Eötvös Loránd Tudományegyetem

Informatikai kar

Programozáselmélet és Szoftvertechnológiai Tanszék

Webes programozási beadandók automatizált tesztelése és távoli végrehajtása

Témavezető:
Cserép Máté
Egyetemi tanársegéd

Szerző:
Kostenszky Kálmán Ákos
Programtervező Informatikus MSc

Budapest, 2022

EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

DIPLOMAMUNKA TÉMABEJELENTŐ

Hallgató adatai:

Név: Kostenszky Kálmán Ákos

Neptun kód: AYEQQL

Képzési adatok:

Szak: programtervező informatikus, mesterképzés (MA/MSc)

Tagozat : Esti

Belső témavezetővel rendelkezem

Témavezető neve: Cserép Máté

munkahelyének neve, tanszéke: **ELTE-IK, Programozáselmélet és Szoftvertechnológia Tanszék**

munkahelyének címe: **1117, Budapest, Pázmány Péter sétány 1/C.**

beosztás és iskolai végzettsége: **egyetemi tanársegéd**

A diplomamunka címe: Webes programozási beadandók automatizált tesztelése és távoli végrehajtása

A diplomamunka témája:

(A témavezetővel konzultálva adja meg 1/2 - 1 oldal terjedelemben diplomamunka témájának leírását)

A diplomamunka célja az ELTE Informatikai Karán használt TMS beadandó-kezelő rendszer funkcióinak bővítése annak céljából, hogy az támogatni tudja webes alkalmazások futtatását, automatikus rendszer tesztelését, illetve manuális ellenőrzését. A diplomamunka során feldolgozandó probléma körök:

*** Webes alkalmazások futtató környezetének kialakítása.**

*** Webes alkalmazások interfészeinek (pl: GUI, REST) automatizált rendszertesztelése.**

*** Webes alkalmazások távoli futtatása manuális tesztelés céljából.**

A diplomamunka magában foglalja a fentebb felsorolt funkciók követelményeinek részletes feltárását (követelményelemzés), a piacon elérhető alternatívák vizsgálatát és összehasonlító elemzését. Ezt követően foglalkozik a megoldások megtervezésével (szoftvertervezés). Mindezt oly módon, hogy azok illeszkedjenek a már meglévő rendszerkomponensekhez. A tervezés során megvizsgálására kerülnek az automata tesztek végrehajtásához szükséges eszközök, keretrendszerek (pl Selenium, Cypress). A tervezési és elemzési munka eredményeként kiválasztásra kerülnek azok a programkönyvtárak és keretrendszerek, melyek támogatják a kijelölt funkcionalitások megvalósítását. Végül pedig cél az is, hogy a diplomamunka keretében elkészüljön egy, a meglévő TMS rendszerbe integrált prototípus alkalmazás is. Az elemzési, tervezési és megvalósítási munkák kiemelt szempontja, hogy a megoldások jól dokumentáltak és tovább fejlesztésre alkalmasak legyenek.

Budapest, 2021. 12. 07.

Tartalomjegyzék

1. fejezet

Bevezetés

Diplomamunkám témájául az ELTE Informatikai karán fejlesztett és aktívan használt TMS rendszer tovább fejlesztésének lehetőségét vizsgálom. A cél, hogy a TMS képes legyen a webes hallgatói beadandók távoli környezetben történő futtatására és automatizált tesztelésére.

Az elosztott, hálózati alkalmazások térnyerésének köszönhetően rohamosan nő a kereslet a webfejlesztés különböző területein jártas szakemberek iránt, ennek megfelelően a témakört feldolgozó egyetemi kurzusok is nagy népszerűségnek örvendenek.

A programozási számonkérések automatizált tesztelése rég jelenlévő igény az egyetemi képzésben, különösen azokon a területeken, ahol nagyszámban tanítanak hallgatókat. Napjainkban számos ilyen megoldás van használatban, olyan ami webtechnológiákat is támogat azonban kevés.

Dolgozatomat egy átfogó irodalomkutatással (2) kezdem, melynek során először röviden ismertetem a TMS-hez hasonló beadandó kezelő rendszerek web alkalmazás tesztelési képességait. Ezt követően a szakdolgozati címben megfogalmazott kettős célhoz kapcsolódó eredményeket vizsgálom külön-külön. Először is a virtualizációs technológiákat elemzem, különös tekintettel a Dockerre, mint lehetséges eszközei a webalkalmazások távoli végrehajtásának. A szakirodalmi elemzés utolsó részében pedig a teszt automatizálás lehetőségeit vizsgálom a webes alkalmazások esetén: egyrészt megvizsgálva a jellemző kihívásokat, másrészt egy technológiai áttekintést adva a lehetséges eszközökről.

A következő fejezetben, a követelményelemzés (3) során, részleteiben feltárom a funkcionális követelményeket az egyes felhasználói szerepkörök szerint. Illetve számba veszem a fontosabb nem-funkcionális követelményeket is.

A Rendszerterv (4) fejezetben a fontosabb implementációs részleteket ismertetem, melyek kulcs szerepet játszanak a címadó követelmények megvalósításában. A dolgozat mellé egy prototípus is készült, mely nem valósítja meg az összes funkcionalitást, a lényegei eltéréseket a fejezetben külön is ismertetem.

Végül az Összegzés (5) során röviden összehasonlítom az megtervezett megoldást a 2. fejezetben bemutatott megoldásokkal, illetve egy átfogó képet igyekszem adni a rendszer természetéről és további fejlesztési lehőtésgeiről.

2. fejezet

Irodalomkutatás

A Bevezetőben megfogalmazott célok értelmében szükséges egyrészt távoli futtatási környezet kialakítása, másrészt pedig teszt automatizálás funkció megvalósítása. Ebben a fejezetben ennek a két témának a szakirodalmát tekintem át külön-külön. Elöljáróban ismertetem a webes programozási beadandók értékelésének kihívásait, illetve röviden bemutatok néhány megoldást.

2.1. Webes programozási beadandókezelő rendszerek

A webfejlesztéshez kapcsolódó kurzusok beadandóinak ellenőrzése és értékelése egyedi kihívásokkal tarkított. Bevezető programozás kurzusok fordításához és futtatáshoz általában elégséges egy programnyelvi környezet. A programmal történő interakció parancssoron keresztül történik, azaz a program szöveges bemenetet olvas a parancssorról, kimenetként pedig általában a parancssorra ír valamilyen szöveget, egyéb esetben a bemenet és/vagy a kimenet is egy strukturált fájl lehet. Közös jellemzőjük, hogy a program kimenetét értékelik ki. Webalkalmazások esetén az interakció áttevődik egy böngésző grafikus felületére, amely lehet egy egészen egyszerű HTML oldal, vagy akár egy ún. gazdag webalkalmazás (*Rich Web Application*), mely interkaciók széles skálját biztosítja a felhasználónak. Egyszerűbb esetben a webalkalmazás grafikus felület helyett egy webszervíz interfészen keresztül is elérhető, ezeken keresztül az interakciók könnyebben programozhatóak, de a kérések megfelelő formátumának kialakítása és a válaszok feldolgozása így is komplexebb, mint a parancssoros alkalamzások esetén. További kihívást jelent a megfelelő futtatási környezet kialakítása. Legalább egy webszerverre szükség van, de magasabb szintű

programozási kurzusok során megjelenhetnek egyéb szolgáltatási komponensek (pl. adatbázis), illetve igény merülhet fel arra is, hogy a webalkalamazás az interneten keresztül távoli erőforrásokat érjen el [1]. Egyfelől tehát az automata tesztrendszernek képesnek kell lennie a felhasználói utasítások végrehajtására egy böngészőben, másfelől a hallgatóknak be kell tartania a felhasználói felület kialakítására vonatkozó jó néhány különleges követelményt [2]. Célszerű továbbá az oktatók számára lehetőséget biztosítani a webalkalmazás távoli elérésére, levéve ezzel a vállukról a komplex futtatási környezetek kialakításának terhét.

A következőkben röviden bemutatok néhány, a fenti problémákat részben vagy egészben megoldó rendszert.

2.1.1. AWAT

Az AWAT (*Automated Web Application Testing*) rendszert 2008-ban alkották meg [2], alapját a Ruby nyelven írt WATIR teszt automatizációs könyvtár adja. Az AWAT egy adatvezérelt rendszer, amelyben a tesztesetek és paraméterek leírására egy Excel táblában van lehetőség. A rendszer nem képes a beadott webalkalmazások futtatására. A koncepció megköveteli, hogy a hallgató maga gondoskodjon az alkalmazás telepítéséről úgy, hogy az az oktató számára egy publikus URL-n keresztül elérhető legyen. A tesztek futtatáshoz az előbb említett Excel fájlra és az URL-re van szükség. Az alkalmazás nem képes az oldalak között navigálni, ezért komplexebb felhasználói folyamatokat sem tud megvalósítani. Csupán egy előre definiált címet nyit meg, az Excel táblázatból pedig kiolvassa a bemeneti mezőkbe írandó adatot és az elvárt kimenetet. Az alkalmazás képes egyszerűbb hibák és kivételek kezelésére is.

2.1.2. APOGEE

Az APOGEE (*Prototype of Automated PrOject Grading and instant fEEdback system for web computing*) egy 2008-ban megtervezett rendszer prototípus [3]. A rendszer célja nemcsak funkcionális tesztek végrehajtása, hanem olyan minőségi szempontokat is értékel, mint például a robosztusság és a biztonság. Tervezési szempont volt, hogy az osztályozáson túl részletes visszajelzést adjon a hallgatónak, így például képes részletes leírást adni a teszt során előforduló hibák reprodukálásához. Az AWAT-hoz hasonlóan szintén WATIR alapú. A beadandó specifikációja két

részből áll: funkcionális és nem-funkcionális követelmények, illetve a grafikus felület komponenseinek névadási konvenciója. A funkcionális és nem-funkcionális követelmények további kategóriákra bonthatóak, melyek alá a követelményekhez tartozó tesztesetek rögzíthetőek. A teszteseteket Ruby programozási nyelvvel lehet megadni, azaz az oktatónak ismernie kell a nyelvet, így viszont lehetővé válik az oldalak közötti navigáció és ezáltal a teljes felhasználói folyamatok leírása. A tesztek végrehajtása során a rendszer képernyőképeket készít az böngésző állapotáról, ezzel támogatva a lépésenkénti reprodukálhatóságot. Az AWAT-hoz hasonlóan a hallgató feladata, hogy a webalkalmazását egy távolról is elérhető szerverre telepítse.

2.1.3. WebWolf

A WebWolf 2015-ben készült [4] kifejezetten a belépő szintű webfejlesztési kurzusokhoz. Ennek megfelelően az alkalmazás fejlesztése során fontos szempont volt az egyszerűség, ezzel is ösztönözve az oktatókat a használatra. Az előzőekben bemutatott rendszerek megkövetelték az oktatótól egy specifikus nyelv ismeretét, mely nem tartozott feltétlenül a kurzuson oktatott nyelvek közé. Tevezési szempont volt több böngésző támogatása is, ami lehetővé tette, hogy a hallgató által is használt böngésző verzióban is tesztelhető legyen az alkalamzás. Ezen szempontok miatt a Selenium webdriverre építették az alkalmazás tesztelő komponensét. A felhasználás további egyszerűsítése értekében egy saját API-t is fejlesztettek a Selenium fölé, ezzel is egyszerűsítve komplexebb elő- és utófeltételek leírását, illetve a leggyakoribb felhasználói lépések automatizálását.

2.1.4. Submitty

A Submitty egy korszerű rendszer “Comparing jailed sandboxes vs containers within an autograding system”, amely a teszt komponensét a WebWolfhoz hasonlóan Seleniumra építi. Az előzőekben bemutatott alkalmazásokkal szemben a rendszer maga gondoskodik a hallgatói beadandó futtatási környezetének kialakításáról és futtatásáról, így a hallgatónak elég a forráskódot feltöltenie, amelyet aztán az oktatói konfiguráció alapján az alkalamazás lefordít, futtat és tesztel. A futtatáshoz Docker konténereket használ, amely kifinomult erőforrás felhasználás szabályozást, a párhuzamosan futó értékelések izolálását és hálózati konfigurációk kialalkítását teszi

lehetővé. Hasznos funkcióként megjelent számos biztonsági komponens is, például egy proxy réteg, ami felügyeli és szükség esetén korlátozza a ki- és bemenő forgalmat.

2.2. Webalkalmazások távoli futtatása

Az alkalmazások távoli futtatásához 3 komponensre van szükség:

- Távoli futtatási környezet
- Hálózati protokoll, pl: HTTP, RDP, RBF, SSH, stb.
- Kliens alkalmazás, pl: webböngésző, parancssoros kliens (curl, wget), távoli asztal kliens, stb.

A kliens elsősorban a protokoll függvénye. Mivel webalkalmazások futtatásáról van szó, a protokoll http/https lesz. A továbbiakban csak a távoli futtatási környezet problémakörét vizsgálom. Elvárás, hogy egy szerver több alkalmazást is futtathasson egyidőben egymástól függetlenül, egyedi konfigurációval, interferencia nélkül, a szerver erőforrások megosztásával. Ezeknek az igényeknek a kiszolgálásra virtualizációs technológiák sora áll rendelkezésre. A következőkben röviden bemutatom a két legelterjedtebb megoldást: a virtuális gép menedzser (VMM) alapú hardver virtualizációs technológiákat illetve az operációs rendszer szintű virtualizációs technológiákat [5, 6]. Utóbbiak részletes vizsgálata után teljesítmény összehasonlítást végzek.

2.2.1. Hardver virtualizáció

A VMM, más néven *hypervisor* technológiák magasabb fokú izolációt tesznek lehetővé, így a virtualizált környezet saját operációs rendszert használhat. A VMM megvalósítása történhet szoftveresen (*type 2*) és hardveresen (*type 1*) is. Utóbbi gazdaságosabb erőforrás felhasználással jár, de speciális hardver architektúrát igényel[6]. A magasabb fokú izoláció biztonságosabb végrehajtást eredményez, ennek azonban a teljesítményben kell megfizetni az árát: erőforrást igényel a VMM és a vendég operációs rendszer kiszolgálása is [5]. Bár a virtuális gépek teljes absztrakcióval rendelkeznek a hardver fölött, több folyamat egyidejű futtatására is képesek. A gyakorlatban jellemző, hogy általában egyetlen szolgáltatás futtatását végzik [7].

2.2.2. Operációs rendszer virtualizáció

Az operációs rendszer szintű virtualizáció esetén - amit *type 3* vagy *type-c* virtualizációnak is [6] neveznek, nem alkalmaznak *hypervisort*, és saját operációs rendszerre sincs szükség. A virtualizáció itt a rendszererőforrások (fájlrendszer, hálózat, konfiguráció, szolgáltatások, stb.) elérésére korlátozódik. Legelterjedtebb formája napjainkban a konténerizáció, amely az utóbbi egy évtizedben rohamos fejlődésen ment keresztül. A technológiai alapok már jóval korábban megjelentek a Linux platformokon. A konténerizáció elődjének az úgy nevezett *jailed sandbox* környezetek tekinthetőek, amelyeknek az a lényege, hogy a futtató felhasználó (és az alkalmazás) korlátozott hozzáféréssel rendelkezik az erőforrásokhoz. [8]. Az elnevezés nem egy technológiát takar, hanem egy kontrollált környeztet, amelyet több operációs rendszer szolgáltatás egyidejű használatával alakítanak ki. A *jailed sandbox* környezetek kialakításakor nem alkalmaznak semmilyen virtualizációs eszközt, az alkalmazás közvetlenül az operációs rendszerrel kommunikál, és elérhetőek számára a telepített szolgáltatások. A környezetek kialakításánál alkalmazható Linux kernel szolgáltatások: [8, 6]:

- chroot (change root): virtuális gyökérkönyvtár, melyen kívül a futtató felhasználó nem ér el erőforrásokat.
- névterek: globális rendszer erőforrások futattó felhasználónkénti egyedi konfigurációja.
- seccomp (secure computing mode): rendszerhívások korlátozása folyamatonként.
- rlimit (resource limit): erőforrás felhasználás korlátozás folyamatonként.
- cgroups (control groups): erőforrás felhasználás korlátozása folyamatonként, folyamatcsoportonként.

A *jailed sandboxok* kétségtelen előnye a minimális teljesítmény-veszteség és a teljes körűen finomhangolható konfiguráció. Hátránya azonban, hogy biztonságos kialakítása nagy szakértelmet igényel, illetve hogy a futtatáshoz szükséges szolgáltatások telepítése a gazda operációs rendszerre történik. [1]

Az előbb felsorolt technológiákra épít a konténerizáció, amelynek két altípusa a gép-, illetve az alkalmazás-konténerek [6]. A két megközelítés eltérő technológiai

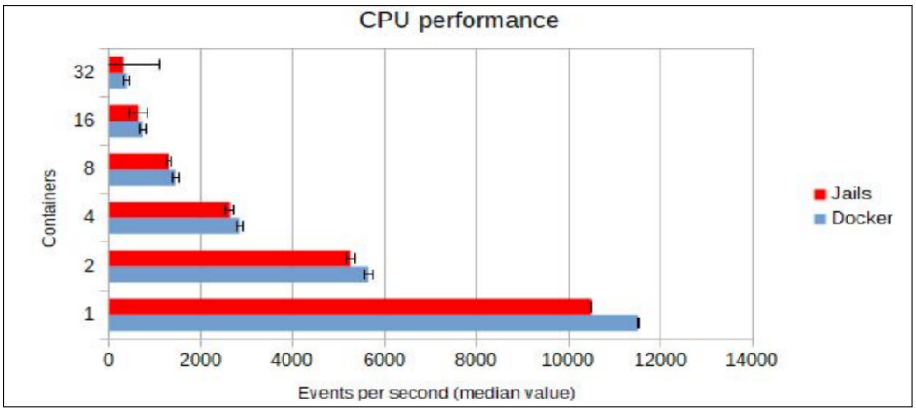
alapokon nyugszik. A gép-konténerek teljes mértékben a gazda operációs rendszerre épülnek, de teljes fájlrendszereket tartalmazhatnak. Használatuk lehetővé teszi több folyamat egyidejű futtatását a konténeren belül, minimális teljesítmény veszteséggel. Az alkalmazás-konténerek ezzel szemben egy folyamat futtatására vannak optimalizálva (amely lehet akár egy operációs rendszer is). Előbbiek alkalmasak könyvtárak, nyelvek és szolgáltatások izolált telepítésére, míg utóbbiak alkalmazások csomagolására és terjesztésére környezetükkel együtt.

2.2.3. Teljesítmény összehasonlítás

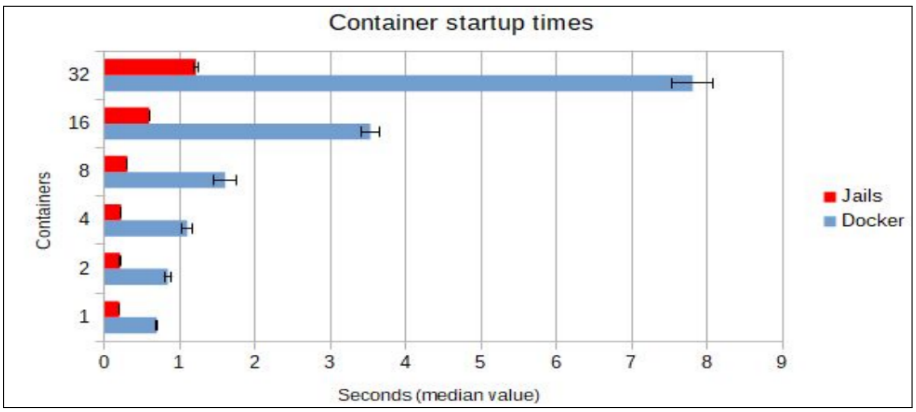
A következőkben röviden bemutatom az említett technológiák teljesítményének összehasonlításakor nyert eredményeket.

Jails és Docker

Először is vizsgáljuk meg a különbségeket egy gép-konténer és egy alkalmazás-konténer között! A Jails gép-konténer technológia, amit BSD Unix operációs rendszerre készítettek, míg a Docker alkalmazás-konténer technológia, mely a Linux alapú Linux Containers (LXC) technológia kiterjesztése [9]. A "Jails vs Docker: A performance comparison of different container technologies" tanulmányban végzett teljesítmény tesztek összehasonlítják a két technológia erőforrásigényét (CPU, memória, IO, hálózat) valamint indítási idejüket. A metrikákat külön-külön 6 tesztesetben vizsgálták, mindegyik tesztesetet tízszer végezték el 6 különböző konténer konfigurációban: 1, 2, 4, 8, 16, 32 konténer futása egy időben (összesen 720 teszt). A párhuzamosan futtatott konténerek száma vizsgálatom szempontjából különösen érdekes, mivel feltehető, hogy az automata osztályozó rendszer több beadandót fog ellenőrizni egy időben. Az eredmények azt mutatják, hogy a Docker általánosságban, a vizsgált erőforrások mindegyikénél jobb eredményeket mutat erőforrás kihasználás tekintetében - különösen nagyobb konténer szám esetén -, azonban az indítási időben messze alulmarad, ahogy az a 2.1, 2.2 ábrákon is látható.



2.1. ábra. CPU teljesítmény összehasonlítás. Másodperenként végzett műveletek számának a tíz teszt futásra vett medián értéke. [9]



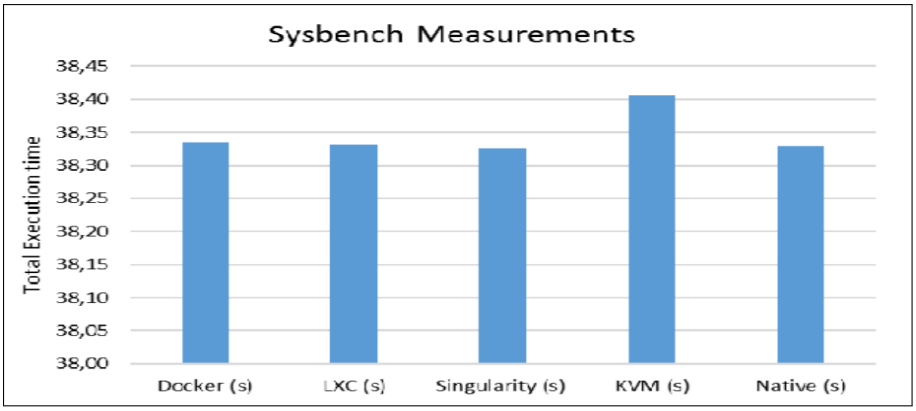
2.2. ábra. Indítási idő összehasonlítás. Másodpercben mérve a tíz teszt futásra vett medián értéke. [9]

KVM és Docker

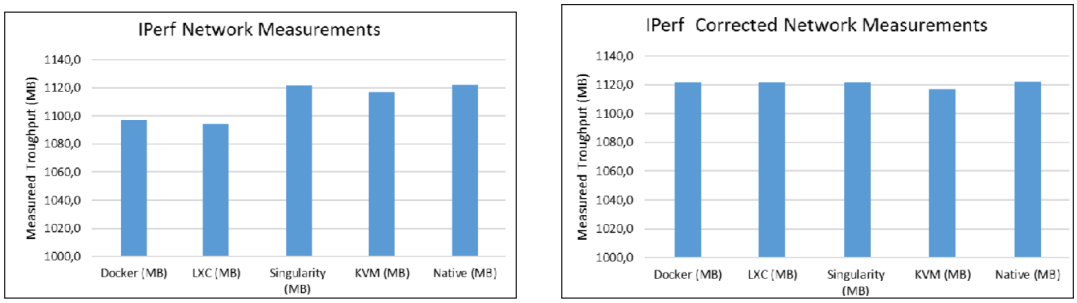
A KVM (Kernel Virtual Machine) Linux alapú, natív VMM technológia. A "Comparison of different Linux containers" 3 alkalmazás-konténer alapú technológiát (LXC, Docker, Singularity) hasonlít össze a KVM virtualizációval. Külön érdekesség, hogy referenciaként a közvetlenül a gazda operációs rendszeren futó alkalmazás teljesítményét is mérték. A mérések során a CPU és hálózati erőforrások kihasználást vizsgálták, mindegyik tesztesetet tizenegyszer végezték, majd átlagolták az eredményeket. Az eredmények szerint a konténer technológiák számítási teljesítménye közel azonos a natív teljesítménnyel, és a KVM sem teljesít kiugróan rosszabbul 2.3. A hálózati teljesítmény mérését több módban is elvégezték, ennek oka hogy a hálózati híd [1] (*network bridged*) módban mért eredmények szerint az LXC és a Docker jelentősen

[1] A hálózati-híd egy olyan hálózati eszköz, mely két hálózati szegmens összekötését teszi lehetővé az OSI modell szerinti adatkapcsolati rétegben. Létezik fizikai és virtuális híd: a konténerizációs technológiák virtuális hidakat használnak.

alul marad. A natív és KVM futáshoz képest több mint 3 %-os teljesítmény romlást tapasztaltak nagy szórás mellett. *Host networking* módban a konténerek közvetlenük osztoznak a hálózati kártyán a gazdagéppel, ennek hátránya hogy a gazda operációs rendszer és a konténer közös IP és port névteret használ. Az így megismételt mérések azt mutatják, hogy a natív futáshoz képest a konténerek kevesebb, mint 1 %-nyi teljesítményveszteséggel operáltak 2.4. A tanulmány ennek tükrében javasolja, hogy a felhasználási esetnek megfelelően válasszunk hálózati módot.



2.3. ábra. A teszt teljes végrehajtási ideje. CPU Sysbench benchmark alkalmazásával, másodpercben kifejezve a tizenegy teszt átlagolásával. [10]



(a) *Network bridge* mód (b) *Host network* mód

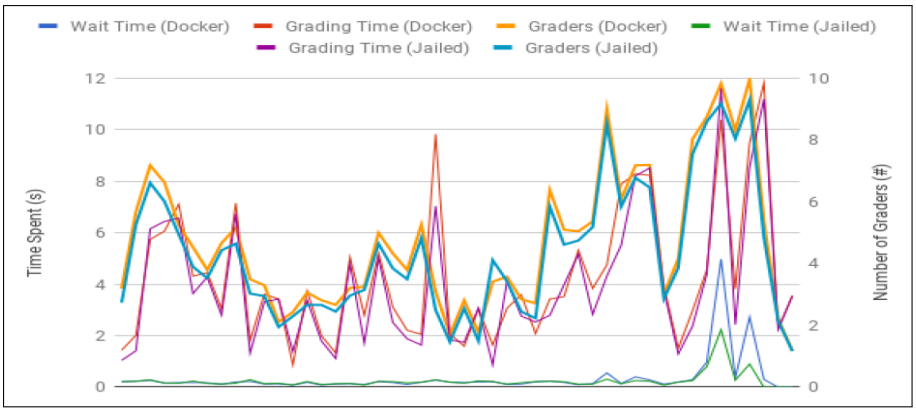
2.4. ábra. Hálózati áteresztő képesség, IPerf benchmark alkalmazásával, Megabyteban kifejezve a tizenegy teszt átlagolásával. [10]

Jailed sandbox és Docker

Az "Comparing jailed sandboxes vs containers within an autograding system" című [1] tanulmány érdekessége, hogy a jelen problémához – beadandók automatizált tesztelése – hasonló esettanulmány keretében valósult meg. Ennek köszönhetően a tesztesetek valós-életbeli forgatókönyveket vesznek alapul. Az elemzés röviden kitér a virtuális gépek használatának nehézségére ilyen környezetben (pl: indítási idő),

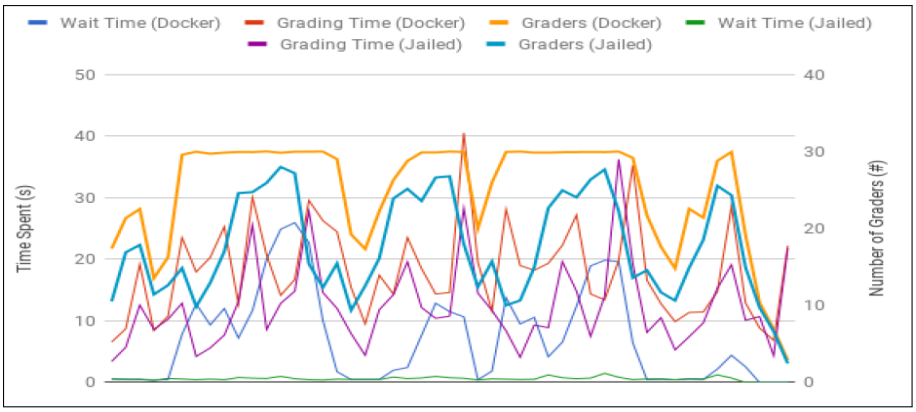
de ugyanezen okok miatt részletesebben nem foglalkozik velük. A tanulmány így a *jailed sandbox* és a Docker alapú megvalósításokat hasonlítja össze. A méréshez egy olyan időszak és kurzus adatait vették alapul, amikor a hallgatók nagyobb terhelésnek vetették alá a rendszert: kb. 540 beadandó érkezett a leadási határidő előtti éjszakán. A beadandók között előfordultak olyan hibás megoldások, melyek végtelen ciklust eredményeztek. Ezt az időszakot aztán újrajátszották a rendszerben, különböző gyorsítás mellett; a négyszeres gyorsítás azt jelenti, hogy egy óra terhelését 15 perc alatt engedték a rendszerre. További tesztparaméter volt a párhuzamosan futtatható osztályozó folyamatok maximális száma. Így tehát a rendszererőforrások kihasználásának vizsgálata mellett azt is mérték, hogy egy-egy osztályozás meddig tart, illetve mennyi időt vár egy alkalmazás a beadástól az osztályozás kezdetéig.

Négyszeres gyorsítás mellett és 10 párhuzamos osztályozó mellett azt találták, hogy a processzorhasználat majdnem azonos a két technológia esetében, a memóriahasználat azonban a Docker esetén átlagosan 1%-kal volt magasabb. A várakozási és végrehajtási idő közel azonos volt. Ez alól kivétel a határidő előtti utolsó néhány perc, amikor a hirtelen nagy mennyiségű beadandó érkezett a rendszerbe. Ekkor a Docker konfiguráció elérte a maximális 10 osztályozó kihasználást, ezzel 2-3 másodperccel megnövelve a várakozási időt. Ennek az volt az oka, hogy a konténerek a *jailed sandboxhoz* képest rendelkeznek egy indulási és leállási idővel, amelyek átlagosan 1,4, illetve 1 másodpercig tartanak. A 2.5 ábrán látható, hogy a tesztek átlagos futási ideje a két technológia esetén közel azonos (piros és lila adatsorok), az átlagos várakozási idő a grafikon utolsó tizedében Docker esetén (sötétkék adatsor) több mint kétszerese a *jailed sandboxéhoz* képest (zöld vonal). Ennek oka, hogy Docker esetén itt már mind a tíz értékelő komponens állandó kihasználtsággal futott (narancs adatsor), és összességében is elmondható, hogy átlagosan több értékelő komponens futott egy időben, mint a *jailed sandbox esetén* (világoskék adatsor).



2.5. ábra. A tesztek várakozási és futási ideje négyszeres gyorsítás mellett. [1]

Ezt követően tizenhatszoros gyorsítás mellett is megvizsgálták a rendszerek terhelését, azaz az előbbi időszakot tovább negyedelték és négyszer egymás után lejátszották, emellett 28-ra emelték a párhuzamosságot - a tesztek egy 40 magos hardveren futottak. Ekkor a Docker konfiguráció már szinte folyamatosan kihasználta a maximálisan rendelkezésre álló párhuzamosságot, miközben a *jailed sandboxok* esetén erre nem volt szükség. A maximális processzor és memória kihasználás is 5, illetve 15 %-kal nőtt. A várakozási idő legrosszabb esetben 25 másodperccel volt hosszabb, de az átlagos várakozási idő csak 10 másodperccel nőtt. Ezzel szemben a *jailed sandboxok* szinte 0 másodperc közeli várakozási időt eredményeztek. Átlagosan 7,4 és 3,3 másodpercig tartott egy konténer létrehozása és leállítása. A négyszeres gyorsítás 2.5 ábráját összevetve a tizenhatszoros gyorsítás 2.6 ábrájával azonnal látható, hogy a görbe párok már egyáltalán nem követik szorosan egymást. A várakozási idő (zöld adatsor) a *jailed sandbox* esetén szinte elhanyagolható, míg Dockernél szélsőségesen ingadozó (világos kék adatsor). Ennek elsődleges oka is leolvasható: Docker esetén kisebb megszakításokkal a 30 értékelő komponens teljes kihasználtsággal futott (narancs adatsor).



2.6. ábra. A tesztek várakozási és futási ideje tizenhatszoros gyorsítás mellett. [1]

Utolsó kísérletként a tizenhatszoros gyorsítás mellett a párhuzamosságot a fizikai kapacitás (40) fölé emelték, azonban itt már a Docker környezet instabillá vált és a konténerek egy része hibával terminált. Ezzel szemben a *jailed sandbox* környezet stabil maradt.

Amellett, hogy a tanulmány rávilágít a konténeres konfiguráció korlátaira, egyértelműen jó iránynak tartja használatukat. A konténerek használatával könnyebb biztonságos, izolált környezetet kialakítani, bár a *jailed sandboxoknál* bemutatott néhány eszköz használatát továbbra is szükségesnek tartja (pl. rlimit) a biztonság fokozása érdekében. Hangsúlyozza, hogy a VM-ek esetén megszokott 30-60 másodperces indulási időkhöz képest a konténerek indítása a legrosszabb esetben sem ment 10 másodperc fölé. Rávilágít továbbá az optimalizáció lehetőségeire, ami történhet a konténerképek optimalizálásával vagy konténer életciklus optimalizálásával. Előbbi esetén jelzi, hogy a konténerképek nem voltak specializálva az egyes beadandók igényeire, így méretük meghaladta a 2.1 GB-t, ami szélsőségesen nagynak mondható: összehasonlításképpen egy Apache Webszerverre optimalizált konténer méretre 50 Mb körül van. A képek optimalizációja kisebb képméretet eredményez és gyorsabb elindulási idő érhető el általa. Az életciklus optimalizációját más erőforrások esetén létező koncepciók inspirálják, mint például *resource poolok* és szemétgyűjtők. Az *resource poolok* használata lehetővé tenné, hogy a konténereket előre legyártsuk és azokat *poolban* tároljuk, ahonnan a folyamatok igény szerint kérhetnek egy futtatásra kész példányt. A szemétgyűjtő pedig aszinkron módon, bizonyos időközönként törölné a használt konténereket.

2.2.4. Összefoglalás

A fentiek tükrében úgy gondolom, hogy az alkalmazás-konténerizáció optimális választás. Erőforrás és életciklus tekintetében használatuk olcsóbb, mint a virtuális gépek üzemeltetése. Emellett üzemeltetésük kényelmesebb, mint a *jailed sandbox* környezetek üzemeltetése és nagyobb fokú izolációt tesznek lehetővé a telepített erőforrások tekintetében. Problémát jelenthet az optimalizált konténerképek kialakítása, azonban egy-egy kép kialakítására a félév, tantárgy esetében egyszer van szükség. A Docker térnyerésének köszönhetően széleskörűen elérhetőek olyan konténerképkönyvtárak, melyek számos futtatási környezetet tesznek elérhetővé konténerkép formában, nem ritkán a gyártó által karbantartva. Ilyen konténer könyvtár például a Docker Hub, mely közel tízmillió konténerképet tartalmaz jelenleg, melyből közel tízezer hivatalos gyártói kép.

2.3. Teszt automatizálás

2.3.1. Tesztek kategorizálása

A szoftverciklus során alkalmazott teszteket számos szempont mentén lehet csoportosítani [11]. Az ismertebb szoftverciklus modellek [11, 12, 13] mindegyikében megjelenik a tesztelendő egység (továbbiakban SUT)[2] hatálya mentén történő osztályozás. Ezt az osztályozást nevezhetjük tesztszinteknek vagy tesztpiramisnak is [13]. Az egyes módszertanok némiképp eltérnek az elnevezésben, illetve az osztályozási szempontok részleteiben. A táblázatban 2.1 röviden ismertetem ezeket a kategóriákat.

Teszt szint	**Teszt hatálya**
Egység teszt	Egy szoftverrendszer legkisebb, önállóan tesztelhető egységei (pl: osztályok, függvények, csomagok).
Integrációs teszt	Az előbbi szinten azonosított egységek közötti kommunikáció, interfészek illeszkedése.
Rendszer teszt	Teljes szoftverrendszer
Átvételi teszt	Teljes szoftverrendszer tesztelés valós futtatási környezetben.

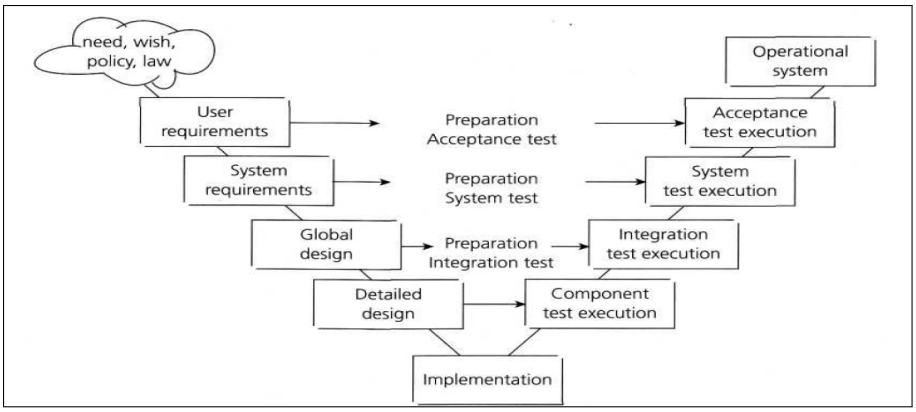
2.1. táblázat. Teszt szintek

A fenti kategorizálás jól illeszkedik két szempontra:

[2]System Under Test, esetleg Application Under Test

- Szoftverrendszer-specifikációs szintek és tesztek kapcsolata.
- Tesztautomatizálás jellemzői.

Előbbi azért fontos szempont, mert bár mindegyik tesztszint megjelenhet a webalkalmazások szoftverciklusában is, a beadandó feladatokat magasabb szintű specifikációval adják meg: például csak a szoftverfunkciók és alkalmazandó technológiák megadásával. Ezen a szinten jellemzően a felhasználói - elsősorban funkcionális - követelmények és rendszerkövetelmények dominálnak. A V-modell 2.7 pontos leírást ad arról, hogyan feleltethetőek meg a tesztszintek a szoftverrendszer-specifikációs szinteknek [12].



2.7. ábra. V-modell [12]

A specifikáció magasabb szintjeihez átvételi teszt és rendszerteszt társul. Tekintve, hogy a webalkalmazás-beadandók specifikációja ezekbe a kategóriákba illeszkedik, ezért a továbbiakban ezeket a szinteket vizsgálom. Ezeknek a tesztszinteknek közös jellemzője, hogy nehezen strukturálhatóak, a hibák kiváltó okait nehéz azonosítani, fekete dobozként tekintenek a SUT-re, funkcionális és nem funkcionális teszteket is tartalmazhatnak. Nem funkcionális tesztek tipikusan a terheléses és sérülékenységi vizsgálatok. A továbbiakban a funkcionális tesztek megvalósításával foglalkozom.

2.3.2. A tesztautomatizálás kihívásai

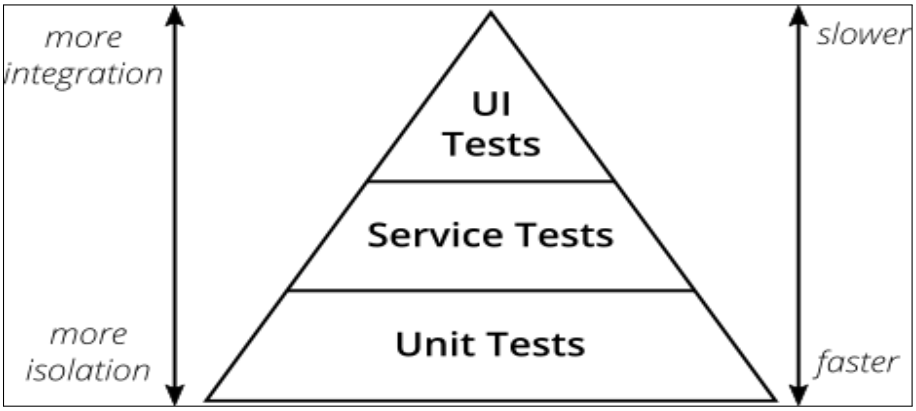
A rendszer- és átvételi tesztek a SUT külső interfészeivel lépnek interakcióba. Ezeket két kategória oszthatjuk:

- Grafikus felhasználói interfész (GUI),

- Alkalmazásprogramozási felület (API).

Webalkalmazások esetén a GUI-k alatt a webböngészők által megjelenített felületeket, API-k alatt pedig a webszervíz technológiákat (pl. SOAP, REST) értjük.

A tesztpiramis (2.8. ábra) különböző szintjein a tesztautomatizálás különböző kihívásokkal néz szembe. A korábban bemutatott tesztszintek a piramisban elsősorban a felhasználóifelület-tesztek (UI) és kisebb részben a service-tesztek alá sorolható.



2.8. ábra. Tesztpiramis [14]

A piramis arra reflektál, hogy egy szoftverrendszer tesztelése során az UI-tesztek a legkisebb hányadot teszik ki. Ennek oka, hogy ezek a tesztek a következő jellemzőkkel bírnak [13]:

- **Törékenyek:** A tesztesetek a felhasználói felület csekély változása eredményeként is hibássá válhatnak.
- **Drága karbantartani:** Ezen a szinten a tesztesetek komplexitása magas, mivel teljes felhasználói folyamatokat írnak le, ezért megírásuk időigényes.
- **Futásuk erőforrásigényes:** A felhasználói felületeken futó tesztek futtatása időigényes, mindemellett a teljes rendszer telepítése és elindítása erőforrásigényes.

A Web API tesztelés, mely inkább a service-teszt és UI-teszt szint határán helyezkedik el, könnyebben automatizálható. Ennek oka, hogy ezek a felületek jobban strukturáltak, így kevésbe törékenyek, ugyanis nem gép-humán interakcióra, hanem gép-gép interakcióra vannak tervezve. A következőkben kizárólag a GUI tesztek megvalósításával foglalkozom.

A megvalósítandó alkalmazásban cél, hogy egy-egy megírt teszteset több, ugyanazon specifikáció szerint megírt beadandó alkalmazásra is futtatható legyen. Ezen szempont mentén a tesztek kellő robosztusságának problémájára különös figyelmet kell fordítani.

A tesztek karbantartására fordított költségek mérsékelése szintén egy fontos szempont. Különböző tesztforgatókönyv (*script*) generáló eljárások állnak rendelkezésre.

Teszt scriptek generálása

A *scriptek* generálásának egy lehetséges módja a felvétel-és-visszajátszás (*record and replay*) típusú teszteset generálás. Ennek lényege, hogy a teszt készítője manuálisan végrehajtja a teszteset lépéseit, amelyet egy program rögzít. A program ezt követően elkészíti a később automatikusan futtatható futtatható scriptet. Ezek a rendszerek azonban igen törékenyek még [15], mivel a lokátor stratégiájuk[3] nem elég robosztus. Alkalmazásuk a jelenlegi keretek között azért sem lehetséges, mert feltételezi, hogy már létezik a tesztelhető alkalmazás egy futtatható változata.

Az adatvezérelt tesztelés alatt olyan módszert értünk, amelynek során a teszt script dinamikusan, futási időben egy strukturált adathalmazból veszi a tesztparamétereket [16]. A tesztparaméterek egyaránt lehetnek vezérlési szerkezetek és a SUT input adatai is. Hátránya ennek a megközelítésnek, hogy a teljes teszt *scriptet* előzetesen implementálni kell. Itt tartom érdemesnek megemlíteni ennek a megközelítésnek egy egyelőre még csak koncepció szintjén létező ígéretes továbbgondolását [17]. A leírt rendszer a teszt script-írást a következő eljárással helyettesíti: Egy *parser* elemezi a HTML fájlt, majd a tartalom alapján teszteseteket generál, melyek leírják a végrehajtandó utasításokat, azonosítják az input mezőket, de tesztadatot még nem tartalmaznak. A teszt adatokat a rendszer szintén generálja szélsőérték-analízis (*boundary value analysis*) segítségével, ennek bemenete egy xsd séma fájl, mely leírja az egyes input mezők típusát és lehetséges értékkészletét.

További lehetséges módszer, amikor egy magasabb szintű szöveges specifikáció értelmezésével (fordításával) áll elő a teszt script. Ennek egyik elterjedt változata a kulcsszó alapú tesztelés. Segítségével jól strukturált, természetes nyelvhez közeli

[3] A lokátor stratégia alatt azt az módszert értjük, ahogy az alkalmazás lokalizálja a felhasználói felület interaktív komponenseit.

mondatokban lehet definiálni a tesztesetek lépéseit, ideértve az előfeltételeket, teszt lépéseit, és az utófeltételeket. Egy ilyen mondat 3 tagból áll [18]:

- **Kulcsszó:** Egyszerű kifejezés, mely utasításokat, fogalmakat jelöl (pl: kattintás, kijelölés).
- **Objektum azonosító:** Azonosít egy objektumot a felhasználói felületen (pl: gomb, szöveges mező).
- **Paraméterek:** A kulcsszóhoz tartozó paraméterek, bemeneti adatok.

Például `Input Text id="name" John` azt az utasítást jelenti, ami a *name* azonosítóval ellátott szöveges mezőbe a *John* értéket gépeli. A kulcsszavakat hierarchiába lehet szervezni: magasabb szinten helyezkednek el a felhasználói kulcsszavak, melyek több alacsonyabb szintű, úgynevezett könyvtári kulcsszót fognak össze. A futtatás során a fordító egy végrehajtható kóddá alakítja a specifikációval adott tesztesetet. A könyvtári kulcsszavakból függvényhívások, utasítás-sorozatok állnak elő, az objektumok és paraméterek pedig függvény- argumentumokként jelennek meg. Szükség van tehát egy köztes rétegre, mely definiálja a kulcsszavak jelentését egy adott programnyelvi környezetben, ami által a teszttervezés és a technikai implementáció szétválik. A kulcsszavak 7 kategóriába sorolhatóak [18]:

- **Akció:** Művelet végrehajtása a SUT-en.
- **Állítás:** Egy predikátum igazságtartalmának vizsgálata a végrehajtás egy pontján.
- **Vezérlés:** A végrehajtás folyamatának megváltoztatása, pl elágazás.
- **Hozzáférő:** Érték kinyerése a SUT-ból.
- **Naplózás:** Naplózás a végrehajtás során
- **Összehangolás:** Várakozás a SUT állapotátmenetére.
- **Felhasználó:** Felhasználói - összetett - kulcsszavak

Az automatizált elfogadási tesztek használatát visszafoghatja az a probléma is, hogy az általános célú programozási nyelvek imperatív stílusában körülményes leírni azokat a magas szintű tesztlépéseket, melyek az elfogadási teszteket alkotják.

Példának okáért vegyünk egy tesztesetet, mely egy egyszerű keresést hajt végre a Google honalpon. Szövegesen a teszteset az alábbi lépések leírásával foglalható össze egy manuális tesztelő számára:

1. Nyisson egy böngésző ablakot.
2. Navigáljon a https://www.google.com oldalra.
3. A kereső mezőbe írja be a 'Wikipedia' szót.
4. Nyomjon rá a keresés gombra.
5. Ellenőrizze, hogy a találalti lista első eleme a 'Wikipeida' szó.

A megadott teszteset forráskódját egy kulcsszó alapú teszt-keretrendszerben a 2.1 kódrészlet írja le, míg egy lehetséges imperatív megfogalmazására például a 2.2 kódrészlet szolgál.

```
*** Settings ***
Library   Browser

*** Test Cases ***
Wiki test
    New Page    https://www.google.com
    Fill Text   input[title=Search]         Wikipedia
    Click       input[value=Google Search]
    Get Text    h3:first-child  should be   Wikipedia
```

2.1. forráskód. Teszteset Robot Keretrendszerben megírva

```
//importok elhagyva a tomorseg erdekeben

public class TestCases {

    @Test
    public void wikiTest() {
        WebDriver driver = DriverFactory.createChromeDriver();
        driver.get("https://www.google.com");

        WebElement searchBar = driver.findElement(By.cssSelector("
            input[title=Search]"));
        searchBar.sendKeys("Wikipedia");
```

```
12
13          WebElement searchButton = driver.findElement(By.cssSelector
                ("input[value=Google]"));
14          searchButton.click();
15
16          String result = driver.findElement(By.cssSelector("h3:first
                -child")).getText();
17          assertEquals("Wikipedia", result);
18
19          driver.quit();
20      }
21  }
```

2.2. forráskód. Teszteset Selenium és JUnit keretrendszerekben megírva

Jól látható, hogy a kulcsszavas megadás deklaratív formátuma tömörebb és szerkezetében is nagyon közel áll a természetes nyelvi megadáshoz. A kulcsszó alapú teszt-keretrendszerek megjelenése lehetőséget nyitott az *Acceptance Test Driven Testing* és annak egy speciális, egyben ismertebb változatának, a *Behaviour Driven Development* fejlesztési gyakorlatok elterjedésének. Ezek segítségével a tesztesetek, illetve a szoftverspecifikáció nyelvezete közelebb került az üzleti sepcifikáció nyelvezetéhez [19], ezzel segítve a különböző szerepkörök együttműködését.

2.3.3. Teszt automatizálási technológiák

A következőkben bemutatom a webes alkalmazások átvételi teszteléséhez használt technológiák közül a munkám szempontjából legfontosabbnak ítélteket.

WebDriver

A WebDriver [20] egy standard specifikáció, melyet a W3C konzorcium *Browser Testing and Tools Working Group* munkacsoportja fejleszt és tart karban. Fejlesztésének elsődleges célja, hogy külső folyamatokból egy nyelvfüggetlen felületet definiáljon a webböngészők programozott eléréshez. Ennek segítségével a böngészők elemzése és irányítása végezhető programozottan, így támogatva az átvételi tesztek automatizálást. A napjainkban ismert böngészők közül mindegyiknek létezik WebDriver implementációja, ami által lehetővé válik, hogy ugyanazon teszt *script*

különböző webböngészőkben (pl: Firefox, Chrome, Safari, Edge) is végrehajtható legyen.

WebDriver BiDi

A WebDriver által képzett köztes réteg a kliens és a böngésző között bizonyos esetekben körülményessé teszi az automatizálást. A kliens ugyanis a WebDriver által alkalmazott HTTP protokol miatt csak kérés-válasz stílusú kommunikációt tud folytatni a böngészővel. Ez ugyan első ránézésre életszerűnek tűnik, hiszen egy valós felhasználó is hasonló módon használja a böngészőt: rákattint egy linkre, majd a kérésre válaszként betöltődik az oldal, ahol újabb műveleteket végezhet. A valóságban, különösképpen az AJAX elterjedése óta, a felhasználás sokkal inkább aszinkron módon történik az weboldalak részleges frissítése mellett. Például a felhasználó egy terméklistát tekint át, majd némi várakozás után a kiválaszott termékre kattintva annak részletes leírása jelenik meg egy erre szolgáló mezőben. Természetesen az oldal többi része használható marad az alatt az idő alatt, amíg a felhasználó vár a tartalom egy részének frissülésére. Ennek a fajta aszinkronitásnak a kezelése az automaziálás során igen nehézkes. A WebDriver ugyanis nem teszi lehetővé a böngésző vagy az oldal állapotának ellenőrzését, így valamilyen várakoztatásra van szükség az automatizált lépések végrehajtása között, hogy a soron következő lépés már a betöltött oldalon legyen végrehajtható.

A WebDriver által nyújtott absztrakció elfedi a kliens elől a bögésző által nyújtott alacsonyabb szintű funkcionalitásokat, melyek segítségévével az automatizációs kódok írása egyszerűsödne. Gondoljunk csak a böngészőkbe épített tárakra, eseménykezelőkre, geolokációs funkcionalitásokra.

Az előbb említett két problémára adott egyik válasz a Google DevTools Protocol[4] (CDP) megalkotása volt, amellyel az automatizációs folyamatok számára lehetővé vált a böngésző fejlesztői eszközeinek elérése. A protokol magját websocket tecnhológia adja, ezáltal biztosítva a kétirányú, aszinkron kommunikációt. A név ellenére ugyanakkor a CDP nem egy egyezményes sztenderd, fejlődését elsősorban a Google Chrome böngésző befolyásolta, így a Chrome-on kívül egyedül a Firefox implementálta a protokolt részlegesen. A CDP-t kezdetben nem automatizációs célokra szánták, de néhány automatizációs és átvételi teszt-keretrendszer épít rá, például: Puppeteer, Cypress.

[4] `https://chromedevtools.github.io/devtools-protocol`

A WebDriver BiDi[5] protokoll megalkotását az az igény vezérli, hogy legyen egy sztenderizált böngésző-automatizálási eszköz, mely a megszokott WebDriver funkcionalitások mellett a CDP által biztosított eszközöket is támogatja. A fejlesztésben számos iparági szereplő vesz részt, közölük a legnagyobbak: Apple, Google, Microsoft, Mozilla. Jelenleg a fejlesztés egyelőre a prototipizálás szakaszában tart.

Selenium

A Selenium[6] egy 2011-ben megalapított projekt, mely több kisebb projektet fog össze, amik különböző módon támogatják a böngésző automatizálást. A magját WebDriver implementáció adja, amelyre számos böngésző automatizálást támogató eszköz és könyvtár épül. A WebDriver közvetlenül nem képes a böngészővel kommunikálni, ehhez szükség van a böngésző natív *driverének* telepítésére. Ugyanakkor a natív *driver* a felhasználó számára transzparens, tehát ugyanazok az instrukciók tetszőleges böngészőben futtathatóak. A WebDriver protokolnak megfelelően HTTP protokolon keresztül kommunikál a natív, API REST alapú *driverrel*. A natív *driver* tulajdonképpen egy HTTP szerver, amely a böngészőt gyártónként változó protokolokon keresztül vezérli, általában a böngésző debugger interfészét használva vagy közvetelnül DOM eseményeken keresztül. Felépítését tekintve három telepítési modell alakítható ki:

- **Lokális végrehajtás:** A WebDriver, a natív *dirver* és a meghajtott böngésző egyazon gazdarendszeren fut
- **Távoli végrehajtás:** A WebDriver a kliens gépen fut, ami a szerver gépen telepített Remote WebDriverrel kommunikál. A Remote WebDriver – a lokális telepítéssel analóg módon – a szerveren futó natív *driveren* keresztül hajtja meg a böngészőt.
- **Elosztott végrehajtás:** A Selenium Grid komponens lehetővé teszi, hogy klaszterezett módon, párhuzamosan és elosztottan több különböző böngészőt is meghajtsunk. A Grid szerver egy *hubon* fut, ami több Remote Webdriverrel (*nodeok*) kommunikál, hajtatja végre az utasításokat. Ezzel a kialakítással párhuzamosan tesztelhető az alkalmazás több különböző böngésző típussal és verzióval, továbbá lehetőséget biztosít terhelési tesztek végrehajtására is.

[5] `https://developer.chrome.com/blog/webdriver-bidi/#webdriver-bid`
[6] `https://www.selenium.dev/`

A Selenium napjainkban a legszélesebb körben alkalmazott böngésző automatizálási eszköz, használatának előnyeit és hátrányait a "Automation testing framework for web applications with Selenium WebDriver: Opportunities and threats" című tanulmányban gyűjtötték össze.

Előnyök [21]:

- Nyílt forráskódú, bővíthető architektúra.
- Főbb böngésző típusok támogatása, mint például Firefox, Google Chrome, Safari, Opera, Internet Explorer és natív mobil platformok, mint iOS és Android.
- Több operációs rendszerrel is kompatibilis, mint például Linux, Windows, OS X.
- Több nyelvű felhasználói API, így támogatja többek közt a Java, C#, Ruby, Python, JavaScript, Kotlin nyelveket, ezzel segítve a gyors adaptációt.
- Flexibilis tesztkeretrendszer integráció.
- Képernyő felvételek készítése.

Hátrányok [21]:

- Használatához magasfokú technikai kompetencia szükséges. A könyvtár ismeretén túl szükség van rá, hogy a teszteket író jártas legyen valamely – a Selenium által támmogatott – programozási nyelvben.
- Az imperatív programozási nyelvekkel körülményes a felhasználói műveletek leírása.
- Önmagában nem használható tesztek végrehajtására, egy tesztkeretrendszerbe kell beágyazni.
- A WebDriver használata költséges és időigényes, mivel a futtatás során teljes weboldalakat be kell tölteni a memóriába.
- Mivel számos kiegészítőre van szükség az önáló futtatáshot (pl: natív *dirver*, tesztkeretrendszer) a futtatható állomány mérete nagy.

Playwright

A Playwright[7] a Microsoft által fejlesztett, 2020-ban bemutatott teszt automatizálási eszköz. Kihasználja a WebDriver bemutatása óta megjelent újabb automatizálási technológiákat, így nem a Seleniummal ellentétben nem WebDriver alapú, hanem közvetlenül a böngésző motorjával kommunikál, kihasználva például a GDP nyújtotta lehetőségeket. Ugyanakkor a Selniumhoz hasonlóan támogatja a főbb böngésző típusokat és operációs rendszereket. A Playwright motorja NodeJs-re íródott, de a Seleniumhoz hasonlóan több programozási nyelvet is támogat: JavaSrcipt, Python, Java, C#.

Előnyök:

- Mobil könyezet emulálása.
- Böngésző alancsony színtű API-jának elérése, pl: geolokáció kezelés, tabok és inkognitó ablakok kezelése, hálózati folyamatok figyelése, stb.
- Automatikus várakozás az oldal betöltésre, újrapróbálkozási szabályok adhatóak meg.
- Képernyőkép és videó felvétel készíthető a végrehajtásról.
- Párhuzamosan futó tesztek teljes izolációja böngésző kontextusok segítségével. Autentikációs adatok újrafelhasználhatósága.

Hátrányok:

- Kevesebb böngészőt támogat, mint a Selenium.
- Használatához bizonyos böngésző motorokat patchelni kell.
- Viszonylag új és kevésbé elterjedt, mint a Selenium.

Robot keretrendszer

Az előbb említett eszközök elsősorban automatizációs eszközök, bár a playwright bizonyos nyelvek esetén rendelkzik beépített tesztfunkciókkal is. Szükség van tehát – egy lehetőleg egységes – teszt keretrendszerre, amelybe a *scriptek* tesztként beágyazhatóak, futthatóak.

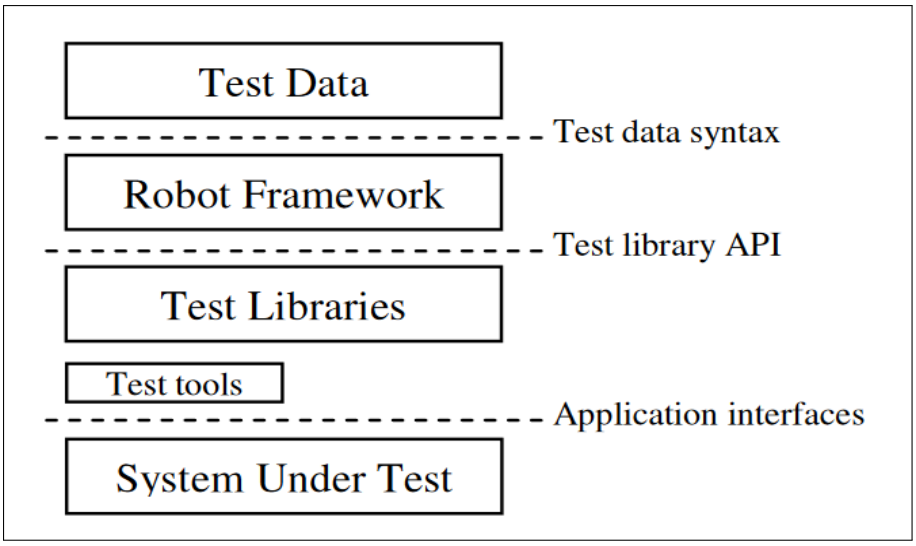
[7] `https://playwright.dev/`

További hátrányuk, hogy megkövetelik a teszt készítőjétől valamely támogatott programozási nyelv mélyebb ismeretét. Igaz, hogy ezt a magas belépési küszöböt némiképp ellensúlyozza, hogy több programnyelvet is támogatnak, ugyanakkor a több nyelv tényleges támogatása extra terhet jelent üzemeltetői oldalon, hiszen több programnyelvi környezetet kell karabantartani.

A Robot[8] egy nyílt forráskódú, kulcsszavas, általános célú tesztautomatizálási keretrendszer, amelynek magját Python nyelven írták. A Robot alapjait 2005-ben Pekka Klärck fektette le "Data-driven and keyword-driven test automation frameworks" [16] című diplomamunkájában. A fejlesztését kezdetben a Nokia Siemens Networks karolta fel, napjainkban pedig az erre a célre létrehozott Robot Framework Foundation fejleszti és tartja karban. Rendelkezik saját fejlesztői környezettel, melyet RIDE névre kereszteltek. A Robot architektúrájának egy magaszintű leírása a 2.9 ábrán látható. Az egyes komponensek a következő funkciókat látják el:

- **Test Data:** Teszeset definíciók, tartalmazzák a lépéseket, argumentumokat és validációkat. Szintaxisuk kötött, táblázatos elrendezésű, kulcsszó alapú.
- **Robot Framework:** A rendszer alapja, mely fordítja és futtatja a teszteseteket, betölti a teszthez szükséges adatokat, és a teszteredmények alapján riportot generál. Rendelkezik egy alap kulcsszó készlettel és könyvtárakkal, ugyanakkor nem rendelkezik a SUT-re vontakozó speciális tudással.
- **Test Libraries:** Ez a réteg biztosítja a keretrendszer bővíthetőségét. A könyvtárak rendelkeznek a megfelelő interfészekkel és protokoll ismeretekkel, hogy meghajtsák a SUT-t. A robot framework a tesztkönyvtárak által biztosított egyésges API-kon keresztül tudja leképezni a kulcsszavas formában adott tesztlépéseket a könyvtár metódusaira. A test library gyakran valamilyen önállóan is használható teszteszközt (*Test tools*) csomagol, így lehetőség van például Seleniumot vagy Playwrightot használni a Robot keretein belül.
- **System Under Test**: a tesztelendő rendszer maga.

[8] `https://robotframework.org/`



2.9. ábra. Robot framework magasszintű architektúrája [22]

Előnyök [22]:

- A tesztesetek leírása tömör és egységes formátumú.
- Újrafelhasználható, bővíthető kulcsszó készlet.
- A teszt platform és a tesztelt alkalmazás (*Application Under Test*) izolált.
- Bővíthető egyedi könyvtárakkal.
- A tesztesetek címkézhetőek, így lehetőség van osztályozni őket, illetve szelektív végrehajtást kérni.

2.3.4. Összefoglalás

A fejlesztés során figyelembe kell venni, hogy az oktatók eltérő programnyelvi ismeretekkel rendelkeznek. Feltételezhető az is, hogy kevéssé jártasak a megbízható átvételi tesztek fejlesztésében, amely, mint láttuk, nagy körültekintést igényel. A megvizsgált technológiák közül egyértelműen a Selenium rendelkezik a legszélesebb körű felhasználási lehetőséggel, továbbá architektúrájának köszönthetően igen robosztus tesztrendszer építhető Selenium alapokon. Ugyanakkor azt gondolom, hogy webes programozási beadandók ellenőrzésénél – ellentétben az üzleti célú felhasználással – a különböző böngészőtípusok és verziók támogatása nem kimondottan fontos, a hangsúlyt inkább a funkcionalitás tesztelhetőségére kell helyezni. A Seleniummal szemben a Playwright fel van készítve a modern webalkalmazások tesztelésével járó

kihívásokra (például implicit várakoztatás, újrapróbálkozás), így leveszi az oktatók válláról a technikai problémák kezelésének egy részét, akik ezáltal az üzleti logika ellenőrzésére fókuszálhatnak. A Robot keretrendszert ideális választásnak tartom, mivel a kulcsszóalapító tesztek készítése rövid és tömör megfogalmazást tesz lehetővé. A bővíthető architektúra pedig lehetőséget biztosít a TMS fejlesztői számára, hogy az oktatók munkáját megkönnyítve újabb könyvtárakkal, kulcsszavakkal bővítsék a rendszert.

3. fejezet

Követelményelemzés

Ebben a fejezetben feltárom a címben megfogalmazott célokat megvalósító rendszerkövetelményeket. Először is röviden bemutatom a magas szintű, kontextusba helyezett követelményeket, majd a soron következő alfejezetekben részletesen kifejtem a funkcionális és nem-funkcionális követelményeket.

A TMS rendszer felhasználói oktatók, hallgatók és adminisztrátorok. Az oktatók egy tárgy kurzusának keretében beadandó feladatokat írhatnak ki a kurzus hallgatói számára [23, 24]. A rendszerben automatizálható programozási beadandók tesztelése, amihez az oktatónak definiálni kell a futtatási környezetet és a teszteseteket. A hallgató a programozási beadandókat a forrásfájlok feltöltésével, vagy verziókezelő rendszeren keresztül adhatja be (Git) [25]. A rendszer jelenleg a TMS parancssoros alkalmazások tesztelésére képes [26].

A hallgatók általában felsőbb évfolyamok kurzusain találkoznak web-technológiákon alapuló fejlesztési feladatokkal. Ezeken a kurzusokon a beadandók elkészítése általában nagyobb technikai felkészültséget igényel a hallgatótól: meg kell ismerkedniük valamely web-fejlesztési keretrendszerrel, legalább alapfokú számítógépes hálózati tudással kell rendelkezniük és architekturális tervezési mintákat kell alkalmazniuk, illetve akár adatbázist is kell kezelniük.

Az ilyen beadandók értékelése sokrétű, nem kizárólag a program helyességre koncentrál. Az értékelési szempontok a funkcionális követelmények validációja mellett kiegészülhet a program nem funkcionális aspektusainak ellenőrzésével, mint például a konfigurálhatóság, a biztonság, a robusztusság, a hordozhatóság, vagy a skálázhatóság. Emellett hangsúlyosabbá válhatnak a kód minőségre vonatkozó szempontok: tervezési paradigmák követése, formázási szabályok követése, teszt lefedettség,

dokumentáció. Az oktatóknak tehát huzamosabb időt kell eltölteniük egy-egy beadandó ellenőrzésével, ami lassítja az értékelési folyamatot, így a hallgató lassabban jut visszacsatoláshoz. Ráadásul az összetett értékelési szempontok akadályozhatják a konzisztens pontozást. További nehézséget jelenthet, hogy az oktatónak komplex tesztelési környezetet kell kialakítania a saját gépén, ráadásul számításba kell venniük egy-egy beadandó fordítási, telepítési idejét, illetve a környezetet két beadandó ellenőrzése között vissza kell állítani. Ezek a kihívások egyszerűbb beadandó feladatok kiadására, illetve az egyes értékelési szempontokat mellőzésére sarkallhatják az oktatókat.

Hogyan lehet a webes beadandók értékelését gyorsabbá, egyszerűbbé és megbízhatóbbá tenni? Az lehet a megoldás, ha az előbb említett oktatói értékelési lépések közül a lehető legtöbbet automatizáljuk a TMS keretein belül. Erre támaszkodva az oktató megszabadul a mechanikusabb ellenőrzési lépések végrehajtásának terhétől, így idejét a magasabb hozzáadott értékű szempontok értékelésére fordíthatja. A hallgató szinte azonnali visszajelzést kaphat a legtriviálisabb hibákról, mint amilyen például a nem futtatható programkód, vagy a nem teljesített funkcionális követelmények. Az automatizált ellenőrzések által generált riportok pedig nemcsak az oktatói értékelést segítik, de a hallgató számára is iránymutatást adhatnak a hibák javításához. Az automatizált visszacsatolások arra ösztönzik a hallgatókat, hogy megoldásaikat gyakrabban, kisebb inkrementumokban töltsék fel [1].

A főbb követelmények tehát a következők:

- Az oktatóknak legyen lehetősége a hallgatói beadandókat egy távoli környezetben futtatni.
- Az oktatóknak legyen lehetősége a beadandó beadásakor automatikusan futó átvételi tesztek definiálására.
- A hallgató kapjon azonnali, kellő részletezettségű visszajelzést az automatizált lépések eredményéről, hogy az esetleges hibákat időben javíthassa.

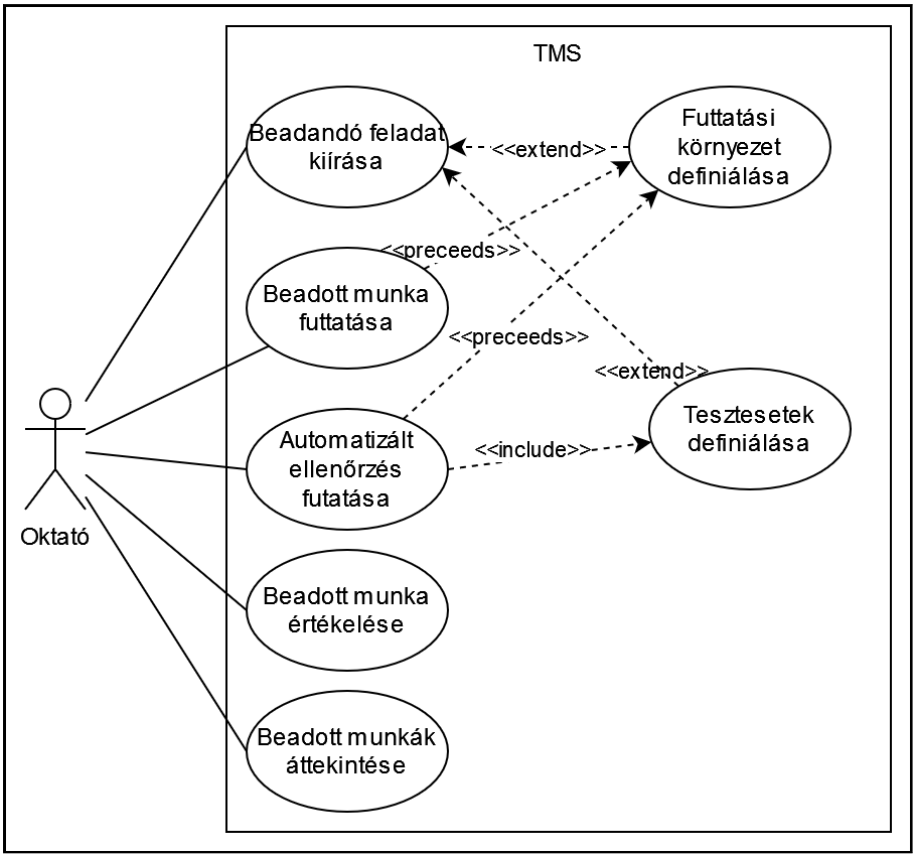
Tekintve, hogy a TMS beadandó kezelési funkciói már adottak, az új funkciók megvalósítása során törekedni kell arra, hogy azok illeszkedjenek a már meglévő elemekhez.

3.1. Funkcionális követelmények

A részletes funkcionális követelményeket a felhasználói szerepkörök szerinti bontásban ismertetem.

3.1.1. Oktatói funkciók

A 3.1 használati-eset diagram szemlélteti az oktatói funkciókat.



3.1. ábra. Oktatói használati-eset diagram

Beadandó feladat kiírása

A feladat kiírásának módja változatlan marad, de az automatikus ellenőrzőnél az oktatónak lehetősége van választani, hogy parancssoros vagy webes futtatási környezetet akar definiálni. A webes környezet alapját a parancssoroshoz hasonlóan egy Docker kép adja, azonban mivel a konténerben futó webszervert távolról is elérhetővé kell tenni, ezért az oktatónak meg kell adnia a webszerver port-számát, amihez a külső portot kötni kell. Az oktatónak a parancssoros alkalmazásokhoz hasonlóan lehetősége van feltölteni oktatói fájlokat, továbbá meg tud adni fordítási és futtatási

parancsokat is, azonban webalkalmazások esetén a fordítási és futtatási parancsok megadása nem kötelező. Ugyanakkor a parancssoros alkalmazásoknál, ahol a futtatási parancs minden tesztesetnél végrehajtásra kerül, a webalkalmazások esetében a futtatási parancs célja a webalkalmazás elindítása (pl: szerver indítása, alkalmazás telepítése).

Mivel a webalkalmazás-fejlesztési kurzusok gyakran használnak adatbázis technológiát, ezért az oktatónak lehetősége van egy másodlagos, adatbázist és tesztadatokat tartalmazzó Docker képet is megadni. A webszervert tartalmazó Docker képhez hasonlóan itt is meg kell adni az adatbázis szerverportját annak érdekében, hogy a futási környezet kialakításakor a konténerek közötti hálózati kapcsolatot fel lehessen építeni. A teszteseteket a parancssoros alkalmazások esetén input és output párok alkotják. A webes alkalmazások ellenőrzéséhez szükséges átvételi tesztek definiálása komplexebb feladat, mivel egy-egy teszt számos tesztlépésből állhat. Mielőtt az ellenőrzés megtörténhetne, az előfeltételek szerint meghatározott állapotba kell léptetni a rendszert.

A parancssoros alkalmazásoknál az oktató beállíthatja, hogy a hallgató ne lássa a teszt futtatás részletes hibaüzeneteit, ami meggátolja, hogy a hallgató beégetett értékeket használjon. Ezt a megközelítés csak korlátozottan alkalmazható a webalkalmazások esetén, mert a hibás tesztek reprodukálásához a hallgatónak szüksége van a hibaüzenetben lévő információra. Egy lehetséges megoldás, hogy az oktató két részre bontja a teszteseteket. Az egyik halmaz részletes eredményei láthatóak a hallgató számára, míg a másikéi nem. Ideális helyzetet lehet teremteni a hallgató és az oktató számára is, ha a két halmaz tesztesetei csak attribútumaikban különböznek egymástól. A hallgató a látható halmaz teszteredményei alapján ki tudja javítani a hibákat oly módon, hogy ezzel a nem látható halmaz tesztesetei is működőképessé váljanak. Ugyanakkor ha a hallgató beégetett értékek megadásával próbálkozna a második teszteset halmaz elbukna, hiszen azok más be- és kimeneti értékekkel működnek.

Az oktatók tehát két darab, az összes tesztesetet leíró fájlt tudnak feltölteni. Az egyik fájl futtatásának eredményei a hallgató számára elérhetőek a másik részletes eredményeit csak az oktató látja - ez utóbbi feltöltése lehet opcionális. Az oktatói munkát segítendő a TMS-ből letölthető egy sablon, amelybe a teszteseteket kell leírni: a kitöltött sablon visszatöltésekor a TMS ellenőrzi a fájl szintaktikai helyességét. További támogatást jelenthet az oktatók számára, hogy a TMS fejlesztői a Robot

keretrendszert további kulcsszavakkal tudják bővíteni, amelyet az oktatók a teszt írásakor fel tudnak használni.

Beadott munka futtatása

Az oktatónak lehetősége van a hallgatók által feltöltött megoldást elindítani és megtekinteni. Az alkalmazás sikeres elindítását követően az oktató megkapja az alkalmazás eléréséhez használható URL-t. Az alkalmazás megtekintéséhez szintén a TMS által generált jelszót kell megadni.

Annak érdekében, hogy az oktató össze tudja hasonlítani a hallgatók beadandóit, párhuzamosan több példányt is elindíthat, de egyazon hallgató beadandójából nem indíthat el több példányt.

Az oktató manuálisan is leállíthatja az elindított alkalmazásokat, de a rendszererőforrások gazdaságos kihasználása érdekében egy konfigurálható időintervallum után automatikusan leállításra kerülnek.

Amennyiben egy beadandó elindítása fordítási hiba miatt sikertelen, az oktató és a hallgató erről értesítést kap, és a hiba javításáig a beadandót, az oktató nem tudja futtatni. Az automatikus tesztelés során fordítási hibával megbukó alkalmazásokat az oktató szintén nem tudja elindítani addig, amíg a hallgató nem tölt fel újabb megoldást.

Automatizált ellenőrzés futtatása

Az automatizált ellenőrzések a parancssoros alkalmazásokéhoz hasonló módon ütemezetten futnak. Webalkalmazások esetén az oktatónak lehetősége van a még nem értékelt alkalmazás automatizált tesztjeit megfuttatni, hogy az eredményeket az értékelés során fel tudja használni.

Beadott munka értékelése

A beadott munka értékelésének módja nem különbözik a parancssoros alkalmazások értékelésétől.

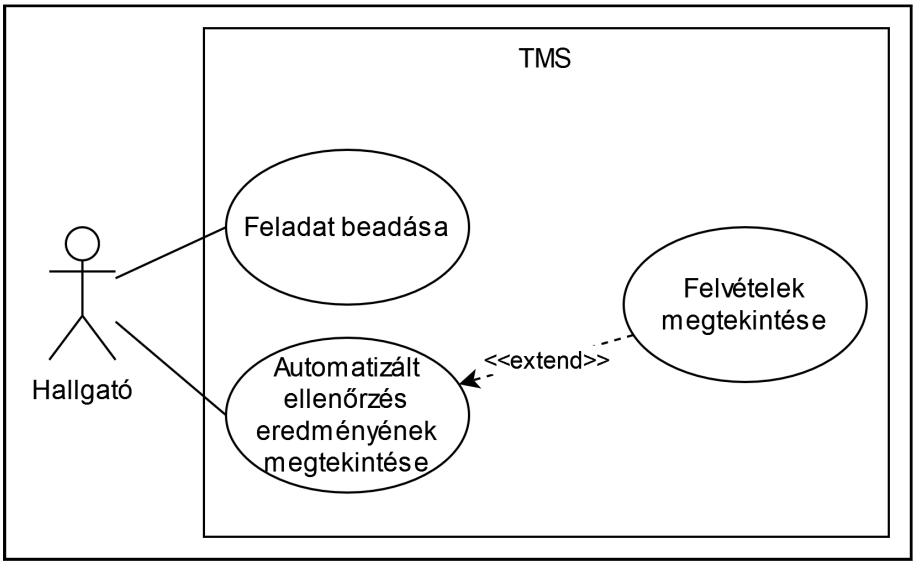
Beadott munkák áttekintése

A beadott munkákat az oktató a parancssoros alkalmazásokéhoz hasonló módon listázhatja ki. Webalkalmazások esetén azonban megjelenik a legutolsó fordítás

eredménye is, hogy az oktató lássa, mely alkalmazásokat nem tudja távoli futtatásra kiválasztani.

3.1.2. Hallgatói funkciók

A 3.2 használati-eset diagram szemlélteti a hallgatói funkciókat.



3.2. ábra. Hallgatói használati-eset diagram

Feladat beadása

A programozási feladatok beadása a jelenlegi megvalósítással azonos módon történik. A hallgató egy tömörített állományban tudja feltölteni a forráskódot, vagy közvetlenül tud kapcsolódni egy verziókezelőben tárolt projekthez. A tömörített állományok felöltésénél figyelemmel kell lenni a feltölthető fájl maximális mértére, ugyanis webalkalmazások esetén a hallgató projektje akár nagyobb méretű fájlokat is tartalmazhat (pl: kép).

Automatizált ellenőrzés eredményének megtekintése

A 3.1.1 alfejezetben már röviden említettem, hogy a webalkalmazások esetében miért van szükség arra, hogy a hallgató legalább az esetek egy részében láthassa a tesztek részletes eredményeit. Előfordulhat, hogy hallgató programja funkcionálisan helyes, azonban a teszt elbukik, mert nem képes a megtalálni a keresett HTML elemet; például akkor, ha a hallgató elgépeli egy elem azonosítóját, vagy rossz CSS

attribútumot használ. Segítség nélkül ezeknek a hibáknak a felderítése szinte lehetetlen. Magasszintű elemzéshez a riportok és napló fájlok megfelelőek. Utóbbi tesztesetenként minden lépés állapota nyomon követhető a HTML állományban, így a hallgató láthatja, hogy pontosan melyik lépés végrehajtása volt sikertelen kulcsszó-, bemenet-, predikátum-, hibaüzenet-szinten. Ennél részletesebb információk is gyűjthetőek a tesztek végrehajtása során:

1. **Nyomkövetés:** Segítségével lépésről lépésre nyomon követhető, hogy az adott oldalon az automatizált lépések a DOM mely elemeivel léptek kapcsolatba, milyen műveleteket végeztek, stb. Akár a DOM pillanatnyi állapota, a sütik és gyorsítótárak állapota is lementhető.

   Előnye, hogy a fájlok mérete kicsi, de elég részletes információt szolgáltatnak a teszteset reproukálásához. Hátránya, hogy értelmezésük kíván némi szakértelmet.

2. **Képernyőkép:** Képernyőkép készíthető a végrehajtás bármely kiválasztott pontján vagy automatikusan minden hibánál.

   Előnye, hogy segítségével könnyű kontextusba helyezni a hibát. Hátránya, hogy sok képernyőkép készítése tárhelyigényes és növeli a teszt futási idejét.

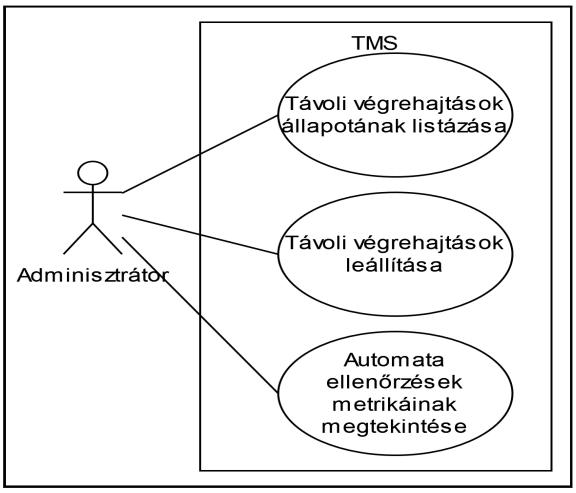
3. **Videófelvétel:** A teljes teszt végrehajtásának felvétele.

   Előnye, hogy lépésről-lépésre azonnal értelmezhető a teszt futása. Hátránya, hogy jelentősen lassítja a végrehajtást, a videók tárolása sok tárhelyet igényel. Hátránya még, hogy csak a látható lépések nyomonkövetése lehetséges, az olyan implicit műveletek, mint például a sütik vagy gyorsítótárak kezelése viszont nem.

Hathatós hallgatói támogatást az előbb felsorolt eszközök kombinációjával lehet adni. A megfelelő kombináció kialakítása azonban minden bizonnyal a hallgatói visszajelzéseken alapuló iteratív folyamat. Kezdetben a nyomkövetés és a hibás lépésekről készített képernyőképek érhetőek el a hallgatók számára. Utóbbi ugyan lassítja a tesztek végrehajtását, mivel a böngészőt grafikus felhasználói interfésszel kell futtatni, de kézzelfogható bizonyítékot szolgáltat a felmerült hibáról. A hallgatók számára a teszteredmények elérhetőek a TMS-ben.

3.1.3. Adminisztrátori funkciók

A 3.3 használati-eset diagram szemlélteti a adminisztrátor funkciókat.



3.3. ábra. Adminisztrátori használati-eset diagram

Mind a távoli végrehajtás, mind pedig az automatizált tesztek futása jelentős hardver erőforrásokat igényelnek. A végrehajtás során felléphetnek olyan hibák, melyek beragadt vagy elárvult folyamatokat eredményezhetnek. Az adminisztrátor elsődleges felelőssége ezeknek a hibáknak a kezelése, a gazdaságos erőforráskihasználás biztosítása.

Távoli végrehajtások állapotának listázása

A tesztekkel ellentétben a távoli végrehajtások huzamosabb ideig is futhatnak. Több alkalmazás egyidejű futtatása jelentősen terheli az erőforrásokat, vagy akár meg is gátolhatja az oktatókat abban, hogy további alkalmazásokat indítsanak el. Ennek nyomonkövetésére az adminisztrátornak lehetőséget kell biztosítani arra, hogy listázza a futó alkalmazásokat és elérje a legfontosabb információkat: ki indította, mikor indította, mikorra van ütemezve a leállítása, illetve az alkalmazás mennyit használ a főbb erőforrásokból. Az információforrás a TMS adatbázisa és a Docker hostok. Utóbbira azért van szükség, hogy a valamilyen hiba során elárvult konténereket is nyilvántarthassuk.

Távoli végrehajtások leállítása

Az adminisztrátornak lehetősége van a kiválasztott vagy az összes távoli végrehajtás leállítására, ezáltal az erőforrások felszabadítására.

Automata ellenőrzések metrikájának megtekintése

Az automatizált tesztek végrehajtása erőforrás- és időigényes. A tesztek futtatásához több Docker konténerre és hálózatra is szükség lehet, az átvételi teszt lépéseinek végrehajtása jelentős időt igényel. A végrehajtási idő természetesen függ a tesztek, illetve a bennük lévő lépések számától. Adminisztrátori szinten ilyen részletezettségű adatokra nincsen szükség, de feladatszinten szükség van a kumulált átlagos végrehajtási idők és az erőforrásfelhasználás nyomonkövetésére, így azonosíthatóvá válnak a rendszert kiemelkedő mértékben terhelő beadandók és tesztcsomagok.

3.2. Nem-funkcionális követelmények

3.2.1. Biztonság

A rendszer biztonságosságának megtervezésénél különleges szempontokat kell figyelembe venni, ugyanis mind a távoli futtatás, mind az automatizált tesztek végrehajtása azzal jár, hogy a hallgató kódját kell futtatni az ELTE szerverein. Az ilyen kódot nem szabad megbízhatónak tekinteni, ugyanakkor ezekben az esetekben a szokásos védelemi mechanizmusok egy jó része nem alkalmazható, pl. a tűzfal szabályok vagy a telepítési és végrehajtási korlátozások. Természetesen nem csak a szándékos károkozás kockázatát kell csökkenteni, mert előfordulhat, hogy a beadandóba olyan hiba kerül, amelynek végrehajtása okozza a kárt, vagy a hallgató valamilyen nem biztonságos függőséget használ. A beadandó távoli végrehajtásának támadási felületei 2 főbb kategóriába oszthatóak. Egyrészt a támadás érheti a szervereket belülről, ha a futtatandó alkalmazás rosszindulatú kódot tartalmaz. Másrészt a támadás kívülről is érkezhet, abban az esetben, ha az oktató számára a távoli végrehajtás során az alkalmazás elérhetővé válik publikus URL-n. Különösen fontos kiemelni, hogy az alkalmazás akár olyan szerveren is futhat, melynek nincs közvetlen internetelérése. A következő két alfejezetben ezen támadási vektorok kivédésével foglalkozom.

Biztonságos végrehajtási környezet

Az alkalmazás biztonságos végrehajtásához a környezetnek a következő követelményeknek kell megfelelnie.

- A folyamatot futtató felhasználó nem rendelkezhet olyan privilégiumokkal, mellyel irányítást szerezhet az operációs rendszer fölött, nem telepíthet további alkalmazásokat.
- Az alkalmazás nem hozhat létre újabb felhasználót és nem férhet hozzá a root felhasználói szerepkörhöz.
- Az alkalmazás számára csak a fájlrendszer egy számára dedikált része érhető el, mellyel nem osztozhat más folyamatokkal.
- A memória, processzor és egyéb rendszer erőforrás felhasználása kvótához kötött, melyek elérése a folyamat leállításával vagy korlátozásával kell járjon.
- A teljes futási időt, és az egyes végrehajtási lépések (pl. fordítás, konténerek indítása) végrehajtási idejét maximálni kell, ennek elérésekor a folyamatokat le kell állítani és minden állományt fel kell szabadítani.
- A konténerek csak az előre meghatározott hálózati erőforrásokat érhetik el, ide értve más konténerekkel történő kommunikációt.
- A Docker képek ellenőrzött forrásból származzanak.
- A konténereket vezérlő folyamat (Docker deamon) csak megbízható felhasználók érhetik el.

Biztonságos távoli hozzáférés

Az oktatók által távoli végrehajtásra kiválasztott beadandó manuális teszteléséhez hálózati elérést kell biztosítani a konténerben futó webalkalmazáshoz. Garantálni kell az oktató kizárólagos hozzáférését a futó beadandóhoz. Egy lehetséges megoldás lehetne, ha az alkalmazás csak az egyetemi hálózatról vagy VPN-ről lenne elérhető. Bár ez elég biztonságos megoldás, de nehézkessé teszi az oktató számára a tesztelést, illetve nem biztosítja, hogy az adott hálózaton dolgozó más felhasználók ne férhessenek hozzá az alkalmazáshoz. Éppen ezért egy olyan megoldást kell kialakítani, aminek segítségével az oktató bárhonnan, biztonságosan hozzáférhet az alkalmazáshoz. Az alkalmazás elérését ezért kézenfekvő jelszóhoz kell kötni. A jelszó nem lehet az oktató TMS-ben használt jelszava, ezért egyszer használatos, generált jelszavakat kell alkalmazni (OTP).

Skálázhatóság

A konténerek futtatása erőforrásigényes, egy futtatási környezethez pedig több konténer is tartozhat (pl. webszerver és adatbázis). A tanév során a rendszer terhelése ingadozó. Az oktatók által indított távoli végrehajtások száma feltehetően nem jelent számottevő terhelést a rendszernek, azonban az automatizált tesztek futtatása a tömeges beadások időszakában komoly terhelési ugrást jelent. A beadandó határidejéhez közeledve a beadások száma feltehetően folyamatosan nő, majd a határidő után visszaesik. Maximális terhelésre zárthelyi időszakban, illetve szorgalmi időszak végén lehet számítani.

Ez a jellegzetesség szükségessé teszi a rendszer horizontális skálázását, azonban - mivel a kritikus rendszerterheltség időszakai jól tervezhetőek - dinamikus skálázásra nincs szükség.

4. fejezet

Rendszerterv

Ebben a fejezetben egy magasszintű rendszerterv részletességével bemutatom, hogy a követelmény elemzés során meghatározott célok miként kerülnek megvalósításra a TMS-ben. Először is bemutatom a funkcionális követelmények megvalósításának tervét, majd pedig az biztonságot és skálázhatóságát támogató elemeket ismertetem.

Prototípus

A diplomamunka keretében elkészült egy prototípus[a] is, melynek elsődleges célja a funkcionális követelmények megvalósíthatóságának demonstrálása. Ez a prototípus a következőkben bemutatott rendszertervhez képest lényegesen egyszerűbb, bizonyos implementációs részletek megvalósításától eltekintettem. Ezeket az eltéréseket keretes írásban feltüntetem a vonatkozó részeken.

[a] `https://gitlab.com/tms-elte`

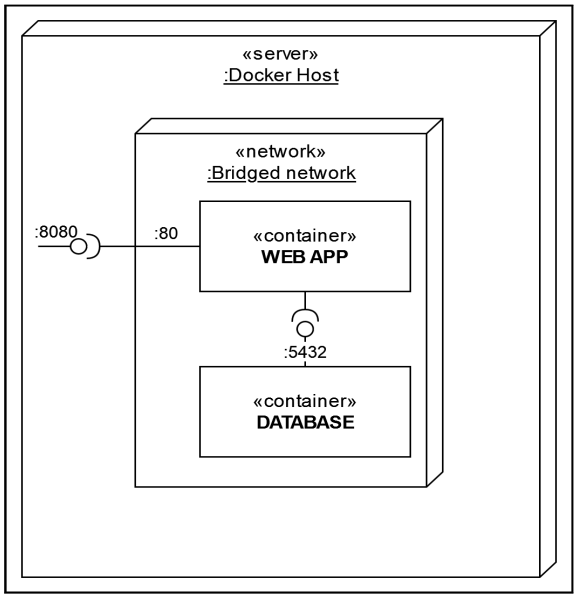
4.1. Webalkalmazások távoli futtatása

A webalkalmazásokat a konzolos alkalmazásokhoz hasonlóan Docker konténerben futtatja a TMS. Ezek konfigurációja a konzolos alkalmazásokkal analóg módon történik, de további beállításokra van szükség. A konténerben egy az alkalmazás típusának megfelelő webszerver kap helyet, ennek minimális konfigurációja a port szám, melyen a szerver a beérkező kéréseket kiszolgálja. Annak érdekében, hogy a konténerben futó webszerver a külvilág számára is elérhető legyen, a konténer hálózati konfigurációját el kell végezni. A Docker számos hálózati üzemmódot kínál, a felvázolt használati eseteknek az úgynevezett hálózati híd üzemmód felel meg.

Ennek lényege, hogy a konténer egy virtuális hálózatban fut, amit egy hálózati híd köt össze a külvilággal. A konténer hálózati konfigurációjának két fontos paramétere van:

- **Konténer port:** A konténer létrehozásakor (vagy a Docker kép definíciójakor) meg kell adni azokat a portokat, melyek a virtuális hálózat felé publikusak.
- **Port kötés:** A virtuális hálózaton publikált portok a virtuális hálózaton kívülről nem elérhetőek közvetlenül. Szükség van egy port kötésre, mely a gazdagép egy szabad portját köti a publikált porthoz.

Az oktatónak a feladat létrehozásakor definiálnia kell a publikálandó webszerver portot. A TMS a port kötéshez a gazdagépen használható, nyitott portokat egy konfigurációs paraméterből olvassa. Egy webalkalmazás telepítési diagramját a 4.1 ábra szemlélteti.



4.1. ábra. Webalkalmazás telepítési diagram

A hálózatban két konténer fut: egy adatbázis és egy webszerver. Az adatbázis a külvilág számára nem elérhető, de a virtuális hálózaton belül a másik konténer számára a 5432-es porton elérhető. A webalkalmazás 80-as portja a gazda gép 8080-as portjához van kötve, így minden a gazdagép 8080-as portjára érkező kérés a webszerver irányába kerül továbbításra.

A kialakítás előnye, hogy párhuzamosan több azonos környezet is futtatható egy gépen mindaddig, amíg a gazdagépen van kioszthatő port, a virtuális hálózaton belü-

li konfiguráció pedig rejtve marad. Hátránya, hogy a virtuális hálózat által implikált extra hálózati interfész lassítja a kapcsolatot.

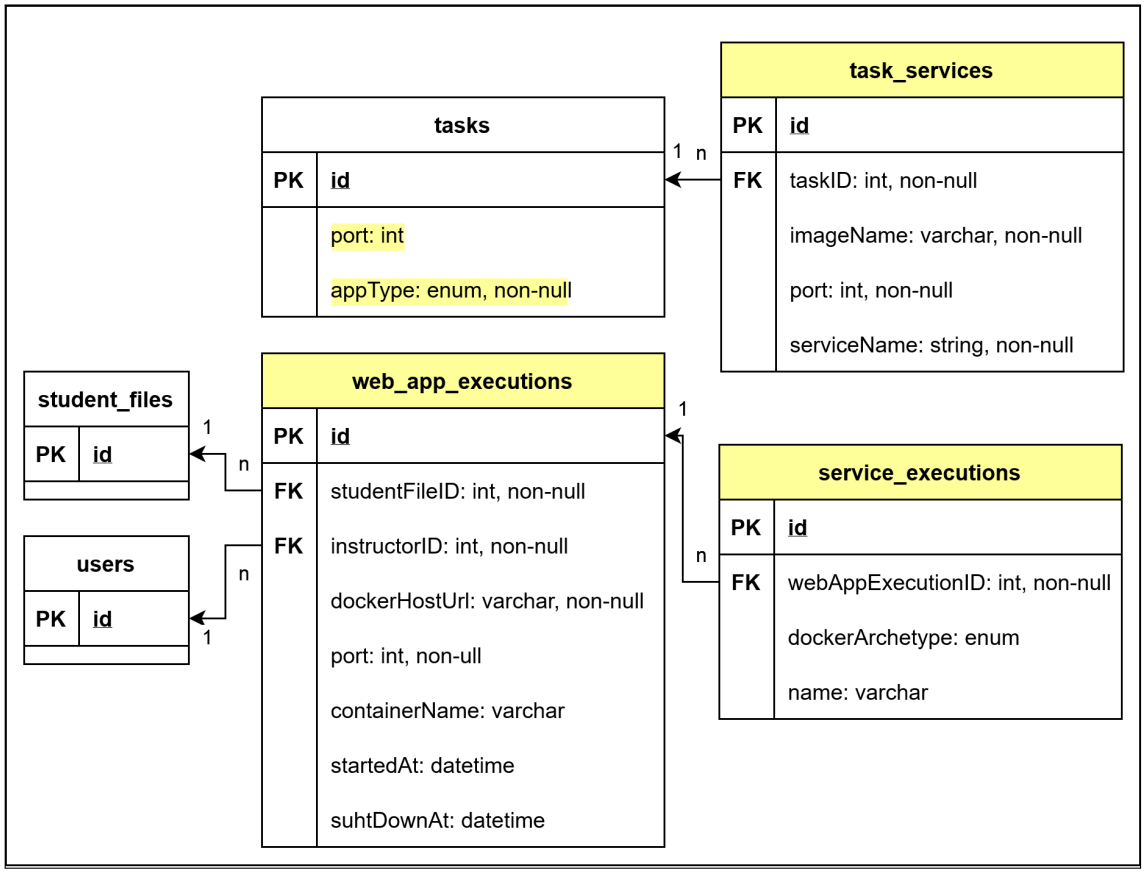
A futtatási és konfigurációs módok eltérése miatt a TMS-ben meg kell különböztetni a feladat típusokat, aszerint hogy konzolos vagy webes alkalmazást futtatnak.

4.1.1. Adatmodell

A TMS feladatokat leíró adatmodelljébe jól illeszthető a webalkalmazások környezetének konfigurációja. Az oktató által elindított alkalmazást a TMS-nek nyilván kell tartania annak érdekében, hogy azokat leállíthassa, illetve a foglalt portokat rögzítse.

A követelményelemzés során megfogalmazott igény szerint az oktatónak lehetőséget kell biztosítani, hogy a webszerver mellett a futtatási környezetben további szolgáltatásokat (pl. adatbázis) hozhasson létre. Ezek definíciója szintén egy új táblát igényel.

Az adatmodell releváns részét az 4.2 ábra szemlélteti, sárga színnel kiemelve a mozdításokat.



4.2. ábra. Adatmodell

Az alkalmazás típusát és webszerver port számát a `tasks` tábla tárolja, ebben a táblában tehát a webszerver konténer konfigurációja található, a környezetet alkotó további szolgáltatások definícióját a `tasks_services` tábla tartalmazza. Az oktató által elindított környezetről a `web_app_executions` és a `service_executions` táblák szolgáltatnak információt. A táblák részletesebb magyarázatát a 4.2 táblázat tartalmazza.

Tábla	
tasks	
Mező név	**Leírás**
port	A konténerben futó webszerver port száma.
appType	Az alkalmazás típusa: Web, Console.
task_services	
Mező név	**Leírás**
taskID	A feladat, amihez a szervíz tartozik.
imageName	A szervízt tartalmazó Docker kép neve.
port	A szervíz port száma, amin keresztül az webalkalmazással kommunikál.
serviceName	A szervíz tetszőleges neve, pl. database, messageQueue.
web_app_executions	
Mező név	**Leírás**
studentFileID	A környezetben futó beadandó azonosítója.
instructorID	A környezetet létrehozó oktató azonosítója.
dockerHostUrl	A Docker Hostot kiszolgáló szerver URL-je.
port	A szerver port száma, amin a webszerver porthoz van kötve.
startedAt	A környezet indításának ideje.
shutDownAt	A környezet leállításának időpontja. Az indítási és futtatási időből számítva.
service_executions	
Mező név	**Leírás**
webAppExecutionID	Mely webalkalmazás környezethez tartozik a komponens.
dockerArchetype	A Docker komponens típusa, pl. container, network.
name	A Docker komponens azonosítója.

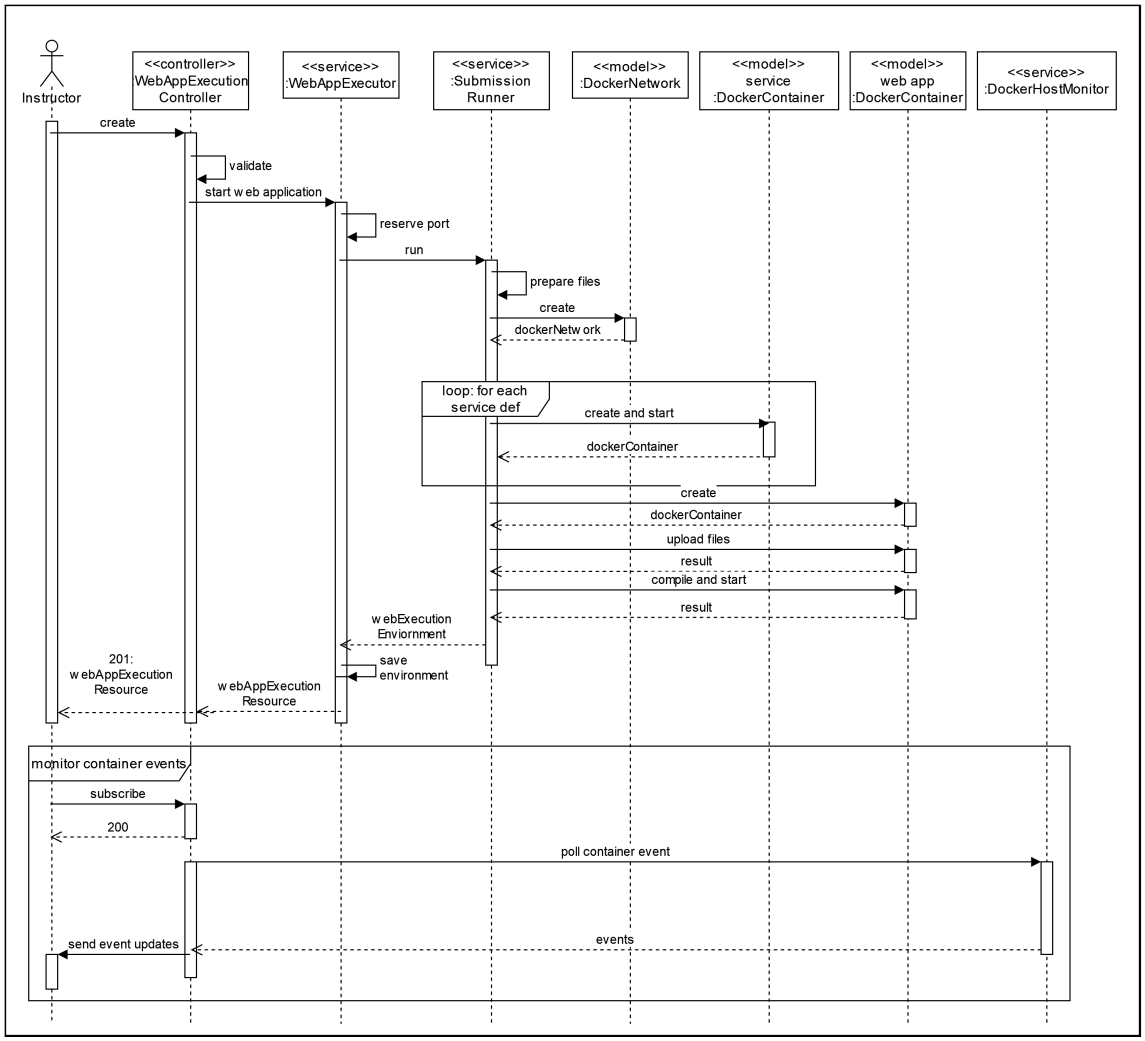
4.2. táblázat. Adatmodell mezők

Adatmodell megvalósítása

A prototípusban csak a webalkalmazás konténer konfigurációja készült el, így `tasks_services` és `service_executions` táblákat nem készítettem el.

4.1.2. Backend komponensek

A feladat létrehozása a konzolos alkalmazásokéval analóg módon történik, csak néhány – az 4.1.1 szakaszban bemutatott – további paraméter megadására van szükség. A beadandók távoli futtatása során egy teljes környezet kerül kialakításra a Docker komponensekből a feladat és szervíz definíciók alapján. A futtatás eredményeképpen az oktató egy URL-t kap válaszul a szervertől, melyen az alkalmazás elérhető. Az alkalmazás indítás szekvencia diagramját a 4.3 diagram szemlélteti.



4.3. ábra. Webalkalmazás futtatásának szekvencia diagramja

Az oktató kiválasztja a megfelelő beadandót a futtatásra, illetve megadja a környezet futtatásának időtartamát. Az időtartam lejárta után a szerver automatikusan felszámolja a kialakított környezetet. A szerverre érkező kérést a `WebAppExecutioncontroller` kontroller szolgálja ki, mely ellenőrzi a jogosultságokat, és validálja a kérést, majd továbbítja azt a `WebAppExecutor` szervíznek.

Ez az osztály további ellenőrzéseket végez, majd megpróbál egy szabad host portot foglalni az webalkalmazás számára. Amennyiben a port foglalása sikeres létrehozza a környezetet a `SubmissionRunner` szervíz segítségével. Ennek a szervíznek a felelőssége, hogy az oktató által definiált környezet komponenseit létrehozza `DockerNetwork` és `DockerContainer`. A webszervert tartalmazó konténer létrehozása után a `SubmissionRunner` feltölti a hallgatói és oktatói állományokat a konténerbe és végrehajtattja a konténerrel a fordítási és futtatási parancsokat. Ha minden sikeres volt a `WebAppExecutor` az adatbázisban tárolja környezet információit és `WebAppExecutionController` visszatér a végrehajtás eredményével, benne a webalkalmazásra mutató URL-lel.

Webalkalmazás futtatás megvalósítása

Mivel a webszervert leíró konténeren kívül további szolgáltatásokat az oktató nem tud konfigurálni és a végrehajtás azon lépései, melyek a hálóztatot és a szervízeket hozzák létre nem készültek el. A végrehajtás csak egy konténert hoz létre, mely a webszervert tartalmazza, benne a hallgató megoldásával.

Az alkalmazás leállításakor, mely történhet a kliens vagy szerver oldalról is, a Docker erőforrásokat leállítja és törölni a TMS, majd a `web_app_executions` tábla megfelelő sorait is törli, ezzel felszabadítva a lefoglalt portot.

Konténerek állapotának felügyelése

A konténerek elindulását követően a benne futó alkalmazások nem válnak azonnal elérhetővé, így előfordulhat, hogy az oktató bár már visszakapja az URL-t a környezet kialakítását követően, de a benne futó szerver még nem elérhető. Komplexebb környezetek kialakításánál ez több okból is problémát jelenthet. Egyrészt az egyes konténerekben futó szolgáltatások függhetnek egymástól, így az indításuk meghatározott sorrendben kell, hogy történjen és meg kell várni, míg a szolgáltatások elérhetővé válnak. Másrészt az oktató rákényszerülhet, hogy akár percekig is frissítgesse a webalkalmazás URL-jét, amíg az betölt. A megoldás az, hogy minden szervíz esetén definiálni kell egy úgynevezett *health check* parancsot. Ezt a konténer létrehozásakor lehet megadni, és a Docker deamon rendszeres időközönként felügyelje a parancs végrehajtását a konténerben, ennek státuszát pedig frissíti a konténer att-

ribútumai között. Ilyen parancs lehet például egy HTTP GET kérés a webszerver egy végpontján.

A konténer állapotát meg kell jeleníteni a kliens oldalon, ennek egy egyszerűbb módja a rendszeres időközönkénti pollozás, azonban egy felesleges hálózati forgalmat generálna. A megoldást, a *Server Sent Eventek*[1] használata jelent. A kliens a konténer web alkalmazás környezet létrehozásakor feliratkozik a szerveren a megfelelő csatornára és a szerver pedig értesítést küld a konténerek státusztának változásáról. A TMS a konténerek státuszát a Docker Host API egy erre a célra kialakított végpontjára[2] történő kérés segítségével végezheti, így a teljes folyamat esemény vezérelt, azaz sem a kliens, sem a szerver oldalon nincs szükség pollozásra. A folyamat szekvencia diagramját a 4.3 ábra alsó része mutatja be.

Konténer állapot felügyelése

A dinamikus felügyelés nem került megvalósításra. Az oktató számára a konténer napló állományok letölthetőek. Illetve a webalkalmazás végrehajtásának állapotát pollozza a kliens annak érdekében, hogy az automatikus leállítás megtörténtéről tudomást szerezzen.

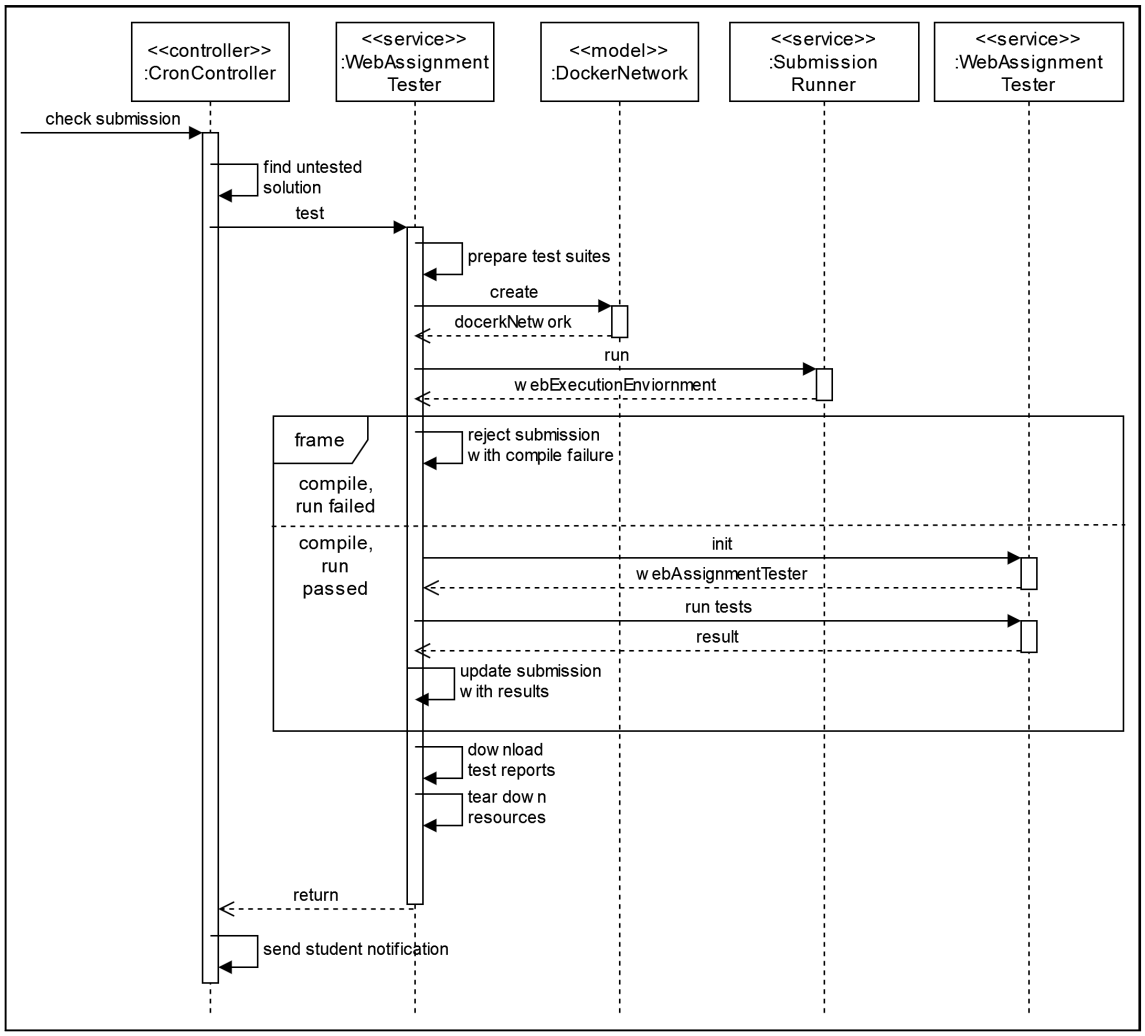
4.2. Automatizált tesztek végrehajtása

Az automatizált tesztelés kialakítása nem igényelt módosításokat az adatmodell szerkezetében. Csupán egy újabb instruktori fájl típust kellett bevezetni `Web test suite` néven. Az oktató által feltöltött teszteseteket a többi instruktori fájllal analóg módon kezeli a TMS. A fájl szintaxisát a TMS feltöltéskor ellenőrzi és hiba esetén az oktató egy részletes leírást tartalmazó fájlt kap eredményül a validációs hibáról.

Az automatizált teszt végrehajtásának szekvencia diagramját a 4.4 ábra mutatja be.

[1] HTTP alapú push technológia, melynek segítségével az szerver oldaról egyirányú üzenet küldésre van lehetőség a kliens irányába

[2] https://docs.docker.com/engine/api/v1.41/#operation/SystemEvents



4.4. ábra. Webalkalmazás automatizált tesztelésének szekvencia diagramja

A végrehajtást a `CronController` osztály vezérli, mely periodikusan hajtja végre a teszteket. első lépésként kiválaszt egy még nem tesztelt beadandót, majd azt átadja tesztelésre a `WebAssignmentTester` szervíznek. Az osztály összegyűjti a feladatkiíráshoz tartozó teszt állományokat, illetve létrehoz egy Docker hálózatot. Minden teszt a saját virtuális hálózatában fut, ezzel teljesen izolálva a párhuzamosan futó teszteket. A hálózat létrehozását követően a `SubmissionRunner` segítségével elindítja a SUT-t az előbbiekben konfigurált hálózatban (a `SubmissionRunner` felelősségeit részletesen bemutattam a 4.1.2 szakaszban). Amennyiben a SUT fordítása és futtatása sikertelen a tesztelés ezen a ponton véget ér, a beadandó értékelésre kerül a hibaüzenetek alapján. Amennyiben a SUT indítása sikeres a `SubmissionRunner` inicializál egy `WebAssignmentTester`-t, mely végrehajtja a teszteket.

A `WebAssignmentTester` tulajdonképpen egy `DockerContainer`-t csomagol speciálisan a Robot keretrendszer futtatására kialakítva. A tesztelő konténer ugyanazt

a *network stack-t* használja, mit a SUT így hálózati látencia nélkül, közvetlenül a loopback (localhost) IP címen eléri a SUT-t. A tesztek futtatása *fail fast* módon történik, azaz az első hibás teszteset után további teszteket nem hajt végre a rendszer így spórolva az erőforrásokkal. A tesztek végrehajtását követően a teszt riportokat a TMS kinyeri a konténerből és mind az oktató, mind a hallgató számára letölthetőek.

Automatizált tesztelés megvalósítása

Az automata tesztelés a fentiekben leírtaknak megfelelően került megvalósításra. A követelményelemzésben megfogalmazott rejtett és nem rejtett tesztesetek megkülönböztetését nem valósítottam meg.

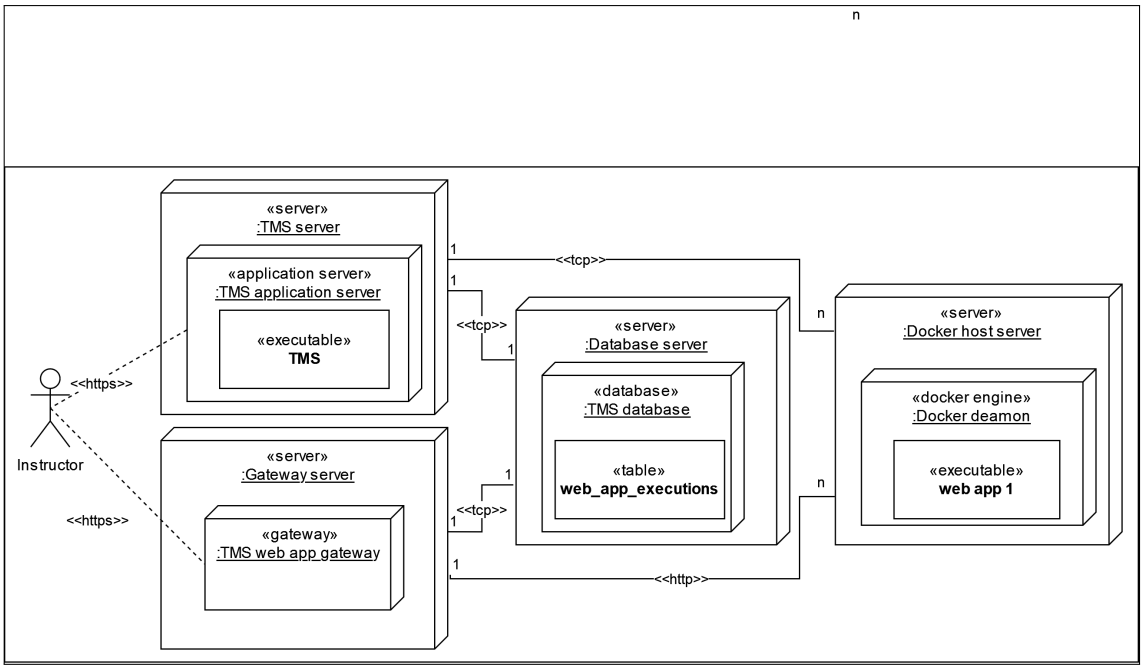
4.3. Biztonság és skálázhatóság

A követelményelemzés során meghatározott legfontosabb nem-funkcionális jellemző megvalósítása részben átfedő módon történt, ezért egy alfejezetben tárgyalom őket

4.3.1. Távoli hozzáférés korlátozása

A Docker Hostok olyan tűzfal mögötti szervereken is futhatnak, melyek közvetlenül nem elérhetőek az internetről. Ha a hallgató beadandóját egy ilyen szerverre telepíti a TMS az fokozott biztonsági kockázatot jelent. Ezeknek a szervereknek a külvilág számára rejtettnek kell maradnia és garantálni kell, hogy ahhoz csak a környezetet létrehozó oktató tud hozzáférni.

Egy reverse proxy segítségével a biztonsági rés lezárható. A 4.5 ábra egy teljesen elosztott telepítési diagramját mutatja be a TMS-nek.



4.5. ábra. TMS komponensek telepítési diagramja

A webalkalmazások elérése egy Apache reverse proxyn keresztül történik. Az oktató a sikeres alkalmazás indítást követően megkapja a webalkalmazás URL-t és egy a TMS által generált egyszeri jelszót, mely a webalkalmazás példányhoz tartozik.

Az URL a proxy szerver irányába mutat és *subdirectoryként* tartalmazza a webalkalmazást tartalmazó konténer nevét. A linkre navigálva egy egyszerű bejelentkezési lap jelenik meg, ahol az oktatónak be kell ütnie a helyes jelszót a tovább haladáshoz. A proxy szervernek olvasási hozzáférése van a TMS adatbázishoz, így az ott tárolt adatok alapján a konténer nevének ismeretében fel tudja oldani a futtató szerver IP címét és a konténerhez tartozó publikus port számát. Az ehhez tartozó jelszó egyezése esetén a proxy szerver létrehoz egy munkamenetet a felhasználó számára és a további kéréseket továbbítja a konténerben futó webalkalmazás felé.

Webalkalmazás elérésének korlátozása

A proxy mechanizmus nem került kialakításra, de a TMS adatbázisa és konfigurációja fel lett készítve a funkció bevezetésére. Amíg a funkció nem elérhető a webalkalmazások csak abban az esetben futtathatóak távolról, ha a Docker Hostot és TMS-t ugyanaz a szervergép szolgálja ki.

4.3.2. Biztonságos végrehajtási környezet

A Docker konténerek futtatása több biztonsági rést is magában hordoz.

Docker biztonsági lépések

Ebben az alfejezetben tárgyalt koncepciók nem kerültek egységesen megvalósításra a prototípusban.

Docker képek sérülékenysége

Potenciális biztonsági rés lehet, ha a Docker kép önmagában tartalmaz sérülékenységet, leginkább akkor fordulhat elő ilyen eset, ha nem hivatalos, gyártói képet használ a TMS. Az ilyen sérülékenységek feltárását segítik a statikus kódelemező eszközök, például: Clair[3], Trivy[4].

Erőforrás felhasználás korlátozása

A Linux kernel *control groupok* segítségével lehetőség van erőforrás felhasználási limiteket beállítani a konténereknek. A Docker támogatja ez erőforrás felhasználás kalibrálását[5] A TMS a konténer indításkor alapértelmezett értékek használatával ésszerű gátat szabhat az esetlegesen elszabaduló erőforrás használatnak. Azonban a pontos értékek meghatározása nagy körültekintést igényel.

Jogosultságok korlátozása

A jogosultságok nem engedélyezett eszkalálását elkerülendő a konténereknek indításakor a TMS-nek egy speciális, korlátozott jogokkal rendelkező felhasználót kell használnia minden esetben. Ezenfelül az alkalmazás típusának függvényében a nem szükséges Linux kernel képességeket[6] le kell tiltani és a további privilégiumok kiosztását is tiltani kell[7].

Hálózati hozzáférés

Az egyes környezetek külön, izolált hálózaton kell, hogy fussanak. Abban az esetben is, ha a környezet egy konténerből áll létre kell hozni egy dedikált hálózatot a környezetnek.

[3] `https://github.com/quay/clair`
[4] `https://github.com/aquasecurity/trivy`
[5] `https://docs.docker.com/config/containers/resource_constraints/`
[6] `https://docs.docker.com/engine/reference/run/#runtime-privilege-and-linux-capabilities`
[7] `https://docs.docker.com/engine/reference/run/#security-configuration`

Ha egy konténernek, környezetnek nincs szüksége hálózati kommunikációra, akkor az tiltani kell[8], pl: konzolos alkalmazások

A *Host* hálózati üzemmód használatát kerülni kell.

4.3.3. Skálázhatóság

A webalkalmazások futtatásának kritikus erőforrása a Docker Host, a TMS csak lényegében az alkalmazások indításában és leállításában vesz részt. A 4.5 ábrán egy olyan telepítési diagram látható, melyen a TMS minden komponense elosztottan fut. A gyakorlatban erre természetesen nincsen szükség, azonban a proxy mechanizmusnak köszönhetően akár több Docker Host egyidejű kezelésére is lehetőség van, így megvalósítva a horizontális skálázást.

Docker Hostok skálázása

Jelenleg a TMS két Docker Hostot tud kezelni: egy dedikált windows és egy dedikált linux hostot.

[8] `https://docs.docker.com/network/none/`

5. fejezet

Összegzés

A diplomamunkámban részletesen megvizsgáltam a webes alkalmazások távoli futtatásának és automatizált átvételi tesztelésének technológiai hátterét és lehetőségeit.

A bemutatott rendszerterv két az iparban elterjedt technológián alapszik. A Dockerrel pehelysúlyú izolált alkalmazás környezetek kialakíthatóak, komplett virtuális hálózatokkal. A Robot keretrendszer és plugin könyvtárai köszönhetően az átvételi tesztek magasszintű definiálására biztosít lehetőséget. Az technológiák jól integrálhatóak a TMS infrastruktúrájával, így az elkészített prototípus jól beágyazható.

A prototípus fejlesztése és a rendszerterv elkészítése során a legnagyobb kihívást a konténerizált környezetek és az automatizált tesztek robusztussága jelentette. A hálózati alkalmazások esetén, ugyanis gyakoriak a tranziens hibák, az alkalmazások indításakor ezek gyakran előfordulnak. A tervezés során tehát ügyelni kellett arra, hogy ezekkel a hibákkal szemben a TMS kellően ellenálló legyen, ugyanakkor a beépített extra várakozások ne menjenek a felhasználói élmény és gazdaságos erőforrás felhasználás túlzott rovására. Bár a prototípusban ezen megoldások közül kevés került megvalósításra, a rendszerterv során ezek megoldása nagy hangsúlyt kapott.

A kulcsszó vezérelt átvételi tesztek és teszt függvénykönyvtárak által nyújtott szolgáltatások a prototípus kipróbálása során a vártnál is jobb felhasználói élményt nyújtottak. A modul használata megköveteli az oktatóktól a vonatkozó technológiák legalább felszínes megismerését, ugyanakkor azt gondolom, hogy a tesztesetek tömörsége és nyelvfüggetlensége vonzó megoldás lesz számukra.

A futási környezetek definiálásakor a konfigurációs lehetőségeket a minimálisra

szorítottam, hogy a Dockert nem ismerő oktatók számára is könnyen használható legyen a rendszer. Ugyanakkor a prototípus kipróbálása során a megfelelő fordítási és futattási parancsok megírásához szükséges legalább az adott Docker képhez tartozó főbb konfigurációs paraméterek ismerete. Így a jövőben elképzelhető, hogy a TMS által biztosított konfigurációs opciókat is bővíteni lehet.

A következő két alfejezetben röviden összehasonlítom a TMS-ben elkészült megoldást más megoldásokkal, illetve a rendszer tovább fejlesztésének a lehetőségeit is számba veszem.

5.1. Összehasonlítás

A 2.1 alfejezetben röviden bemutattam a TMS-hez hasonló rendszerek webalkalmazás tesztelési képességeit. Ezek közül egyedül a Submitty az, amit a mai napig aktívan fejlesztenek, így csak ezzel szemben végzem el a funkcionális összehasonlítást.

A Submittyben nem csak webalkalmazásokat, hanem tetszőleges hálózati alkalmazásokat lehet tesztelni (pl. elosztott algoritmusok). Éppen ezért a TMS-sel szemben egy sokkal kifinomultabb hálózati konfigurációt biztosít. Lehetőség van például routerek létrehozására, melyek a konténerek közötti adatforgalmat vezérlik és naplózzák. Utóbbit az oktató az értékelés és tesztelés során használhatja fel.

A környezetek definiálása a Submittyben egy JSON állományban, deklaratív módon történik. Ennek előnye, hogy tömör strukturált formában lehet megadni egy komplexebb futtatási környezetet egy egyszerű, specifikus nyelvvel. A TMS ezzel szemben egy grafikus felületet biztosít, mely később, bonyolultabb környezetek esetén akadályozó tényező lehet. A felhasználási szokások függvényében egy hasonló lehetőség kialakítása a TMS-ben is indokolt lehet.

A Submitty erősségeként említhető a finomhangolható hálózati konfiguráció és biztonsági beállítások. Például megadható, hogy egy futtatási környezet milyen külső weboldalakat, hálózati erőforrásokat érhet el.

Ugyanakkor a TMS-sel ellentétben a Submittyben nincs lehetőség a webes alkalmazások távoli elérésére, ami meglepő tekintve, hogy a teszteléshez a távoli környezetek kialakítására amúgy is szükség van. Egyszerűbb webalkalmazás környezetek esetén a TMS által biztosított konfigurációs felület könnyebben használható.

A Submittyben a hálózati alkalmazások tesztelése nem igazán kiforrott megoldás. Tulajdonképpen a routerek által naplózott forgalom kezelhető a validáció alapjaként. A TMS-be integrált Robot keretrendszer ezzel szemben közvetlenül az hallgató alkalmazásával kommunikál. Ráadásul a plugin könyvtár struktúra segítségével akár a hálózat többi komponensével is interakcióba tud lépni (pl. adatbázis lekérdezések futtatása).

Összességében elmondható, hogy a Submitty konfigurálhatóság szempontjából bizonyos esetekben többet tud a TMS-nél, de utóbbi lényegesen felhasználóbarátabb megoldásokat nyújt.

5.2. Továbbfejlesztés lehetőségei

A diplomamunkámban ismertetett rendszerterv és a prototípus közötti eltéréseket a 4. fejezetben bemutattam, a prototípus kiegészítése a hiányzó komponensekkel javasolt. Ezek növelik a rendszer robusztusságat és javítják a felhasználói élményt. A következőkben néhány további, a funkcionalitásokat bővítő lehetőséget mutatok be:

- **Automatikus osztályozás:** Az automatikus tesztek eredménye felhasználható lenne, hogy a hallgató munkáját előzetesen osztályozza. Így a jelenlegi bináris értékelés helyett a sikeres tesztek arányának függvényében lenne pontozható a hallgató munkája. Az automatikus tesztek eredménye elérhető xUnit sztenderd formátumban, így ez a feldolgozás alapjául szolgálhat

- **Sebességkorlátozás:** A webalkalmazás környezetek kialakítás és a tesztek futtatása idő és erőforrás igényes. Ha egy hallgató gyakran tölt fel előzetesen nem letesztelt megoldást, azzal jelentős rendszerterhelést okozhat. Érdemes lehet korlátozni a hallgató által feltölthető maximális megoldások számát egy adott időintervallumban.

- **Hálózati alkalmazások tesztelése:** Bár az infrastruktúra kialakítása során a webes alkalmazások igényeit tartottam szem előtt, a hálózati komponensek kellő fokú általánosításával, akár hálózati algoritmusok, adatbázis beadandók tesztelése is elvégezhető lenne.

Irodalomjegyzék

[1] Matthew Peveler, Evan Maicus és Barbara Cutler. "Comparing jailed sandboxes vs containers within an autograding system". *Proceedings of the 50th ACM Technical Symposium on Computer Science Education*. 2019, 139–145. old.

[2] Mate' Sztipanovits, Kai Qian és Xiang Fu. "The automated web application testing (AWAT) system". *Proceedings of the 46th Annual Southeast Regional Conference on XX*. 2008, 88–93. old.

[3] Xiang Fu és tsai. "APOGEE: automated project grading and instant feedback system for web based computing". *ACM SIGCSE Bulletin* 40.1 (2008), 77–81. old.

[4] Antonio Carvalho Siochi és William Randall Hardy. "Webwolf: Towards a simple framework for automated assessment of webpage assignments in an introductory web programming class". *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*. 2015, 84–89. old.

[5] Susanta Nanda Tzi-cker Chiueh és Stony Brook. "A survey on virtualization technologies". *Rpe Report* 142 (2005).

[6] Junzo Watada és tsai. "Emerging trends, techniques and open issues of containerization: a review". *IEEE Access* 7 (2019), 152443–152472. old.

[7] Anil Madhavapeddy és David J Scott. "Unikernels: Rise of the Virtual Library Operating System: What if all the software layers in a virtual appliance were compiled within the same safe, high-level language framework?": *Queue* 11.11 (2013), 30–44. old.

[8] Jurenka. "Virtualization using Docker Platform". *Faculty of Informatics Masaryk University* (2015).

[9] Christian Ryding és Rickard Johansson. "Jails vs Docker: A performance comparison of different container technologies". 2020.

[10] Ákos Kovács. “Comparison of different Linux containers”. *2017 40th International Conference on Telecommunications and Signal Processing (TSP)*. IEEE. 2017, 47–51. old.

[11] Pierre Bourque és Richard E. Fairley. *SWEBOK: Guide to the Software Engineering Body of Knowledge*. IEEE Computer Society, 2014.

[12] Erik Van Veenendaal, Dorothy Graham és Rex Black. *Foundations of software testing: ISTQB certification*. Delmar Learning, 2012.

[13] Mike Cohn. *Succeeding with agile: software development using Scrum*. Pearson Education, 2010.

[14] Ham Vocke. “The practical test pyramid”. *martinfowler. com* (2018).

[15] Zhenyue Long és tsai. “WebRR: self-replay enhanced robust record/replay for web application testing”. *Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 2020, 1498–1508. old.

[16] Pekka Laukkanen és tsai. “Data-driven and keyword-driven test automation frameworks”. *Master’s thesis. Helsinki University of Technology* (2006).

[17] Pantakarn Sriwichainan és Taratip Suwannasart. “Generating Test Scripts for Web-Based Applications”. SSIP 2020. Association for Computing Machinery, 2020, 84–89. ISBN: 9781450388283. DOI: 10.1145/3441233.3441248. URL: https://doi.org/10.1145/3441233.3441248.

[18] Renaud Rwemalika és tsai. “On the evolution of keyword-driven test suites”. *2019 12th IEEE Conference on Software Testing, Validation and Verification (ICST)*. IEEE. 2019, 335–345. old.

[19] Sumit Bisht. *Robot framework test automation*. Packt Publishing Ltd, 2013.

[20] David Burns Simon Stewart. *WebDriver*. Techn. jel. World Wide Web Consortium, 2022. ápr.

[21] Elior Vila, Galia Novakova és Diana Todorova. “Automation testing framework for web applications with Selenium WebDriver: Opportunities and threats”. *Proceedings of the International Conference on Advances in Image Processing*. 2017, 144–150. old.

[22] T Harsha és BA Sujatha Kumari. “Software Test Automation with Robot Framework”. *International Journal on Recent and Innovation Trends in Computing and Communication* 5.6 (2017), 432–433. old.

[23] Gergő Prepok. “Online beadandó kezelő alkalmazás fejlesztése”. BSc szakdolgozat. ELTE Informatikai Kar, 2017, 51. old.

[24] Péter Kaszab. “Beadandókezelő webalkalmazás fejlesztése REST API alapokon”. BSc szakdolgozat. ELTE Informatikai Kar, 2021, 57. old.

[25] Éva Hartung. “Verziókezelt beadandó menedzsment webalkalmazás fejlesztése”. BSc szakdolgozat. ELTE Informatikai Kar, 2020, 48. old.

[26] Zoltán Zele. “Automatizált beadandó tesztelő webalkalmazás fejlesztése”. BSc szakdolgozat. ELTE Informatikai Kar, 2018, 39. old.

Ábrák jegyzéke

Táblázatok jegyzéke

Forráskódjegyzék