComment.frmComment_B.txtComment.SetText(Comment);

//Nostros
var Nostro1 = inFoRex.frmDealCheck.frmFX.frmFXI_Nostro.frmFXI_Nostro_B.cmbFXI_Nostro.wText;
this.Nostro1 = checknostro(Nostro1,this.Currency1);
var Nostro2 = inFoRex.frmDealCheck.frmFX.frmFXO_Nostro.frmFXO_Nostro_B.cmbFXO_Nostro.wText;
this.Nostro2 = checknostro(Nostro2,this.Currency2);

//Validation
var BookValue = inFoRex.frmDealCheck.frmDeal_B.cmbDesk.SelectorPanel.cmbInput.wText;
ValidateBook(this.BookName, BookValue);

var CustomerValue = inFoRex.frmDealCheck.frmDeal_B.cmbCustomer.frmCustomer.frmCustomer_B.cmbCustomer.wText;
ValidateCustomer(this.Customer, CustomerValue);

var DealDateValue = inFoRex.frmDealCheck.frmDeal_B.frmDealDate.frmDealDate_B.dtpDealDate.wDate;
ValidateDate(this.DealDate, DealDateValue, "DealDate");

var ValueDateValue = inFoRex.frmDealCheck.frmFX.frmFX1_B.frmFXValueDate.dtpFXValueDate.wDate;
ValidateDate(this.ValueDate, ValueDateValue, "ValueDate");

//Check the deal

//Synchpoint
NetworkSuite.SynchPoints.DealCreate.WaitFor();

frmDealCheck.frmAction.cmdCheck.ClickButton();

inFoRex.frmMain.Activate();

if(frmDealCheck.Exists)
{
   frmDealCheck.Close();
}
```

Forrás: saját szerkesztés.

A teszt elején érdemes az azonos ügyletek azonos akcióinak összehangolása. Ezután az ügylettípusok különböző kombinációit, majd az akciók vegyítése következik. A teszt utolsó harmadában már kombinálhatjuk az akciókat és a típusokat is együtt, így a teszt végére érve szinte minden lehetséges kombináció tesztelhető.

Egy-egy kombinációt többször érdemes lefuttatni, ezzel is növelve a lehetséges hibák feltárásának esélyét. Továbbá statisztikai adatok szerzésére is alkalom nyílik: az egyes akciók igénybe vett idejét lehet mérni.

A kódrészletben látszik (*9. ábra*), hogy a korábban megírt és módosított metódusokat hívjuk meg a tesztesetben, nem olyan módon, mint a frissítések esetén, ahol minden akció külön tesztlépésként szerepel a végrehajtási tervben.

9. ábra: **Kódrészlet**

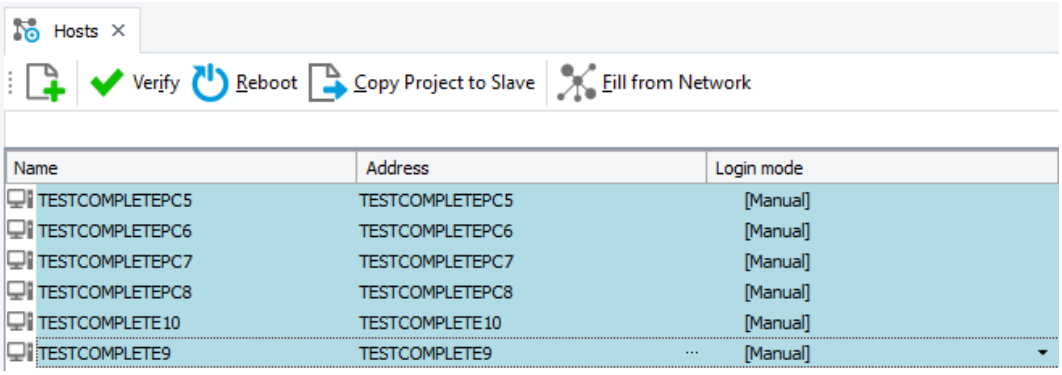
```
function DealCreatStress()
{

  //Spot
    for (let i = 0; i < 30; i++)
    {
      let SpotDeal = new Spot.Instance("default");
      SpotDeal.Create();
      SpotDeal.Book();
      SpotDeal.Refuse();
      SpotDeal.Delete();


  //Outright
      let OutrightDeal = new Outright.Instance("default");
      OutrightDeal.Create();
      OutrightDeal.Book();
      OutrightDeal.Refuse();
      OutrightDeal.Delete();

  //BuyAndSell
      let BuyAndSellDeal = new BuyAndSell.Instance("default");
      BuyAndSellDeal.Create();
      BuyAndSellDeal.Book();
      BuyAndSellDeal.Refuse();
      BuyAndSellDeal.Delete();

 //Lend
      let LendDeal = new Lend.Instance("default");
      LendDeal.Create();
      LendDeal.Book();
      LendDeal.Refuse();
      LendDeal.Delete();

  //FreeBuy
      let FreeBuyDeal = new FreeBuy.Instance("default");
      FreeBuyDeal.Create();
      FreeBuyDeal.Book();
      FreeBuyDeal.Refuse();
      FreeBuyDeal.Delete();


  //Repo
      let RepoDeal = new Repo.Instance("default");
      RepoDeal.Create();
      RepoDeal.Book();
      RepoDeal.Refuse();
      RepoDeal.Delete();
    }
}
```

Forrás: saját szerkesztés.

3.3. A stresszteszt futtatása

Az indítás előtt meg kell adni, hogy melyik számítógép melyik projektet használja a teszt futása során (*10. ábra*).

10. ábra: **Virtuális gépek kezelése**



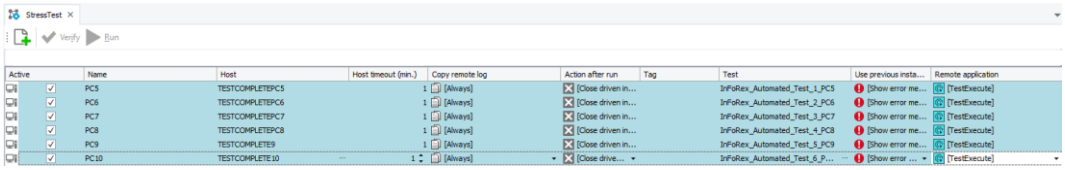
Forrás: saját szerkesztés.

A forrás útvonal (Source Path) oszlopban az állítható be, hogy a lokális számítógépen hol találhatja azt a projektet, amit az adott virtuális gépre szeretnénk másolni. Az alapútvonal (Base Path) beállításával megadható, hogy a virtuális gépen hova másolja a projektet.

Ezek beállítása után a kiemelt gomb (Copy Project to Slave) megnyomásával indítható el a másolás folyamata. Az elején hitelesíti a virtuális gépeket, hogy az előkövetelményeknek megfelelőek-e, de ez egy külön gomb segítségével (Verify) is megtehető. A folyamat pár percet vesz igénybe, függően a projektek méretétől.

A másolás befejeztével nincs más dolgunk, mint létrehozni a munkamenetet a Jobs menüpont alatt (*11. ábra*).

11. ábra: **Jobs menüpont**



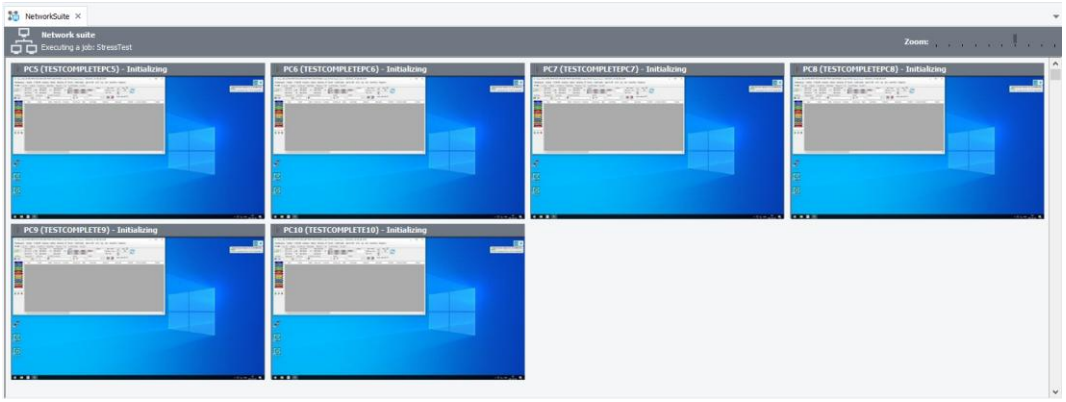
Forrás: saját szerkesztés.

Ezen a felületen még finomíthatunk a tesztünkön. Lehetőség nyílik arra, hogy megadjuk mi történjen, ha lefut a teszt és a tesztfutási eredmény mentési helyét és részletességét is állíthatjuk. A futást a létrehozott munkamenetre jobb gombbal kattintva indíthatjuk el.

3.3.1. Tesztfutás monitorozása

A futás elindítása után a Network Suit menüpont Run State oldalán követhető élőben a tesztfutást (*12. ábra*).

12. ábra: **Futás monitorozása**



Forrás: saját szerkesztés.

Az élő képek segítségével pontosan tudjuk, hogy hol tart a tesztfutás és melyik gépen éppen mi történik. A szinkronizációs pontokat is itt lehet nyomon követni. Abban az esetben, ha az egyik gép elért egy ilyen pontra akkor a „Running” felirat helyett a szinkronizációs pont neve fog megjelenni.

3.3.2. Tesztfutás eredménye

A tesztfutás végén az összes gépről felmásolja a TestExecute a tesztfutás eredményét, pontosan olyan formátumban, mint a funkcionális tesztfutás során. Mivel ebben az esetben nem csak arra vagyunk kíváncsiak, hogy funkcionálisan hibátlan a rendszer, így célszerű adatbázis oldalon is vizsgálódni.

A TestComplete megközelítőleg másfél perc alatt hoz létre ellenőrzött státuszban egy ügyletet, ebbe beletartozik a szinkronizációk és az adatok ellenőrzése miatti időtöbblet. A hat gép segítségével óránként megközelítőleg kétszáznegyven ügyletet lehet felvinni, ha folyamatosan ezt csinálná a teszt.

A következő fejezetben a tesztfutás eredményét az adatbázis oldaláról mutatjuk be, ami az akciók futási idejét és a lehetséges hibalehetőségeket fogja tartalmazni.

4. Értékelés, elemzés

A teszt futása során összesen négyezer-kilencszázharminckilenc akció hajtott végre a TestComplete. Ezeket ügylettípusonként, akciónként és altípusonként csoportosíthatók.

Az egyes akciók végrehajtására használt időn kívül más adatokat is érdemes ellenőrizni. Az egyik fő problémát ilyen nagy terhelés mellett az adatbázisoknál az úgynevezett holtpont jelenti. Ez alatt azt a helyzetet értjük, amikor kettő (vagy több) folyamat zárolja a különálló erőforrást, ilyenkor a folyamatok elkezdenek várni arra, hogy felszabaduljon. Az SQL szerver ezt úgy kezeli, hogy az egyik folyamatot leállítja, hogy megoldja az ütközési problémát. Ennek köszönhetően az egyik folyamat nem jár eredménnyel, ami akár adatvesztést is okozhat.

Érdekes még az adatbázisban tárolt eseménynaplók vizsgálata is, amik tárolhatnak hibaüzenetet és jelzést is a rendszerrel és más kiszolgálókkal

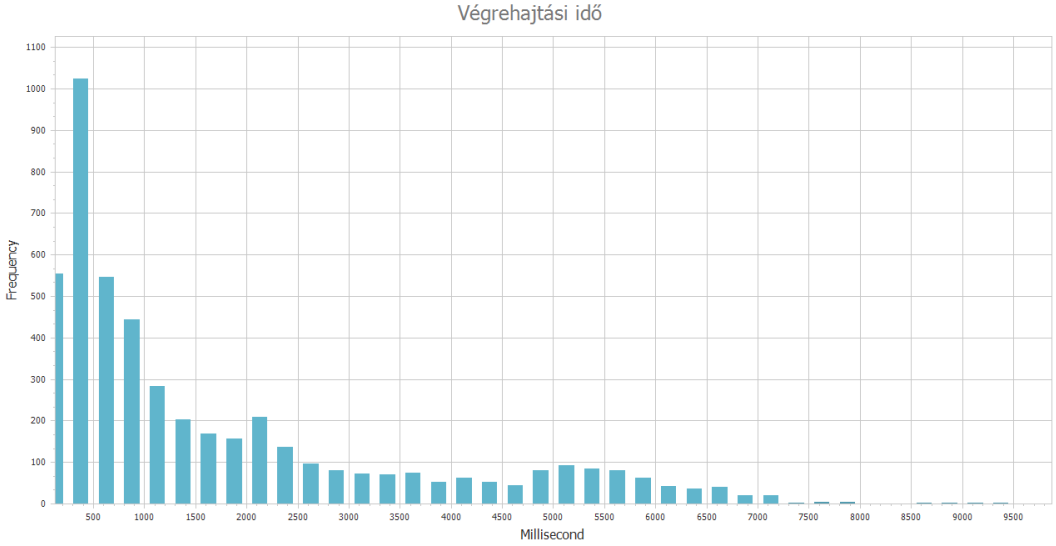
kapcsolatban. Az ellenőrzéseket és naplózásokat az architektúráért felelős csapat definiálja. Az elemzési és futtatási pontokat a cég működéséhez kell igazítani.

A fent leírt adatok tárolása és átláthatósága érdekében érdemes létrehozni külön erre egy adatbázist és az adatok megjelenítésére egy programot. Ebben tesztenként kerülnek rögzítésre az adott futás eredményei, amibe nem csak a végrehajtási idők, hanem az esetleges hiba- és figyelmeztető üzenetek is beletartoznak. Ennek köszönhetően egy helyen át lehet nézni az eredményeket, emellett figyelemmel lehet követni, hogy az egyes hibajavítások vagy fejlesztések milyen hatással voltak a rendszerre: ha az új verzióban lassulás látható, akkor abból következtethetünk arra, hogy az implementált fejlesztés okozta. Vezetői riport készítésére is alkalmas, ami a menedzsmentnek ad visszajelzést, hogy jó-e az irány vagy sem.

A teszt futtatása során a rendszer viselkedését, – annak köszönhetően, hogy a TestComplete lehetőséget ad az élő monitorozásra – nem csak az adatbázisból kinyert adatokból lehet megvizsgálni. A stressztesztet futtatása során látható volt, hogy a felület hogyan reagál a megnövekedett terhelésre, hiszen a funkcionális tesztelésre használt szkriptek voltak alkalmazva. Ennek köszönhetően a TestComplete által generált eredményből a felületen keletkezett esetleges működési rendellenességek is kiszűrhetőek.

Az adatbetöltés stressztesztelése során a tesztelt felületekre külön-külön meghatározott értékek alatt töltődtek be az adatok. A felületeken nem történtek fennakadások, így megállapítható, hogy a rendszer nagy adatmennyiség kiszolgálására is alkalmas. A végrehajtási idők elvárt értékét öt másodpercben határozta meg a cég.

13. ábra: **Végrehajtási idő**



Forrás: saját szerkesztés.

A *13. ábra* negyed másodperces intervallumokra van bontva. Látható, hogy a közel ötezer akció 52 százaléka egy másodperc alatt végrehajtódott. Az esetek 92

százaléka az elvárt átlag érték alatt ment végbe, így kijelenthető, hogy a rendszer a megnövekedett terhelés ellenére is az elvárt szint közelében működött. Az elvárt értkénél nagyobbak (8 százalék) esetén nem következtethetünk egyértelműen hibára, előbb érdemes megvizsgálni, hogy mi okozhatta azokat. A felfelé eltérő értékeket okozhatja például a gyorsítótáras megoldások esetén az első felhasználói kapcsolat, ahol a gyorsítótár feltöltését is elvégzi, így magyarázattal szolgálhat az értékhatár fölötti eredményre, hiszen ez tervezetten történik. Az olyan külső interfész, amellyel a fejlesztő cég nem rendelkezik elérhető, valós végponttal, szintén okozhat pár tizedmásodperces lassulást. Az ilyen esetekben kapott érték pontos kimutatása fontos értékként szolgál az architektúráért felelős csapat számára, hiszen segítségükkel kijelölhetik az adott területen a fejlesztési irányokat. Például a gyorsítótár feltöltését át lehet szervezni a rendszerindítás után végrehajtandó folyamatokba. Interfészek esetén, ha üzletileg van rá mód, akkor az úgynevezett „Fire and Forget” feladásokra való átállás jelenthet megoldást.

A stressztesztfutás alatt holtpont nem keletkezett, így az is kijelenthető, hogy a rendszer egyidejűleg több felhasználó aktivitását is képes kezelni, anélkül, hogy bármilyen fennakadás történne. A TestComplete futási eredményből az is kiderült, hogy a megnövekedett terhelés ellenére a felületen nem következtek be hibák.

Az eredményeknek köszönhetően már egy tisztább képet kapunk a szoftver határairól, aminek segítségével garantálni tudjuk a megrendelőnek felé, hogy a rendszer képes kiszolgálni az igényét, amennyiben az a határon belülre esik.

5. Összegzés

Korábban Béli et al. (2022) rávilágított arra, hogy az automatizált funkcionális tesztelés bevezetése hatékonyabb tesztelési formának bizonyul ismétlődő teszteléseknél, azonban a manuális tesztelésnek is megmaradt a létjogosultsága egyszeri tesztek esetén. Ebben a tanulmányban egy másik teszttípust vizsgálunk: a nem funkcionális tesztelés automatizálását. Pontosabban azt vizsgáljuk, hogy az automatizált szoftvertesztelő rendszer, a TestComplete megoldást tud-e nyújtani egy szoftver funkcionális tesztelése mellett a hatékony és pontos terheléses tesztelésre. A stressztesztet gyakorlati példán, valós üzleti problémán keresztül mutatjuk be: az FX Software Zrt. által fejlesztett InFoRex rendszert vetjük terheléses teszt alá. A stressztesztelés kivitelezéséhez a TestComplete automatizált tesztelési környezetet használunk, amely garantálja a szoftver magas minőségét, használata időt spórol és pénzt takarít meg a vállalatnak. A stressztesztfuttatásának köszönhetően egy tisztább képet kaptunk a tesztelt rendszer (InFoRex) határaival kapcsolatban. A teszt eredményei szerint garantálható a rendszer megbízhatósága kritikus helyzetekben is, míg az eredmények tárolására készített szoftver segítségével korán észrevehetővé tehető, ha rossz irányba indul el egy fejlesztés. A gyakorlati rész megvalósítása rávilágított arra, hogy drasztikusan kisebb erőforrás igény mellett sokkal nagyobb terhelés alá tudunk vetni egy-egy szoftvert, mintha ugyanezt a tesztelést manuálisan végeznénk el. Az automatizált tesztelés során viszont továbbra is elengedhetetlen az emberi kontroll, hiszen lehetnek olyan helyzetek, hogy egy adott futás már az elején valami miatt megáll, ekkor pedig a kapott eredmény további részét hibásnak fogja

jelezni. Ebben az esetben el kell tudni dönteni, hogy magával a teszttel történt a hiba vagy valós rendszerhiba az oka. Ehhez viszont jelenleg még humánerőforrásra van szükség. Összességében a stressztesztautomatizálása hozzásegít egy céget egy minőségi és stabil szoftver fejlesztéséhez, mindezt alacsony erőforrásfelhasználás mellett.

Köszönetnyilvánítás

Jelen cikk a 00116-os számú projekt részeként, az Innovációs és Technológiai Minisztérium Nemzeti Kutatási Fejlesztési és Innovációs Alapból nyújtott támogatásával, a 2020-1.1.2-PIACI-KFI-2020 pályázati program finanszírozásában valósult meg.

Irodalomjegyzék

Béli M., Tóth F., Varga T. (2022): „Innováció a szoftvertesztelésben – Megoldás-e az automatizálás?”. *Jelenkori Társadalmi és Gazdasági Folyamatok*, 17 (3-4): 47–66. https://doi.org/10.14232/jtgf.2022.3-4.47-66

Catelani, M., Ciani, L., Scarano, V. L., Bacioccola, A. (2011): Software automated testing: A solution to maximize the test plan coverage and to increase software reliability and quality in use. *Computer Standards & Interfaces*, 33 (2): 152–158. https://doi.org/10.1016/j.csi.2010.06.006

Diaz, E., Tuya, J., Blanco, R. (2003): Automated software testing using a metaheuristic technique based on tabu search. Proceedings of 18th IEEE International Conference on Automated Software Engineering, (2003): 310–313. https://doi.org/10.1109/ASE.2003.1240327

HTB (2023): <https://hstqb.org/> (2023.05.05.)

ISTQB (2021): Masterfield Traning: ISTQB Certified Tester Foundation Level traning program (Titkosított)

ProofIT (2023): A nem funkcionális szoftvertesztelés típusai. <https://proofit.hu/blog/a-nem-funkcionalis-szoftver-teszteles-tipusai/> (2023.05.05.)

SmartBear (2023): TestComplete Documentation. <https://support.smartbear.com/testcomplete/docs/index.html> (2023.05.05.)