**SZAKDOLGOZAT**

Inrastructure as a Code (IaC): Automatizált infrastrukúra Terraform és Jenkins segítségével

Készítette: Baksa Ádám FLEQ8S

Tanszéki konzulens: Farkas Attila

Külső konzulens: Timári Zsolt - SAP Hungary Kft.

# Tartalomjegyzék

[Tartalomjegyzék 2](#_Toc166759660)

[1 Bevezetés 4](#_Toc166759661)

[2 IaC – Infrastructure as a Code 5](#_Toc166759662)

[2.1 Mi az „Infrastruktúra, mint kód”? 5](#_Toc166759663)

[2.2 Az IaC deklaratív és imperatív megközelítései 6](#_Toc166759664)

[2.3 IaC előnyei 6](#_Toc166759665)

[2.4 Milyen problémát old meg az IaC? 7](#_Toc166759666)

[2.5 Mennyire számít az IaC a DevOps-ban? 7](#_Toc166759667)

[2.6 Következtetés 8](#_Toc166759668)

[3 Terraform 9](#_Toc166759669)

[3.1 Mi a Terraform? 9](#_Toc166759670)

[3.2 Terraform szolgáltatók az Azure -infrastruktúrához 9](#_Toc166759671)

[3.3 A Terraform és az Azure előnyei 10](#_Toc166759672)

[3.3.1 Gyakori IaC-eszköz 10](#_Toc166759673)

[3.3.2 Infrastruktúra-kezelés automatizálása 10](#_Toc166759674)

[3.3.3 Az infrastruktúra változásainak megismerése alkalmazás előtt 10](#_Toc166759675)

[4 Jenkins 12](#_Toc166759676)

[4.1 A Jenkins fő jellemzői és előnyei: 12](#_Toc166759677)

[4.2 Mire használják a Jenkins-t a DevOps-ban? 12](#_Toc166759678)

[5 Azure DevOps 13](#_Toc166759679)

[6 A Terraform – Jenkins – Azure kapcsolata 14](#_Toc166759680)

[6.1 Terraform és Azure 14](#_Toc166759681)

[6.2 Jenkins és Terraform 14](#_Toc166759682)

[6.3 Az összekapcsolás működése 14](#_Toc166759683)

[7 Feladat 16](#_Toc166759684)

[8 Környezet előkészítése Azure-ben 18](#_Toc166759685)

[9 Terraform konfiguráció és kódleírás 19](#_Toc166759686)

[9.1 providers.tf 23](#_Toc166759687)

[9.2 terraform.tfvars 24](#_Toc166759688)

[9.3 variables.tf 25](#_Toc166759689)

[9.4 main.tf 26](#_Toc166759690)

[10 Jenkins 46](#_Toc166759691)

[10.1 Jenkins telepítése 46](#_Toc166759692)

[10.2 Jenkins konfiguráció 47](#_Toc166759693)

[10.3 Jenkins pipeline kód 48](#_Toc166759694)

[10.4 Jenkins pipeline létrehozása 51](#_Toc166759695)

[10.5 Jenkins pipeline futtatása 52](#_Toc166759696)

[11 A felépített infrastrukúra működése 53](#_Toc166759697)

[11.1 Azure portal 53](#_Toc166759698)

[11.2 Webapp 54](#_Toc166759699)

[12 Következtetések, javaslatok 55](#_Toc166759700)

[12.1 Következtetés 55](#_Toc166759701)

[12.2 Javaslatok 56](#_Toc166759702)

[13 Összefoglalás 57](#_Toc166759703)

# Bevezetés

Az informatikai iparág folyamatosan fejlődik és a technológiai változásokkal együtt a vállalatoknak is alkalmazkodniuk kell. Az infrastruktúra kezelése és karbantartása korábban időigényes, és erőforrásokat igénylő feladat volt, amely jelentős terhet rótt a vállalatokra. Azonban az elmúlt években az Infrastructure as a Code (IAC) koncepciója megjelent, amely lehetővé teszi az infrastruktúra automatizált kezelését és konfigurálását. A Terraform és a Jenkins olyan eszközök, amelyek segítségével a vállalatok hatékonyabban tudják kezelni infrastruktúrájukat, és lehetővé teszik a folyamatos integrációt és folyamatos szállítást (CI/CD).

A szakdolgozatom témájának választását az SAP-nál végzett munkám inspirálta. Az SAP egy vezető szoftverfejlesztő vállalat, amely számos vállalat számára biztosít üzleti alkalmazásokat és megoldásokat. Az SAP-nál a fő célunk az, hogy az ügyfeleknek minél hatékonyabb és megbízhatóbb szolgáltatásokat nyújtsunk. Az infrastruktúra kezelése és karbantartása azonban jelentős erőforrásokat és időt igényel, amelyeket inkább a fejlesztésre és az ügyféligények kielégítésére szeretnénk fordítani. Az IaC alkalmazása ebben a kontextusban számos előnnyel járhat, beleértve a gyorsabb szolgáltatásnyújtást, a hibák csökkentését, a költségek minimalizálását és a rendszerek átláthatóságának növelését.

A projekt célja egy Windows szerver automatizált létrehozása Microsoft Azure környezetben, felhasználva a Terraform infrastruktúra menedzsment eszközt. A Terraform kód határozza meg a szerver konfigurációt, ez tárolva lesz egy GIT repository-ban. A Jenkins, automatizálja a folyamatot: amint új változtatás érkezik a GIT repo-ba, Jenkins lefuttatja a Terraform kódot, ami létrehozza vagy módosítja a Windows szervert Azure-ben. Ez a megközelítés növeli a hatékonyságot, egységesíti a környezeteket és csökkenti a hibalehetőségeket.

A Terraform egy nyílt forráskódú infrastruktúra automatizáló eszköz, amely lehetővé teszi a felhasználók számára, hogy leírják és biztosítsák infrastruktúra erőforrásaikat a kód használatával. Ez az eszköz platformfüggetlen, ami azt jelenti, hogy támogatja a legtöbb nagy felhőszolgáltatót, beleértve az Amazon Web Services-t, a Google Cloud Platform-ot és az Azure-t is.

A Jenkins ismerete rendkívül fontos egy DevOps mérnök számára, mivel segít automatizálni és felgyorsítani a szoftverfejlesztési folyamatokat. A Jenkins használatával a fejlesztők könnyedén létrehozhatnak és futtathatnak automatikus teszteket, illetve biztosítani tudják a folyamatos integrációt és a folyamatos szállítást a kódváltoztatások során.

# IaC – Infrastructure as a Code

Az infrastruktúra fontos része a szoftverfejlesztési folyamatnak, mivel közvetlenül felelős a szoftveralkalmazások zökkenőmentes működéséért. Szerverek, terheléselosztók, tűzfalak, adatbázisok és bonyolult konténerfürtök mind példák az infrastruktúrára. Mivel az infrastrukturális nehézségek az egész fejlesztési folyamatot áthatják, ezek a termelési helyzeteken túl is relevánsak. Ide tartoznak többek között a CI/CD platformok, állomásozó környezetek és tesztelőeszközök.

A szoftvertermék összetettségének növekedésével ezek az infrastrukturális kihívások egyre kritikusabbakká válnak. Az infrastruktúra manuális kezelésének hagyományos technikája gyorsan skálázhatatlan megoldássá válik, hogy megfeleljen a mai DevOps-alapú gyors szoftverfejlesztési ciklusok törekvéseinek.

Ennek eredményeként az Infrastructure as Code (IaC) mára a ’de facto’ fejlesztési megoldássá vált. Az infrastruktúra kódként (IaC) lehetővé teszi az infrastruktúra változásainak méretezését és nyomon követését, amikor azok felmerülnek.

Vizsgáljuk meg az infrastruktúrát kódként, beleértve annak előnyeit, létfontosságú jelentőségét és még sok mást.

## Mi az „Infrastruktúra, mint kód”?

Az Infrastructure as Code az a folyamat, amikor a megfelelő eszközök és rendszerek manuális konfigurálása helyett kód használatával biztosítanak és konfigurálnak egy környezetet. A fejlesztők a kódparaméterek meghatározása után szkripteket futtatnak, és az IaC platform automatikusan generálja a felhő infrastruktúrát.

Az ilyen automatizált informatikai konfigurációk lehetővé teszik a csapatok számára, hogy gyorsan elkészítsék a termékük teszteléséhez és futtatásához szükséges felhőbeállításokat. Az infrastruktúra, mint kód lehetővé teszi a fejlesztők számára, hogy tetszőleges infrastruktúra-összetevőt hozzanak létre, például hálózatokat, terheléselosztókat, adatbázisokat és virtuális gépeket.

Laikus kifejezéssel élve, ez az infrastruktúra biztosításának és kezelésének folyamata, amelyet kóddal, nem pedig kézzel határoznak meg. Az IaC egy fontos DevOps-technika is.

Lehetővé teszi a DevOps csapatok számára az infrastruktúra gyors felépítését és verziókezelését a forráskód verziószámával megegyező módon, valamint nyomon követhetik ezeket a verziókat, hogy minimalizálják az IT-környezetek közötti inkonzisztenciát, ami komoly problémákat okozhat a telepítés során.

## Az IaC deklaratív és imperatív megközelítései

Az IaC kétféleképpen közelíthető meg: deklaratív vagy imperatív.

Egy IaC-eszköz beállítja a rendszert, ha deklaratív megközelítést használ, amely leírja a rendszer tervezett állapotát, beleértve azt is, hogy milyen erőforrásokra van szüksége és milyen tulajdonságokkal kell rendelkezniük.

A deklaratív megközelítés nyomon követi a rendszerobjektumok aktuális állapotát is, így könnyebben kezelhető az infrastruktúra állásideje. Egy kötelező módszer viszont felvázolja azokat az utasításokat, amelyeket a megfelelő sorrendben kell végrehajtani a tervezett konfiguráció létrehozásához.

Számos IaC technológia deklaratív megközelítést alkalmaz az infrastruktúra kiépítéséhez, és ezt automatikusan megteszi. Egy deklaratív IaC-eszköz módosítja a kívánt állapotot. Ha alapvető eszközt használ, meg kell találnia, hogyan alkalmazhatja ezeket a korrekciókat. Az IaC eszközök gyakran mindkét módban képesek működni, bár előnyben részesítik egyiket a másikkal szemben.

## IaC előnyei

Az IaC automatizált eljárásokon keresztül számos módon segíti a vállalatokat IT-infrastruktúra-igényeik kezelésében. Az IaC telepítésének néhány előnye a következő:

1. *Konzisztencia*: Az IaC növelheti a konzisztenciát és csökkentheti a kézi beállítások során gyakran előforduló hibákat. Ezenkívül megakadályozza a konfigurációs eltolódást, amely a kézi művelet során előfordulhat. Az IaC lehetővé teszi a nem dokumentált, ad-hoc konfigurációmódosítások megelőzését a konfigurációs szabványok kódolásával és dokumentálásával.
2. *Hatékonyság*: Az infrastruktúra kódolásával létrehozható egy kiépítési sablon, ami megkönnyíti a rendszerkonfigurációt, a karbantartást és az adminisztrációt. Rugalmas, megismételhető és méretezhető infrastruktúrát épít ki. Ennek eredményeként a DevOps felgyorsíthatja a szoftverfejlesztés minden szakaszát, így naponta több alkalmazás jelenik meg.
3. *Költség csökkentése*: Az IaC lehetővé teszi a virtuális gépek programozott kezelését, így nincs szükség manuális hardverkonfigurációra és frissítésre. Ugyanazzal a kóddal egy kezelő egy gépet vagy 1000 egységet telepíthet és kezelhet. Ennek eredményeként kevesebb alkalmazottra van szükség, és többé nincs szükség új felszerelésre, ami jelentős költségmegtakarítást eredményez.
4. *Sebesség*: Az IaC lecsökkenti a fejlesztőknek az infrastruktúra biztosításához szükséges időt azáltal, hogy azt egyszerű scripté alakítja. Ennek eredményeként az alkalmazások telepítését már nem késlelteti az infrastruktúra és az új szoftverek lényegesen gyorsabban szállíthatók.
5. *Kockázat csökkentése*: Az IaC lehetővé teszi az infrastruktúra könnyű integrálását a verziókezelési mechanizmusokba, nyomon követhető és auditálható. Ezáltal csökkent a kockázat.

## Milyen problémát old meg az IaC?

Az Infrastructure as Code a kiadási folyamat környezeti eltolódásának problémájának megoldására jött létre. IaC nélkül a csapatok felelősek az egyes telepítési környezetek beállításainak karbantartásáért. Minden környezet egyedivé válik, egy olyan különleges elrendezéssé, amely nem reprodukálható automatikusan.

A telepítések során az eltérések a környezetek között problémákat okoznak. Az egyedi "hópehely" környezetek manuális műveleteket igényelnek, amelyek nehezen kezelhetőek és hibákhoz vezetnek az infrastruktúra kezelésében és karbantartásában.

Az Infrastructure as Code az idempotencia elvét követi. Az idempotencia azt jelenti, hogy egy telepítési parancs mindig azonos módon konfigurálja a célkörnyezetet, függetlenül a környezet kezdeti állapotától. Az idempotenciát vagy egy meglévő cél automatikus beállításával érik el, vagy a meglévő cél figyelmen kívül hagyásával és újrakezdéssel.

Ennek eredményeként, az IaC használatával a csapatok módosítják a környezet leírását és a konfigurációs modell verzióját, amely gyakran jól dokumentált kódformátumokban, mint például JSON, van megírva. A modellt futtatják a kiadási csővezetékben a célkörnyezetek beállításához. Ha változtatásokat kell eszközölniük, a csapat a forrást módosítja, nem a célt.

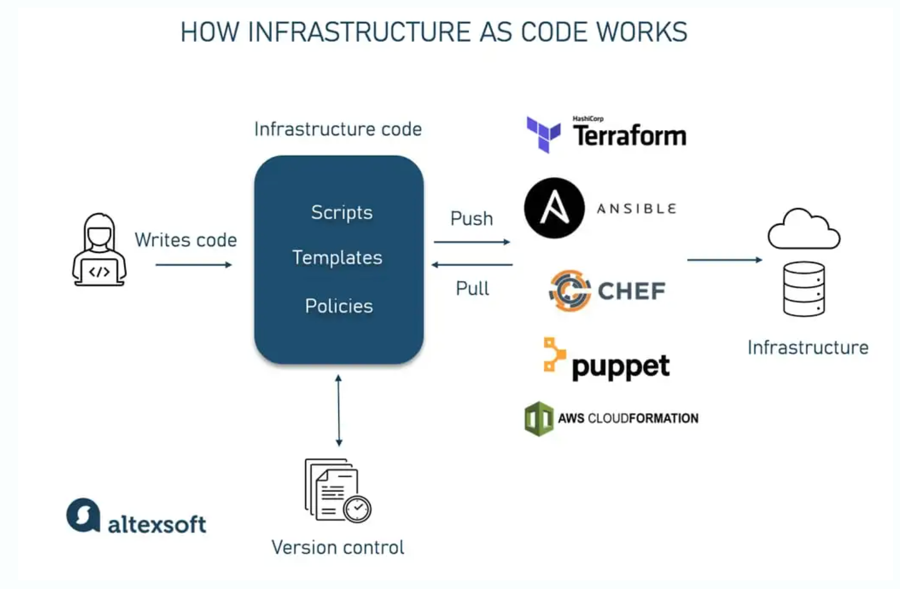
## Mennyire számít az IaC a DevOps-ban?

A DevOps és a folyamatos integráció/folyamatos szállítás (CI/CD) módszertanok alkalmazása az IaC használatát teszi szükségessé. Ezáltal a fejlesztők többségében mentesülnek a provisioning (infrastruktúra előkészítése) feladatai alól, lehetővé téve számukra, hogy egyszerűen egy script futtatásával indítsák el infrastruktúrájukat.

Ennek eredményeként az alkalmazások telepítése nem áll le az infrastruktúra kiépítése miatt, és a rendszergazdák sem terhelődnek az időigényes manuális feladatokkal. Az integrációtól és a teszteléstől kezdve a szállításon és telepítésen át a CI/CD a folyamatos automatizáláson és a folyamatos monitorozáson alapul az alkalmazás életciklusa során. Az automatizálás működéséhez konstans környezetre van szükség.

Amikor a fejlesztői csapat az alkalmazásokat vagy a környezeteket egy módon szállítja vagy konfigurálja, és a műveleti csapat egy másik módon telepíti és konfigurálja a környezetet, az alkalmazások automatizált telepítése nem lehetséges.

A DevOps módszertan a fejlesztési és műveleti csapatok összehangolásával kevesebb hibát, manuális telepítést és inkonzisztenciát eredményez. Mivel mind a fejlesztési, mind a műveleti csapatok ugyanazt a telepítési alkalmazás leírást használhatják, az IaC segít szinkronizálni a fejlesztést és a műveleteket, lehetővé téve a DevOps megközelítést.

Minden környezetnek, beleértve a termelési környezetet is, ugyanazon telepítési módszert kell követnie. Minden alkalommal, amikor az IaC-t használják, azonos környezet jön létre.

## Következtetés

A DevOps nagymértékben támaszkodik az infrastruktúrára, mint kódra. Lehetővé teszi, hogy teljes mértékben kiaknázhassa a cloud-computing adta lehetőségeket, csökkenti azokat a hibákat, amelyek a manuális IT infrastruktúra-kezeléssel járnak, és javítja a szoftverfejlesztés sebességét. Mindezt úgy éri el, hogy közben csökkenti a működési költségeket. (Jay, 2022).

# Terraform Consulting Services and Solution Provider: TX, USATerraform

## Mi a Terraform?

A Terraform egy nyílt forráskódú, infrastruktúra mint kód (IaC) eszköz, amit a HashiCorp fejlesztett ki. Lehetővé teszi az infrastruktúra-erőforrások deklaratív meghatározását és előkészítését egy magas szintű konfigurációs nyelv használatával, így a kívánt infrastruktúra állapotát leírva a Terraform gondoskodik az erőforrások létrehozásáról és kezeléséről. Támogatja a különböző felhőszolgáltatók és helyszíni infrastruktúra erőforrásainak kezelését, lehetővé téve az infrastruktúra egységes kezelését, verziókövetését, együttműködést és ellenőrzött változtatások alkalmazását.

A Terraform lehetővé teszi különböző infrastruktúra-erőforrások, mint virtuális gépek, hálózatok, tárolók, terheléselosztók definícióját és kezelését többféle felhőszolgáltatónál (például AWS, Azure, Google Cloud stb.) és helyszíni infrastruktúránál is. A Terraform szolgáltatói pluginokat használ a különböző infrastruktúra-platformokkal való interakcióra, lehetővé téve az erőforrások következetes kezelését az alapul szolgáló infrastruktúra-szolgáltatótól függetlenül.

A Terraform egyszerű és következetes munkafolyamatot biztosít az infrastruktúra kezeléséhez. Lehetővé teszi az infrastruktúra kódjának verziókövetését, másokkal való együttműködést és az infrastruktúrában végrehajtott változások ellenőrzött, auditálható módon történő alkalmazását. Támogatja az infrastruktúra mint kód koncepcióját, ami azt jelenti, hogy az infrastruktúrát szoftverként kezelheted, alkalmazva a szoftvermérnöki gyakorlatokat, mint a kódellenőrzést, tesztelést és folyamatos integrációt, élvezheted az automatizálás és ismételhetőség előnyeit.

Összességében a Terraform segít automatizálni az infrastruktúra-erőforrások előkészítését és kezelését, megkönnyítve az infrastruktúra idővel történő skálázását, karbantartását és fejlesztését.

## Terraform szolgáltatók az Azure -infrastruktúrához

Számos Terraform-szolgáltató teszi lehetővé az Azure-infrastruktúra kezelését:

* *AzureRM*: Stabil Azure-erőforrások és funkciók, például virtuális gépek, tárfiókok és hálózati felületek kezelése.
* *AzureAD*: Microsoft Entra-erőforrások, például csoportok, felhasználók, szolgáltatásnevek és alkalmazások kezelése.
* *AzureDevops*: Azure DevOps-erőforrások, például ügynökök, adattárak, projektek, folyamatok és lekérdezések kezelése.
* *AzAPI*: Azure-erőforrások és -funkciók kezelése közvetlenül az Azure Resource Manager API-kkal. Ez a szolgáltató az AzureRM-szolgáltatót azzal egészítja ki, hogy engedélyezi a nem kiadott Azure-erőforrások kezelését. Az AzAPI-szolgáltatóval kapcsolatos további információkért lásd: Terraform AzAPI-szolgáltató.
* *AzureStack*: Azure Stack Hub-erőforrások, például virtuális gépek, DNS, virtuális hálózatok és tárolók kezelése.

## A Terraform és az Azure előnyei

### Gyakori IaC-eszköz

A Terraform Azure-szolgáltatók lehetővé teszik az összes Azure-infrastruktúra kezelését ugyanazzal a deklaratív szintaxissal és eszközzel. Az alábbi szolgáltatókkal a következőkre van lehetőség:

1. Konfigurálja az alapvető platformképességeket, például felügyeleti csoportokat, szabályzatokat, felhasználókat, csoportokat és szabályzatokat. További információkért lásd a Nagyvállalati szintű felhőadaptálási keretrendszer Terraform-implementációját.
2. Konfigurálja az Azure DevOps-projekteket és folyamatokat a rendszeres infrastruktúra- és alkalmazástelepítések automatizálásához.
3. Az alkalmazások által igényelt Azure-erőforrások üzembe helyezése.

### Infrastruktúra-kezelés automatizálása

A Terraform sablonalapú konfigurációs fájlszintaxisával megismételhető és kiszámítható módon konfigurálhatja az Azure-erőforrásokat. Az infrastruktúra automatizálása a következő előnyöket foglalja magában:

* Csökkenti az emberi hibák esélyét az infrastruktúra üzembe helyezése és kezelése során.
* Ugyanazt a sablont helyezi többször üzembe, ezzel azonos fejlesztési, tesztelési és éles környezetet hoz létre.
* Csökkenti a fejlesztési és tesztelési környezetek költségeit, mivel igény szerint hozza létre őket.

### Az infrastruktúra változásainak megismerése alkalmazás előtt

Ahogy az erőforrás-topológia összetetté válik, az infrastruktúra-módosítások jelentésének és hatásának megértése bonyolult lehet.

A Terraform CLI lehetővé teszi a felhasználók számára az infrastruktúra változásainak ellenőrzését és előnézetét a terv alkalmazása előtt. Az infrastruktúra változásainak biztonságos előzetes megtekintése számos előnnyel jár:

* A csapattagok hatékonyabban működhetnek együtt a javasolt módosítások és azok hatásának megismerésével.
* A nem tervezett módosítások a fejlesztési folyamat korai szakaszában ragadhatók meg (Microsoft, 2024/a).

# JenkinsJenkins

A Jenkins egy nyílt forráskódú automatizálási szerver, amelyet széles körben használnak a DevOps gyakorlatokban szoftverfejlesztési folyamatok automatizálására, mint például a folyamatos integráció (Continuous Integration, CI) és a folyamatos kiszállítás (Continuous Delivery, CD). A Jenkins lehetővé teszi a fejlesztők számára, hogy gyorsan és hatékonyan építsenek, teszteljenek és telepítsenek alkalmazásokat, így javítva a szoftverfejlesztési ciklus sebességét és hatékonyságát.

## A Jenkins fő jellemzői és előnyei:

1. *Folyamatos integráció és szállítás*: A Jenkins automatizálja a kódfordítást, a tesztelést és az alkalmazások telepítését, lehetővé téve a fejlesztők számára, hogy gyakrabban és megbízhatóbban szállítsanak kódot.
2. *Plugin architektúra*: A Jenkins széles körű plugin rendszerrel rendelkezik, amely lehetővé teszi a felhasználók számára, hogy bővítsék a rendszer képességeit és integrálják a Jenkins-t más eszközökkel és szolgáltatásokkal.
3. *Automatizálási szabályok*: A Jenkins segítségével a fejlesztők automatizálási szabályokat írhatnak, például triggerek beállításával, amelyek alapján a Jenkins automatikusan elindítja a build és teszt folyamatokat.
4. *Skálázhatóság*: A Jenkins képes kezelni egyszerű egyfelhasználós környezetektől kezdve nagy, több projektet és sok szerverből álló környezetekig.
5. *Nyílt forráskódú és közösség*: Mivel Jenkins egy nyílt forráskódú projekt, nagy és aktív közösséggel rendelkezik, amely folyamatosan fejleszti és támogatja a platformot.

## Mire használják a Jenkins-t a DevOps-ban?

1. *Automatizált build-ek*: Automatizálja a szoftver build folyamatát, biztosítva, hogy minden kódmódosítás után újra legyen fordítva a projekt.
2. *Teszt automatizálás*: Integrálja a tesztelési keretrendszereket, hogy automatikusan futtassa a teszteket a build folyamat részeként, javítva a szoftverminőséget.
3. *Deploy automatizálás*: Automatizálja az alkalmazások különböző környezetekbe való telepítését, lehetővé téve a gyors és megbízható kiszállítást.
4. *Folyamatos monitorozás*: Integrálható monitorozó eszközökkel a rendszer állapotának és a teljesítményének nyomon követésére, értesítve a csapatokat a problémákról.

Jenkins kulcsfontosságú szerepet játszik a DevOps kultúrában, elősegítve a gyorsabb fejlesztési ciklusokat, a magasabb szoftverminőséget és a folyamatos innovációt (Sheth, 2020).

# Azure DevOps

Az Azure DevOps egy kollaboratív kultúrát és folyamatokat támogató eszköz, amely fejlesztőket, projektmenedzsereket és hozzájárulókat hoz össze szoftverfejlesztés céljából. Az Azure DevOps lehetővé teszi a cégek számára, hogy gyorsabban hozzanak létre és fejlesszenek termékeket, mint a hagyományos szoftverfejlesztési lehetőségekkel. Az Azure DevOps felhőalapú szolgáltatásokat kínál az Azure DevOps Services-en keresztül, vagy lokális megoldásként az Azure DevOps Server formájában. Integrált funkciókat nyújt, amelyek elérhetőek webböngészőn vagy IDE kliensen keresztül, és lehetőséget ad arra, hogy minden szolgáltatást használjunk, vagy csak azokat válasszuk ki, amelyek kiegészítik a meglévő munkafolyamatainkat. Az Azure DevOps több alapvető szolgáltatást tartalmaz:

1. *Azure boards*: Agile eszközöket biztosít a munka tervezéséhez, a kódhibák és a problémák nyomon követéséhez Kanban és Scrum módszerekkel.
2. *Azure repos*: Git tárolókat vagy Team Foundation Version Control (TFVC)-t biztosít a forráskód-kezeléséhez.
3. *Azure pipelines*: Build és release szolgáltatásokat nyújt a folyamatos integráció és szállítás támogatásához.
4. *Azure test plans*: Eszközöket nyújt az alkalmazások teszteléséhez, beleértve a manuális/kutató tesztelést és a folyamatos tesztelést.
5. *Azure artifacts*: Lehetővé teszi a csapatok számára, hogy olyan csomagokat osszanak meg, mint Maven, npm, NuGet stb. nyilvános és privát forrásokból, és integrálják a csomagmegosztást a pipeline-okba.

Az Azure DevOps támogatja a kiterjesztéseket és integrálható más népszerű szolgáltatásokkal, például a Campfire-rel, Slackkel, Trello-val, UserVoice-szal és több másikkal, valamint lehetőséget biztosít saját egyéni kiterjesztések fejlesztésére is.

Az Azure DevOps Services integrálódik a GitHub.com-mal és a GitHub Enterprise Server repozitóriumokkal, gyors beállítást, karbantartásmentes műveleteket, egyszerű együttműködést különböző domaineken át, rugalmas skálázhatóságot és sziklaszilárd biztonságot kínál. A felhőalapú build és telepítési szerverek, valamint az alkalmazások betekintései is elérhetők ezen a platformon (Microsoft, 2024/b).

# A Terraform – Jenkins – Azure kapcsolata

A Terraform, Jenkins és Azure kombinációja egy erőteljes összeállítás, amely lehetővé teszi a fejlesztők és rendszergazdák számára, hogy automatizálják és optimalizálják az infrastruktúra kezelési folyamatokat a cloud környezetben. Itt van, hogyan kapcsolódnak össze és működnek együtt ezen eszközök egy projekt keretében:

## Terraform és Azure

1. *Terraform*: A Terraform támogatja az Azure erőforrások kezelését, így a felhasználók kód segítségével hozhatnak létre, kezelhetnek és frissíthetnek Azure erőforrásokat, mint például virtuális gépeket, hálózati interfészeket, tárolókat és sok mást.
2. *Azure:* A Terraformmal való integráció lehetővé teszi a fejlesztők számára, hogy hatékonyan kezeljék és automatizálják az Azure infrastruktúrájukat.

## Jenkins és Terraform

Jenkins: A Jenkins integrálható a Terraformmal, lehetővé téve a Terraform scriptek automatizált futtatását a CI/CD pipeline részeként. Ez segít automatizálni az infrastruktúra telepítését és frissítését az Azure-ban.

## Az összekapcsolás működése

1. *Infrastruktúra definíció*: A fejlesztők Terraform kódot írnak az Azure infrastruktúra leírására, beleértve a kívánt erőforrásokat és konfigurációkat.
2. *Verziókezelés*: A Terraform kódokat Gitben tárolják, ami verziókezelést és változások nyomon követését teszi lehetővé.
3. *Automatizálás Jenkins-szel*: Amikor a Terraform kód frissül a Gitben, a Jenkins észleli a változást és automatikusan lefuttatja a Terraform scripteket, ami az Azure infrastruktúra telepítését vagy frissítését eredményezi. Ez biztosítja, hogy az infrastruktúra mindig a kódban definiált állapotnak megfelelően legyen konfigurálva.
4. *Visszajelzés és iteráció*: A Jenkins visszajelzést nyújt a telepítési folyamatról, lehetővé téve a fejlesztők számára, hogy szükség esetén finomítsák a Terraform kódokat.

Ez az összekapcsolás lehetővé teszi a fejlesztők számára, hogy gyorsan és hatékonyan hajtsanak végre infrastruktúra-változásokat az Azure környezetben, miközben minimalizálják a hibák kockázatát és javítják az üzemeltetési hatékonyságot. Az automatizáció és a CI/CD beépítése az infrastruktúra kezelési folyamatokba elősegíti a DevOps gyakorlatok alkalmazását és támogatja a gyorsabb szoftverfejlesztést.

# Feladat

A feladat célja a bevezetésben is említett Windows alapú virtuális gép automatizált létrehozása Azure környezetben Terraform használatával, és e folyamat automatizálása Jenkins segítségével. Az alábbiakban vázolok egy lépésről lépésre haladó útmutatót, hogy miként fogom megvalósítani a projektet:

1. **Környezet előkészítése**
   1. Terraform és a Jenkins telepítése.
2. **Azure előkészítése**
   1. Létrehozok egy Azure Active Directory (AD) alkalmazást és egy szolgáltatói fiókot (Service Principal), amit a Terraform használhat az Azure erőforrások kezelésére
   2. Resource Group készítése Azure-ben.
3. **Terraform konfiguráció** 
   1. Létrehozok egy Terraform konfigurációs fájlt (main.tf), ami leírja a Windows alapú virtuális gép infrastruktúráját Azure-ban. Ez magában foglalhatja az erőforráscsoportot, a virtuális hálózatot, a biztonsági csoportokat, a virtuális gép példányt és bármely más szükséges komponenst. (Hashicorp, 2024/a)
   2. Bekonfigurálom a Terraform **provider** szekcióját, hogy az Azure AD alkalmazás azonosítóival és titkos kulcsaival autentikáljon
   3. Létrehozok egy **variables.tf** fájlt az input változók definiálásához és egy **outputs.tf** fájlt az output változók kezeléséhez.
4. **Jenkins pipeline konfiguráció**
   1. A Jenkinsben létrehozok egy új pipeline projektet
   2. Bekonfigurálom a projektet, hogy használjon GIT SCM-et, hogy a Terraform kód repository-ját képes legyen a klónozni.
   3. A pipline script-ben definiálom a lépéseket, amelyek:
      1. Futtassák a *’terraform init’* parancsot a Terraform konfiguráció inicializálásához.
      2. Futtassák a *’terraform plan’* parancsot az infrastruktúra-terv ellenőrzéséhez.
      3. Futtassák a *’terraform apply’* parancsot az infrastruktúra létrehozásához
5. **Biztonsági megfontolások**
   1. Terraform állapotfájlok biztonságosan legyenek tárolva, például az Azure Storage-ban, és megfelelő hozzáférési irányelvek vannak érvényben.
   2. Az Azure AD alkalmazás titkos kulcsának bizalmas kezelése. Nem tárolom kódbázisban. Jenkins környezeti változók használata.
6. **Tesztelés**
   1. Tesztek futtatása a Jenkins pipeline-on keresztül, hogy ellenőrizzem, a Terraform kód hibátlanul létrehozza-e a kívánt infrastruktúrát az Azure-ban.

# Környezet előkészítése Azure-ben

A Microsoft Azure egy rendkívül sokoldalú felhőalapú platform, amely széles körű szolgáltatásokat és eszközöket kínál a fejlesztők, IT szakemberek és vállalatok számára. Ebben a szakaszban részletesen ismertetem az Azure portál regisztrációjának lépéseit, és bemutatom, hogy ezek a lehetőségek hogyan támogatják az infrastruktúra kialakítását.

**Az Azure portál regisztrációjának lépései:**

* + - 1. [Azure hivatalos oldalán](https://azure.microsoft.com/) kiválasztom a *Try Azure for free* lehetőséget
      2. Microsoft fiók használata a bejelentkezéshez
      3. Személyes adatok megadása
      4. Identitás ellenőrzése: Az Azure biztonsági okokból kérhet a telefonszámot vagy e-mail címet az ellenőrzéshez
      5. A képen szöveg, képernyőkép, szoftver, Számítógépes ikon látható

         Automatikusan generált leírásIngyenes előfizetés kiválasztása: A regisztrációs folyamat során választhatunk az ingyenes próbaidőszak vagy egyéb előfizetési csomagok közül.

1. Azure subscription kezdőfelület

**Az Azure, mint környezet választásának indoklása**

1. Sokoldalúság és skálázhatóság: Az Azure széleskörű szolgáltatásai lehetővé teszik minden típusú alkalmazás támogatását, az egyszerű webalkalmazásoktól kezdve a nagy adatfolyamokig és gépi tanulási modellekig.
2. Globális elérhetőség: Az Azure globális adatközpont-hálózata lehetővé teszi az alkalmazások gyors és megbízható futtatását bárhol a világon
3. Biztonság: Az Azure szigorú biztonsági protokollokat és megfelelőségi szabályozásokat követ, biztosítva az adatok és alkalmazások védelmét.
4. Integráció: Az Azure könnyen integrálható más Microsoft-szolgáltatásokkal és szoftverekkel, valamint támogatja a nyílt forráskódú technológiákat, biztosítva a rugalmasságot és a választási lehetőségeket
5. Költséghatékonyság: Az ingyenes próbaidőszak és a fizetési lehetőségek széles választéka lehetővé teszi, hogy csak azért fizess, amit ténylegesen használsz, segítve ezzel a költségek optimalizálását.

Az Azure platform ezekkel a jellemzőkkel rendkívül vonzó választás azok számomra, mert egy hatékony, biztonságos és rugalmas infrastruktúrát szeretnék kialakítani. Az Azure portál egyszerű regisztrációs folyamata és a felhasználóbarát kezelőfelülete tovább egyszerűsíti az infrastruktúra kialakítását és kezelését.

# Terraform konfiguráció és kódleírás

A Terraform és az Azure összekonfigurálása több lépésből áll, amely lehetővé teszi, hogy a Terraform segítségével automatikusan létrehozzak és kezeljek Azure erőforrásokat.

Első lépésként a legújabb Terraform (v1.7.4) verziót telepítettem a Windows operációs rendszerű gépemre, amely telepítő csomagját a hivatalos hashicorp.com oldaláról töltöttem le. Telepítésnél a legfontosabb lépés, hogy a telepítési lokációt fel kell venni a rendszer környezeti változójába. Ezt úgy tehetjük meg, ha Windows + R billentyűt nyomunk, majd beírjuk a megnyíló ablakba a következő sysdm.cpl parancsot, ami megjeleníti a Rendszer tulajdonságai ablakot. Környezeti változók gombra kattintva érjük el az ablakot, ahol űj változókat tudunk felvenni.

A képen szöveg, képernyőkép, Betűtípus, szoftver látható

Automatikusan generált leírás

A képen szöveg, képernyőkép, szoftver, Weblap látható

Automatikusan generált leírásMásodik lépésként az Azure hitelesítő adatokat konfigurálni kell ahhoz, hogy a Terraform hozzáférjen az Azure számlákhoz és erőforrásokat hozhasson létre. Az Azure App Registration, vagyis az Azure-ban történő alkalmazásregisztráció központi szerepet tölt be az Azure Active Directory (Azure AD) által nyújtott identitás- és hozzáférés-kezelési szolgáltatásokban. Ez a folyamat lehetővé teszi a fejlesztők számára, hogy biztonságosan regisztrálják és konfigurálják alkalmazásaikat az Azure AD-ben, így azok képesek lesznek az Azure AD által nyújtott autentikációs és autorizációs képességek kihasználására. Az Azure App Registration folyamatát általában az Azure portál Alkalmazásregisztrációk (App registrations) menüpontjában végezzük, ahol az alkalmazásokat regisztrálhatjuk, és beállíthatjuk a szükséges API hozzáférési engedélyeket, valamint konfigurálhatjuk az autentikációhoz és autorizációhoz szükséges egyéb beállításokat. Projektem esetében terraform néven regisztráltam az applikációt.

Következő lépésben az imént létrehozott alkalmazásunknak szerepkört (role) rendelünk, amelyet az Add Role Assignment tesz lehetővé. Ezt a Subscription alatt az Access Controll menüpontban találjuk. Ezáltal szabályozható, hogy ki milyen jogosultságokkal rendelkezzen az Azure erőforrások kezelésére. Szerepkörök hozzárendelése lényeges az erőforrásokhoz való hozzáférés biztonságos és ellenőrzött módon történő kezeléséhez. Ennek főbb céljai:

1. Hozzáférés-szabályozás: Meghatározza, hogy a felhasználók, csoportok, vagy alkalmazási identitások milyen műveleteket végezhetnek el az Azure-ban. Lehetővé teszi például, hogy csak bizonyos felhasználók hozhassanak létre vagy kezelhessenek virtuális gépeket vagy hálózati szolgáltatásokat.
2. Szerepkör-alapú hozzáférés-vezérlés (RBAC - Azure Role-Based Access Control): Az RBAC segítségével finomra hangolt hozzáférési szabályokat állíthatsz be, amelyek meghatározzák, hogy a szervezeted különböző tagjai milyen tevékenységeket hajthatnak végre az Azure erőforrásaival.
3. Automatizálás és skálázás: Az Azure szerepkörök hozzárendelése automatizálható, lehetővé téve a nagyobb skálájú felhasználói és erőforrás-kezelést anélkül, hogy minden egyes felhasználó vagy erőforrás számára egyenként kellene konfigurálni a hozzáférési jogosultságokat.
4. Biztonsági megfelelőség: A megfelelő szerepkörök hozzárendelésével biztosítható, hogy a szervezet megfeleljen a belső és külső biztonsági előírásoknak, csökkentve a nem megfelelő hozzáférésből eredő kockázatokat.

Jelen esetben, a „Contributor” szerepkört állítottam az terraform alkalmazásnak. A Contributor szerepkör lehetővé teszi az erőforrások széles körű kezelését, de korlátozásokkal. A közreműködők létrehozhatnak és kezelhetnek erőforrásokat, de nem rendelkeznek jogosultsággal az irányítási és biztonsági beállítások, például az RBAC szerepkör-hozzárendelések kezelésére. nem adhat hozzáférést más felhasználóknak az erőforrásokhoz, és nem rendelhetők további szerepkörök hozzárendelésére. Ez azt jelenti, hogy nem tudnak más felhasználókat "Contributor" vagy "Owner" szerepkörökkel ellátni. Ez a szerepkör ideális számomra, mivel csak az erőforrások kezelésére és konfigurálására van szükségem, de nincs szükség a hozzáférés-vezérlési beállítások módosítására.

Összegezve, az alkalmazás regisztrációjának és a szerepkörök hozzárendelésének befejezése után az Azure-ben most már minden előkészület megtörtént ahhoz, hogy folytatódjon a fejlesztés és elkezdődhessen a kód megírása. Ezek az előkészületek biztosítják, hogy az alkalmazás megfelelően integrálódjon az Azure szolgáltatásaival, és hogy megfelelő hozzáférési jogosultságokkal rendelkezzen az erőforrások kezeléséhez. Ezek az előfeltételek elengedhetetlenek az alkalmazás biztonságos és hatékony működéséhez, így most, hogy ezek a lépések teljesítve lettek, a projekt fókusza áthelyezhető a funkcionális kód megírására és az alkalmazás logikájának implementálására.

Projektem során 4 darab terraform konfigurációs fájlt fogok használni. A Terraform konfigurációs fájlok, mint a main.tf, providers.tf, variables.tf, és terraform.tfvars strukturált módon segítenek kezelni az infrastruktúra mint kód (IaC) projektet. Minden fájl különböző elemeket és beállításokat tartalmazhat, amelyek együttműködve határozzák meg és konfigurálják az infrastruktúrát. Íme mit tartalmazhatnak:

1. main.tf: Az infrastruktúrában létrehozandó erőforrások definícióit fogja tartalmazni, mint például virtuális gépek, hálózati komponensek stb. Az main.tf gyakran az a hely, ahol a Terraform projekt "fő" konfigurációja található.
2. providers.tf: A Terraform provider-ek, mint például az aws, azure, google stb., konfigurációját tartalmazza. Itt adhatjuk meg a provider verzióját, valamint azon beállításokat, amelyek az adott provider autentikációjához szükségesek.
3. variables.tf: Ez a fájl deklarálja azokat a változókat, amelyeket a Terraform projekt használ. Ezek a változók paraméterezik a konfigurációt, lehetővé téve, hogy a kód újra felhasználható és dinamikus legyen. Opcionálisan megadhatunk alapértelmezett értékeket a változókhoz, amelyeket a Terraform automatikusan használ, ha nincs megadva más érték.
4. terraform.tfvars: A variables.tf-ben deklarált változók tényleges értékeit tartalmazza. Ez a fájl lehetővé teszi a változók értékeinek különálló kezelését a többi Terraform konfigurációtól.

Ezek a fájlok együttműködnek a Terraform projekt strukturált és karbantartható kezeléséhez, lehetővé téve az infrastruktúra egyszerű definícióját, változtatását, és verziókezelését.

## providers.tf

A providers.tf fájl fontos szerepet játszik a Terraform projekt inicializálásában, mivel itt van definiálva az Azure Resource Manager (azurerm) provider konfigurációja. Ez a konfiguráció tartalmazza azokat a paramétereket és beállításokat, amelyek szükségesek az Azure erőforrások Terraformmal történő kezeléséhez. Íme, mit csinál ez a konfiguráció részletesen:

terraform {

required\_providers {

azurerm = {

source = "hashicorp/azurerm"

version = "3.95.0"

}

}

}

provider "azurerm" {

subscription\_id = "9485e662-9561-4893-9a75-0d40e8f34cf4"

client\_id = "fe82ff81-626a-43f2-8e31-fb820bcda123"

client\_secret = "rcF8Q~3Ui4cHtzK~4.H2LE45pOP0McT.0kFIlcyN"

tenant\_id = "3189c4b9-8fbc-4f43-bda6-fdba77119c88"

features {}

}

**Terraform blokk**: Meghatározza a szükséges providereket és azok verzióit. Ez biztosítja, hogy a projekt konzisztens maradjon, és mindenki ugyanazt a provider verziót használja, amikor a kódot futtatja vagy módosítja.

*azurerm*: Ez a bejegyzés az Azure Resource Manager szolgáltatás Terraform providerének konfigurációját tartalmazza, specifikusan a hashicorp/azurerm forrást és a 3.95.0 verziót. A hashicorp/azurerm hivatkozás jelzi, hogy a provider a HashiCorp által karbantartott hivatalos Terraform Registry-ből származik.

**Provider "azurerm" blokk**

1. *Subscription ID*: Az Azure előfizetés azonosítója. Ez az azonosító határozza meg, hogy melyik Azure előfizetésen belül lesznek kezelve az erőforrások.
2. *Client ID és Client Secret*: Ezek az Azure Active Directory alkalmazás (Service Principal) azonosítói, amelyek a Terraformnak az előfizetés erőforrásainak kezeléséhez szükséges jogosultságokat biztosítják. Ez a hitelesítési módszer lehetővé teszi a Terraform számára, hogy programozottan hozzáférjen és kezelje az Azure erőforrásokat.
3. *Tenant ID*: Az Azure AD előfizető azonosítója, ami az Azure Active Directory domain-hez kapcsolódik. Ez az azonosító szükséges a Service Principal kontextusának meghatározásához.
4. *Features {}:* Ez egy kötelező mező az azurerm providerben, amely jelenleg üresen marad (jelölve, hogy nincsenek speciális szolgáltatási beállítások meghatározva), de a jövőbeni frissítésekhez szükséges lehet konfigurációs opciókat tartalmaznia.

## terraform.tfvars

location = "North Europe"

resource\_group\_name = "appgrp"

admin\_username = "fleq8s"

admin\_password = "Azure@123"

storage\_account\_name = "appstore20240417"

Ebben az esetben az terraform.tfvars fájl az Azure környezettel kapcsolatos konfigurációs adatokat tartalmazza, amelyek automatizált módon kerülnek felhasználásra az infrastruktúra létrehozásakor.

**Változók és jelentőségük:**

*location*: Ez a változó határozza meg, hogy hol (melyik Azure régióban) lesznek létrehozva az erőforrások. A példában "North Europe" van megadva, ami az észak-európai Azure régiót jelöli. Ez fontos, mert a régió kiválasztása befolyásolhatja az erőforrások elérhetőségét, késleltetését és költségét.

*resource\_group\_name*: Az Azure erőforráscsoport neve, amelybe az összes létrehozott erőforrás tartozni fog. Ebben az esetben "appgrp" név lett megadva. Az erőforráscsoport logikai konténerként szolgál, amely csoportosítja az összes kapcsolódó Azure erőforrást, lehetővé téve egyszerűbb kezelést, költségbecslést és hozzáférés-szabályozást.

*admin\_username és admin\_password*: Ezek a változók rendszergazdai felhasználónevet és jelszót határoznak meg, amelyeket például virtuális gépek létrehozásakor használhatnak az adminisztratív hozzáférés biztosítására.

*storage\_account\_name*: Az Azure Storage fiók neve, amely az adattárolási szolgáltatásokhoz szükséges. Ebben az esetben "appstore20240417" lett beállítva. A Storage Account kulcsfontosságú a blobok, fájlok, lemezek, stb. tárolásában, és az egyedi név biztosítja, hogy az erőforrás globálisan egyedi legyen az Azure-ban.

**A terraform.tfvars fájl szerepe:**

Ez a fájl segít automatizálni és standardizálni az infrastruktúra létrehozását a Terraform használatával, mivel központilag kezeli az összes szükséges konfigurációs paramétert. A fájlban megadott értékek automatikusan beillesztődnek a Terraform konfigurációs fájlokba (\*.tf fájlok), amikor a terraform plan és terraform apply parancsok futnak. Ez a megközelítés csökkenti az emberi hibák kockázatát, gyorsítja az infrastruktúra telepítési folyamatát, és elősegíti a környezetek közötti konzisztenciát.

## variables.tf

variable "location" {

type = string

description = "The location where resources will be created."

}

variable "resource\_group\_name" {

type = string

description = "The name of the resource group."

}

variable "vm\_size" {

type = string

description = "The size of the Virtual Machine."

default = "Standard\_D2s\_v3"

}

variable "admin\_username" {

type = string

description = "The admin username for the Virtual Machine."

}

variable "admin\_password" {

type = string

description = "The admin password for the Virtual Machine."

}

variable "address\_space" {

type = list(string)

description = "The address space for the Virtual Network."

default = ["10.0.0.0/16"]

}

variable "subnet\_prefixes" {

type = list(string)

description = "The address prefixes for the subnet."

default = ["10.0.0.0/24"]

}

variable "storage\_account\_name" {

type = string

description = "The name of the storage account."

}

variable "container\_name" {

type = string

description = "The name of the storage container."

default = "data"

}

A variables.tf fájl Terraformban fundamentális szerepet tölt be, mivel lehetővé teszi a változók deklarálását, ami alapvető a konfigurálható és újrafelhasználható kód írásához. Ez a fájl biztosítja a különböző konfigurációs paraméterek paraméterezését és modularizácóját, amelyek a Terraform kódokban felhasználhatók. A variables.tf fájl a változók meghatározásának és dokumentálásának központi helye, segítve ezzel a kódok átláthatóságát és könnyebb kezelhetőségét.

**variables.tf és a terraform.tfvars összefüggése**:

A variables.tf biztosítja a "sablont" a változóknak: meghatározza a nevüket, típusukat, és opcionálisan alapértelmezett értékeiket.

A terraform.tfvars fájl ezeket a változókat tölti ki konkrét értékekkel, amelyeket a Terraform projektek futtatásakor használnak.

## main.tf

locals {

resource\_group = var.resource\_group\_name

}

resource "azurerm\_resource\_group" "app\_grp"{

name = local.resource\_group

location = var.location

}

A bemutatott main.tf kódrészlet a Terraform használatával leírja, hogyan hozható létre egy erőforráscsoport az Azure-ban. A kód egyszerű, de fontos szerepet tölt be az Azure infrastruktúra automatizált létrehozásánál.

**locals**:

A locals blokk segítségével definiálhatók a lokális változók, amelyek a kódon belül többször is felhasználhatók a redundancia elkerülése és a kód karbantarthatóságának javítása érdekében. Itt egy lokális változót (resource\_group) hoztam létre, amelynek értéke a var.resource\_group\_name változóból származik. Ez a változó egy külsőleg definiált változó, ami a terraform.tfvars-ban van definiálva.

**azurerm\_resource\_group**

A resource szóval kezdődő blokkban egy új erőforrást hoznak létre. Itt a azurerm\_resource\_group típusú erőforrás az, amelyet létrehoznak, és az "app\_grp" azonosítóval rendelkezik a Terraform állapotfájljában. Az erőforráscsoport neve a korábban definiált lokális változó (local.resource\_group) értékére van állítva, és a földrajzi helyszín a var.location változóval van meghatározva, ami szintén külsőleg definiált változó.

Összefoglalva, ez a kód részlet tehát egy erőforráscsoportot hoz létre az Azure-ban a *appgrp* névvel és *North Europe* helyszínnel. Az erőforráscsoportok alapvető konténerek az Azure-ban, amelyek csoportosítják az erőforrásokat logikai egységekké, így megkönnyítve az erőforrások kezelését, költségfigyelését és hozzáférés-szabályozását. Ez a struktúra alapvetően fontos az Azure-ban végzett infrastruktúra-műveletek számára, mivel lehetővé teszi az erőforrások csoportosítását és rendszerezett kezelését a Terraform segítségével. A var és local használatával a kód modularitása és újrafelhasználhatósága is javul, mivel lehetővé teszi a paraméterek egyszerű módosítását külső változók és konfigurációk segítségével.

**azurerm\_virtual\_network**

resource "azurerm\_virtual\_network" "app\_network" {

name = "app-network"

location = var.location

resource\_group\_name = azurerm\_resource\_group.app\_grp.name

address\_space = var.address\_space

depends\_on = [

azurerm\_resource\_group.app\_grp

]

}

Ez az erőforrás egy virtuális hálózatot (Virtual Network vagy VNet) hoz létre az Azure-ban. A virtuális hálózat alapvetően egy izolált hálózat az Azure-ban, amelyen belül az erőforrások kommunikálhatnak egymással. Itt definiált tulajdonságok:

* + - * + *name*: Az erőforrás neve az Azure-ban, jelen esetben "app-network".
        + *location*: A földrajzi hely, ahol a hálózat létre lesz hozva, értékét a var.location változóból kapja. Ez biztosítja, hogy a tárolói fiók ugyanabban a helyszínen legyen, mint az erőforráscsoport.
        + *resource\_group\_name*: Az erőforrás-csoport neve, amely alá a virtuális hálózat tartozik, ezt az azurerm\_resource\_group.app\_grp.name érték adja (appgrp).
        + *address\_space*: Az address space vagy „címtér”, amelyet a virtuális hálózat használni fog. Ennek az értékét a variables.tf-ből veszi, értéke: 10.0.0.0/16

**azurerm\_subnet**

resource "azurerm\_subnet" "subnet\_a" {

name = "subnet\_a"

resource\_group\_name = azurerm\_resource\_group.app\_grp.name

virtual\_network\_name = azurerm\_virtual\_network.app\_network.name

address\_prefixes = var.subnet\_prefixes

depends\_on = [

azurerm\_virtual\_network.app\_network

]

}

Egy alhálózatot (subnet) hoz létre a fenti virtuális hálózaton belül. Alhálózatokat arra használják, hogy tovább bontsák a hálózati teret kisebb, kezelhetőbb egységekre.

* + - * + *name:* Az alhálózat neve, jelen esetben "subnet\_a".
        + *resource\_group\_name:* Megadja, hogy melyik erőforrás-csoportba tartozik az alhálózat.
        + *virtual\_network\_name:* A virtuális hálózat neve, amelyen belül az alhálózat létrejön.
        + *address\_prefixes:* Az alhálózat IP cím tartományai, amelyeket a var.subnet\_prefixes változó tartalmaz.

**azurerm\_network\_interface**

resource "azurerm\_network\_interface" "app\_interface1" {

name = "app-interface1"

location = azurerm\_resource\_group.app\_grp.location

resource\_group\_name = azurerm\_resource\_group.app\_grp.name

ip\_configuration {

name = "internal"

subnet\_id = azurerm\_subnet.subnet\_a.id

private\_ip\_address\_allocation = "Dynamic"

}

depends\_on = [

azurerm\_virtual\_network.app\_network,

azurerm\_subnet.subnet\_a

]

}

resource "azurerm\_network\_interface" "app\_interface2" {

name = "app-interface2"

location = azurerm\_resource\_group.app\_grp.location

resource\_group\_name = azurerm\_resource\_group.app\_grp.name

ip\_configuration {

name = "internal"

subnet\_id = azurerm\_subnet.subnet\_a.id

private\_ip\_address\_allocation = "Dynamic"

}

depends\_on = [

azurerm\_virtual\_network.app\_network,

azurerm\_subnet.subnet\_a

]

}

Két hálózati interfész (network interface) van létrehozva, amelyek virtuális gépekhez kapcsolódnak, lehetővé téve azok kommunikációját a hálózaton keresztül.

* + - * + *name:* Az interfész neve ("app-interface1" és "app-interface2").
        + *location:* Az erőforrás földrajzi helyszíne.
        + *resource\_group\_name:* Az erőforrás-csoport, amely alá az interfész tartozik.
        + *ip\_configuration:* Az IP konfiguráció részletei, mint a neve, a hozzá tartozó alhálózat azonosítója (subnet\_id), és hogy dinamikus vagy statikus IP címet kapjon-e az interfész (private\_ip\_address\_allocation).

Függőségek

A depends\_on attribútumok arra szolgálnak, hogy biztosítsák az erőforrások létrehozásának sorrendjét. Például az interfészek létrehozása előtt szükség van a virtuális hálózat és alhálózat létrehozására, ezért ezeket megelőzően nem hozhatók létre az interfészek. Ez garantálja, hogy minden hálózati komponens megfelelően kapcsolódjon egymáshoz és az erőforrás-függőségek kezelése során ne jelentkezzenek problémák.

**azurerm\_windows\_virtual\_machine**

resource "azurerm\_windows\_virtual\_machine" "app\_vm1" {

name = "njevm01"

resource\_group\_name = azurerm\_resource\_group.app\_grp.name

location = azurerm\_resource\_group.app\_grp.location

size = var.vm\_size

admin\_username = var.admin\_username

admin\_password = var.admin\_password

availability\_set\_id = azurerm\_availability\_set.app\_set.id

network\_interface\_ids = [

azurerm\_network\_interface.app\_interface1.id,

]

os\_disk {

caching = "ReadWrite"

storage\_account\_type = "Standard\_LRS"

}

source\_image\_reference {

publisher = "MicrosoftWindowsServer"

offer = "WindowsServer"

sku = "2019-Datacenter"

version = "latest"

}

depends\_on = [

azurerm\_network\_interface.app\_interface1,

azurerm\_availability\_set.app\_set

]

}

A fentebb bemutatott Terraform kód egy Windows virtuális gép (VM) erőforrás meghatározására és konfigurálására szolgál. A bemutatott kódrészlet csupán egy virtuális gép konfigurációját tartalmazza, azonban van egy másik, hasonló beállításokat használó erőforrás is, amely egy második VM létrehozására szolgál. Mindkét virtuális gép azonos paraméterekkel van konfigurálva, biztosítva ezzel a környezet egységes működését.

Erőforrás deklaráció:

* + - * + *resource "azurerm\_windows\_virtual\_machine" "app\_vm1":* Ez a sor egy új erőforrás típusú deklarációt tartalmaz, amely az Azure Windows VM erőforrását jelöli. Az erőforrás példányának neve app\_vm1.

Konfigurációs attribútumok:

* + - * + *name:* Beállítja az Azure-ban a VM nevét "njevm01"-re.
        + *resource\_group\_name*: Megadja az erőforráscsoport nevét, amelyben a VM létrehozásra kerül.
        + *location:* Meghatározza azt az Azure régiót, ahol a VM elhelyezkedik.
        + size: Megadja a VM méretét (pl. a típus, amely meghatározza a CPU-k számát, a memória mennyiségét stb.). Ezt a méretet egy variables.tf fájban definiáltam, amelynek értéke „Standard\_D2s\_v3”, ami 3 CPU-t, 32 GiB RAM-ot és 200 GiB SSD-t biztosít a VM-nek.
        + *admin\_username és admin\_password*: Ezek a hitelesítő adatok az adminisztrátori fiók létrehozására szolgálnak a Windows VM-en, ezt a terraform.tfvars-ban adtam meg..
        + *availability\_set\_id*: Ez a beállítás kapcsolja a virtuális gépet egy meghatározott elérhetőségi készlethez (availability set). Az elérhetőségi készletek az Azure-ban azért használatosak, hogy magasabb rendelkezésre állást biztosítsanak a VM-ek számára azzal, hogy azokat több fizikai szerver és fault domain között osztják el. Ez csökkenti a VM-ek egyidejű kiesésének kockázatát, amely hardver- vagy infrastruktúra karbantartásokból adódhat. Az availability\_set\_id attribútummal megadott azonosító azt jelzi, hogy a VM melyik elérhetőségi készlethez tartozik, tehát melyik csoportban kezelik a rendelkezésre állását.
        + *network\_interface\_ids*: Egy tömb, amely felsorolja a VM-hez csatlakoztatott hálózati interfészeket. Ebben az esetben egy hálózati interfész szerepel, amelyet app\_interface1.id azonosít.

Operációs rendszer lemez konfigurációja:

* + - * + *Caching*: Ez a beállítás az operációs rendszer lemezén alkalmazott gyorsítótárazási módot határozza meg. A "ReadWrite" opció lehetővé teszi az adatok írását és olvasását a gyorsítótárba, ami javítja az I/O műveletek teljesítményét. Ez különösen hasznos lehet gyakran elérhető fájlok esetén, mivel csökkenti a lemezhez való hozzáférés idejét.
        + *storage\_account\_type*: Ez a beállítás az operációs rendszer lemezének tárolási fiók típusát adja meg. A "Standard\_LRS" (Standard Locally-Redundant Storage) azt jelenti, hogy a tárolt adatok több fizikai lemezen is tárolásra kerülnek az adatközpontban, ami növeli az adatok elérhetőségét és ellenállóságát a lemezhibákkal szemben. A helyben redundáns tárolás egy költséghatékony megoldás, amely biztosítja az adatokat három különböző fizikai lemezen ugyanazon a földrajzi helyen.

Image konfigurációja:

Windows virtuális gép alapjául szolgáló image forrásának meghatározására szolgál. A kód megadja az operációs rendszer image fájljainak pontos specifikációját, amelyet a virtuális gép telepítésekor használ Függőségek:

Összegezve, ez a kódblokk alapvetően utasítja az Azure-t, hogy hozzon létre egy Windows Server 2019 Datacenter VM-et a megadott teljesítmény-konfigurációkkal, tárolási beállításokkal és hálózati kapcsolatokkal, biztosítva, hogy része legyen a meghatározott magas rendelkezésre állású és hálózati beállításoknak.

**azurerm\_availability\_set**

resource "azurerm\_availability\_set" "app\_set" {

name = "app-set"

location = azurerm\_resource\_group.app\_grp.location

resource\_group\_name = azurerm\_resource\_group.app\_grp.name

platform\_fault\_domain\_count = 3

platform\_update\_domain\_count = 3

depends\_on = [

azurerm\_resource\_group.app\_grp

]

}

Ez a Terraform kódrészlet egy Azure availability set-et (elérhetőségi készletet) hoz létre és konfigurál, amely rendkívül fontos szerepet játszik az alkalmazások magas rendelkezésre állásának biztosításában Azure környezetben. Az alábbiakban részletezem, hogy mit jelentenek a kódban szereplő egyes részek:

Erőforrás deklaráció:

* + - * + *resource "azurerm\_availability\_set" "app\_set":* Ez a sor egy új erőforrást deklarál azurerm\_availability\_set típussal, amely az elérhetőségi készleteket kezeli Azure-ban. Az erőforrás neve app\_set.

Konfigurációs paraméterek:

* + - * + *name = "app-set":* Az elérhetőségi készlet neve, amit Azure-ban használni fognak.
        + location = azurerm\_resource\_group.app\_grp.location: Az elérhetőségi készlet földrajzi helye, ami megegyezik az azt tartalmazó erőforráscsoport helyével (app\_grp).
        + *resource\_group\_name = azurerm\_resource\_group.app\_grp.name:* Az erőforráscsoport neve, amelyhez az elérhetőségi készlet tartozik.
        + *platform\_fault\_domain\_count = 3*: A hibadomainek száma az elérhetőségi készleten belül. A hibadomének olyan fizikai területek az adatközpontban, amelyek függetlenül kezelik az energiaellátást és a hálózati kapcsolatot, ezáltal csökkentve az egységes hibapontok kockázatát.
        + *platform\_update\_domain\_count = 3:* Az frissítési domainek száma. Az frissítési domének lehetővé teszik, hogy az Azure karbantartásai során csak a készlet egy része legyen leállítva egyszerre, így a többi rész továbbra is elérhető marad.

Összefoglalva, ez a kód egy elérhetőségi készletet hoz létre, amely segít javítani az alkalmazások rendelkezésre állását, kezelni a hibatűrést és az üzemeltetési frissítések hatásait az Azure virtuális gépei számára.

Az alábbiakban részletesen bemutatom azokat az Azure erőforrásokat, amelyek szükségesek két load balancer létrehozásához és egy webalkalmazás kezeléséhez Azure környezetben.

A webalkalmazás magas rendelkezésre állásának és skálázhatóságának biztosítása érdekében két Azure Load Balancer instance kerül implementálásra, melyek dinamikusan kezelik a bejövő hálózati forgalmat és optimalizálják a terheléselosztást. A rendszer kialakításához szükséges további Azure erőforrások között szerepelnek a nyilvános IP-címek, melyek a load balancer-ek elérhetőségét teszik lehetővé az internet felől, valamint hálózati biztonsági csoportok (Network Security Group), amelyek szabályozzák a bejövő és kimenő hálózati forgalmat, növelve ezzel a rendszer biztonságát.

Minden load balancer egy-egy frontend IP konfigurációval rendelkezik, amely a nyilvános IP címekhez kapcsolódik, és a bejövő kapcsolatok fogadására szolgál. A backend csoportok (Backend Address Pool) tartalmazzák azokat a szervereket, amelyek a kérések feldolgozását végzik, biztosítva az alkalmazás folyamatos működését. Az infrastruktúra állapotának monitorozása érdekében egészségügyi vizsgálatok (Load Balancer Probe) kerülnek beállításra, amelyek ellenőrzik a backend szerverek elérhetőségét és segítenek a hibás szerverek kiszűrésében. Ezenfelül, szabályok (Load Balancer Rule) definiálására kerül sor, amelyek meghatározzák, hogy a kérések melyik szerverekre legyenek irányítva.

A rendszerhez kapcsolódóan létrehoztam egy Storage Account is, amely a naplózás és adatok tárolására szolgál, valamint a VM-ekhez kapcsolódó kiterjesztések (Extensions), amelyek automatizálják a konfigurációs scriptek futtatását és a biztonsági frissítéseket. A Storage Account részeként Blob tárolók is implementálásra kerülnek, ahol a naplófájlok és egyéb adatok tárolhatók.

Ezek az erőforrások és konfigurációk együtt biztosítják, hogy a webalkalmazás gyorsan és hatékonyan kezelje a különböző terheléseket, miközben biztosítja a rendszer stabilitását és biztonságát a felhőalapú infrastruktúrában. (Microsoft, 2024/c)

**azurerm\_storage\_account**

resource "azurerm\_storage\_account" "app\_store" {

name = var.storage\_account\_name

resource\_group\_name = azurerm\_resource\_group.app\_grp.name

location = azurerm\_resource\_group.app\_grp.location

account\_tier = "Standard"

account\_replication\_type = "LRS"

#allow\_blob\_public\_access = true

}

Az fentebb bemutatott Terraform kódrészlet tárolói fiók (storage account) konfigurációját írja le. Ez a tárolói fiók az Azure-ban használt adatok tárolására szolgál, legyen szó virtuális gépek disk-jeiről, blob tárolókról, fájlok megosztásáról, vagy akár archivált adatokról. Az alábbiakban részletezem a kód egyes részeit és azok funkcióit.

Erőforrás deklaráció:

* + - * + *resource "azurerm\_storage\_account" "app\_store":* Ez a sor deklarálja az erőforrást a Terraformban, ami ebben az esetben egy `azurerm\_storage\_account` típusú Azure Storage Account. Az "app\_store" egy belső azonosító, amellyel a Terraformon belül hivatkozhatunk erre az erőforrásra.

Konfigurációs attribútumok:

* + - * + *name = var.storage\_account\_name:* A tárolói fiók neve, amely egy változón keresztül van megadva (`storage\_account\_name`). Ez a név egyedinek kell lennie az Azure régiójában.
        + *resource\_group\_name = azurerm\_resource\_group.app\_grp.name:* Meghatározza, hogy melyik erőforráscsoportba kerül a tárolói fiók.
        + *location = azurerm\_resource\_group.app\_grp.location:* A tárolói fiók földrajzi helyét adja meg.

Tárolási paraméterek:

* + - * + *account\_tier = "Standard":* A tárolói fiók szolgáltatási szintjét jelöli, amely ebben az esetben a "Standard" szint. A Standard szint általánosan a leggyakrabban használt adattárolási opció, amely kiegyensúlyozott teljesítményt és költséghatékonyságot nyújt.
        + account\_replication\_type = "LRS": Az adatok replikációjának típusa, itt "LRS" azaz Locally Redundant Storage, ami azt jelenti, hogy az adatokat ugyanazon a földrajzi helyen több eszközön tárolják, így biztosítva az adatok helyi redundanciáját.

Ez a Terraform kódrészlet tehát egy fontos infrastrukturális elemet hoz létre az Azure környezetedben, amely a különböző alkalmazások és szolgáltatások adattárolási igényeit szolgálja ki. A kód egyszerű, de nagyon fontos beállításokat tartalmaz, amelyek befolyásolják a tárolói fiók teljesítményét, költségeit és biztonságát.

**azurerm\_storage\_container**

resource "azurerm\_storage\_container" "data" {

name = "data"

storage\_account\_name = var.storage\_account\_name

container\_access\_type = "blob"

depends\_on=[

azurerm\_storage\_account.app\_store

]

}

A következő kódrészlet egy Azure Storage Container létrehozására szolgál a fentebb létrehozott Storage Account keretein belül. Ez a konténer adatok tárolására szolgál, amely különböző alkalmazásokból és folyamatokból származó blob adatok kezelésére használható.

Erőforrás deklaráció:

* + - * + *resource "azurerm\_storage\_container" "data":* Ez a sor deklarálja az erőforrást a Terraformban, amely ebben az esetben egy azurerm\_storage\_container típusú Azure Storage Container. Az "data" a belső azonosító, amellyel a Terraformon belül hivatkozhatunk erre az erőforrásra.

Konfigurációs attribútumok:

* + - * + *name = "data":* A tároló konténer neve, amely egyértelműen azonosítja azt az adott Storage Accounton belül. Ebben az esetben a konténer neve egyszerűen "data".
        + *storage\_account\_name = var.storage\_account\_name:* A tárolói fiók neve, amelyhez a konténer tartozik. Ez a név egy változón (var.storage\_account\_name) keresztül van megadva, ami lehetővé teszi, hogy a fiók neve dinamikusan, külső konfigurációból származzon.
        + *container\_access\_type = "blob"*: Ez a beállítás meghatározza a konténer hozzáférési típusát, ami jelen esetben "blob". Ez azt jelenti, hogy a konténerben tárolt objektumok (blobok) hozzáférése külön hozzáférési szabályok alapján történik. A "blob" típus általában nyilvános olvasást engedélyez, hacsak nem korlátozzák külön.

Függőségek:

* + - * + *depends\_on = [azurerm\_storage\_account.app\_store]:* Ez a beállítás biztosítja, hogy a konténer létrehozása csak akkor történjen meg, ha a hozzá tartozó Storage Account (amelyet app\_store néven definiáltak az erőforrások között) már létrejött.

Ez a kódrészlet tehát egy Storage Container létrehozását írja le egy Azure Storage Accounton belül, ahol a tárolt adatok blob formátumban kerülnek kezelésre. A konfiguráció megfelelően biztosítja a hozzáférés kezelését és a függőségek rendezését, amelyek elengedhetetlenek a zökkenőmentes és hatékony infrastruktúra üzemeltetésében.

**azurerm\_storage\_blob**

resource "azurerm\_storage\_blob" "scripts\_a" {

name = "scripts.zip"

storage\_account\_name = var.storage\_account\_name

storage\_container\_name = var.container\_name

type = "Block"

source = "scripts/scripts.zip"

depends\_on=[azurerm\_storage\_container.data]

}

Ez a Terraform kódrészlet egy Azure Storage Blob erőforrás létrehozását írja le, amely egy fentebb létrehozott tároló konténerben helyezkedik el. Ebben az esetben egy 'scripts.zip' nevű blob állományt hoz létre, amelyet egy meglévő tároló konténerbe helyez. A kód számos fontos beállítást tartalmaz, amelyek lehetővé teszik a blob fájl konfigurálását és kezelését.

Erőforrás deklaráció:

- *resource "azurerm\_storage\_blob" "scripts\_a":* Ez a sor deklarálja az erőforrást a Terraformban, amely ebben az esetben egy azurerm\_storage\_blob típusú Azure Storage Blob. Az "scripts\_a" a belső azonosító, amellyel a Terraformon belül hivatkozhatunk erre az erőforrásra.

Konfigurációs attribútumok:

* + - * + *name = "scripts.zip":* A blob fájl neve, amelyet a tároló konténerben tárolnak. Ez azonosítja a fájlt a konténeren belül.
        + *storage\_account\_name = var.storage\_account\_name:* A tárolói fiók neve, amelyhez a blob tartozik. Ez a név egy változón (var.storage\_account\_name) keresztül van megadva, ami lehetővé teszi, hogy a fiók neve dinamikusan, külső konfigurációból származzon.
        + *storage\_container\_name = var.container\_name:* A tároló konténer neve, ahol a blob elhelyezkedik. Ez szintén egy változón (var.container\_name) keresztül van megadva.
        + *type = "Block":* A blob típusa, jelen esetben "Block" típusú blob, ami általánosan használt nagy méretű fájlok tárolására. Block blobok ideálisak olyan adatokhoz, amelyeket gyakran kell frissíteni vagy feldolgozni.

Forrás:

* + - * + source = "scripts/scripts.zip": Az útvonal, ahonnan a blob tartalmát betölti. Ez jelzi, hogy a helyi vagy távoli forrásból származó 'scripts.zip' fájlt kell feltölteni a blobba. Jelen esetben a GitHub-on található útvonal van megadva (<https://github.com/adambaksa/LB/tree/main/scripts>)

Függőségek:

- *depends\_on = [azurerm\_storage\_container.data]:* Ez a beállítás biztosítja, hogy a blob létrehozása csak akkor történjen meg, ha a hozzá tartozó tároló konténer (data néven definiált) már létrejött.

Ez a kódrészlet tehát egy konkrét blob fájl létrehozását és konfigurálását írja le egy Azure Storage Accounton belül, ahol a tárolt adatok kezelése, frissítése és hozzáférhetősége kulcsfontosságú. A konfiguráció megfelelően biztosítja a blob adatok kezelését és a függőségek rendezését, ami létfontosságú a zökkenőmentes és hatékony adattárolás érdekében.

Ezen erőforrások összefüggése abban nyilvánul meg, hogy egy Storage Accountot (azurerm\_storage\_account) kell először létrehozni, amely alapot biztosít a további szervezési és adattárolási struktúrák számára. Ezen belül hozhatók létre tároló konténerek (azurerm\_storage\_container), amelyek specifikus adatcsoportokat kezelnek, és ezeken belül helyezkednek el a tényleges adatok, vagyis a blobok (azurerm\_storage\_blob). Minden szint az előzőre épül, és az adattárolás hierarchikus modelljét követi, amely lehetővé teszi az adatok hatékony kezelését, szervezését, és biztonságát az Azure platformon. (Hashicorp, 2024/b)

**azurerm\_virtual\_machine\_extension**

resource "azurerm\_virtual\_machine\_extension" "vm\_extension1" {

name = "appvm-extension01"

virtual\_machine\_id = azurerm\_windows\_virtual\_machine.app\_vm1.id

publisher = "Microsoft.Compute"

type = "CustomScriptExtension"

type\_handler\_version = "1.10"

depends\_on = [

azurerm\_storage\_blob.scripts\_a

]

settings = <<SETTINGS

{

"fileUris": ["https://${azurerm\_storage\_account.app\_store.name}.blob.core.windows.net/data/scripts.zip"],

"commandToExecute": "powershell -ExecutionPolicy Unrestricted -Command \"Invoke-WebRequest -Uri https://${azurerm\_storage\_account.app\_store.name}.blob.core.windows.net/data/scripts.zip -OutFile C:\\scripts.zip; Expand-Archive -Path C:\\scripts.zip -DestinationPath C:\\inetpub\\wwwroot\\scripts; Remove-Item -Path C:\\scripts.zip; & 'C:\\inetpub\\wwwroot\\scripts\\scripts\\IIS\_Config.ps1'\""

}

SETTINGS

}

A következő Terraform kódrészlet az njevm01 virtuális géphez rendel egy "CustomScriptExtension" típusú kiterjesztést. Ez a kiterjesztés automatizálja a szkriptek és konfigurációs fájlok végrehajtását a virtuális gépen. Ez a megoldás különösen hasznos a webalkalmazások telepítésénél. A kódban leírt vm\_extension1 erőforrás mellett létezik még egy hasonló kiterjesztés, amely vm\_extension02 néven van definiálva. Ez a második erőforrás gyakorlatilag ugyanazt a funkcionalitást valósítja meg, mint az első.

A két azurerm\_virtual\_machine\_extension erőforrás közötti különbség csupán az azonosítókban és néhány referencia névben van. Mindkét erőforrás ugyanazon eljárásokat követi a kód végrehajtása során, beleértve a szkriptek letöltését, kicsomagolását és a megfelelő konfigurációs parancsok futtatását. Az első kiterjesztés beállítása és működése mintaként szolgált a második kiterjesztés létrehozásához. A kódban leírt lépések részletezése a következő:

Erőforrás deklaráció:

* + - * + *resource "azurerm\_virtual\_machine\_extension" "vm\_extension1"*: Ez jelöli az erőforrást a Terraformban, ami egy virtuális géphez rendelt kiterjesztést hoz létre. A "vm\_extension1" egy belső azonosító, amely a projektben a kiterjesztésre hivatkozik.

Konfigurációs attribútumok:

* + - * + *name = "appvm-extension01":* A kiterjesztés neve.
        + *virtual\_machine\_id = azurerm\_windows\_virtual\_machine.app\_vm1.id:* A virtuális gép azonosítója, amelyre a kiterjesztést alkalmazzák.
        + *publisher = "Microsoft.Compute"*: A kiterjesztés kiadója, jelen esetben a Microsoft Compute.
        + *type = "CustomScriptExtension":* A kiterjesztés típusa, ami jelen esetben egy egyéni szkript végrehajtását teszi lehetővé.
        + *type\_handler\_version* *= "1.10":* A kiterjesztés verziója, amely meghatározza, hogy melyik szoftververziót használja a kiterjesztés.

Függőségek:

- *depends\_on = [azurerm\_storage\_blob.scripts\_a]:* Ez a beállítás biztosítja, hogy a kiterjesztés csak akkor kerüljön telepítésre, ha a megadott Storage Blob (amely a szükséges szkripteket és fájlokat tartalmazza) már létrejött és elérhető.

Beállítások:

A settings attribútum egy JSON formátumú szöveg, amely specifikálja a kiterjesztés konfigurációját. Itt többek között a következőket állítja be:

- *fileUris*: A szkriptek és a konfigurációs fájlok elérési útját tartalmazza, ami egy publikus URL a Storage Blobhoz.

- commandToExecute: A végrehajtandó parancsok sorozata, amely letölti a scripts.zip fájlt, kicsomagolja a megfelelő mappába, eltávolítja a letöltött zip fájlt, majd futtatja az IIS\_Config.ps1 PowerShell szkriptet, amely konfigurálja a web szerver beállításait, illetve elhelyezi a sap-html mappát a megfelelő helyen, ami tartalmazza az összes fájlt (html, css, js), ami a webalkalmazáshoz szükséges.

Ez a kódrészlet tehát egy összetett automatizálási folyamatot ír le, amely a webalkalmazás telepítését és konfigurációját végzi el az njevm01-en. A kiterjesztés használatával a felhasználók képesek automatizált módon, manuális beavatkozás nélkül telepíteni és konfigurálni szerveroldali alkalmazásokat, ami jelentősen csökkenti az üzembe helyezési időt és növeli a telepítési folyamatok megbízhatóságát.

**azurerm\_network\_security\_group és azurerm\_subnet\_network\_security\_group\_association**

resource "azurerm\_network\_security\_group" "app\_nsg" {

name = "app-nsg"

location = azurerm\_resource\_group.app\_grp.location

resource\_group\_name = azurerm\_resource\_group.app\_grp.name

security\_rule {

name = "Allow\_HTTP"

priority = 200

direction = "Inbound"

access = "Allow"

protocol = "Tcp"

source\_port\_range = "\*"

destination\_port\_range = "80"

source\_address\_prefix = "\*"

destination\_address\_prefix = "\*"

}

}

resource "azurerm\_subnet\_network\_security\_group\_association" "nsg\_association" {

subnet\_id = azurerm\_subnet.subnet\_a.id

network\_security\_group\_id = azurerm\_network\_security\_group.app\_nsg.id

depends\_on = [

azurerm\_network\_security\_group.app\_nsg

]

}

A kódrészlet az Azure környezetben két fő erőforrás létrehozását írja le Terraform segítségével.

Erőforrás deklaráció és konfigurációi:

* + - * + *resource "azurerm\_network\_security\_group" "app\_nsg":* Ez az erőforrás egy NSG-t hoz létre app-nsg néven, amely biztonsági szabályokat tartalmaz a forgalom szabályozására. Ebben az esetben egy Allow\_HTTP nevű szabály engedélyezi a bejövő TCP forgalmat az 80-as portra (ami a HTTP szolgáltatásokhoz szükséges). A szabály minden forrásból és minden címről fogadja a forgalmat, biztosítva, hogy a webes kérések elérjék az erőforrásokat.
        + *resource "azurerm\_subnet\_network\_security\_group\_association" "nsg\_association":* Ez a konfiguráció az előzőleg létrehozott NSG-t rendeli hozzá egy meghatározott alhálózathoz (azurerm\_subnet.subnet\_a.id). A hozzárendelés biztosítja, hogy az NSG által definiált szabályok alkalmazásra kerüljenek az adott alhálózaton, növelve az ott futó alkalmazások biztonságát.

Ez a konfiguráció alapvető az infrastruktúra biztonságának megteremtésében, mivel lehetővé teszi a szabályozott hozzáférést a hálózati erőforrásokhoz, és védelmet nyújt a nem kívánt hálózati forgalommal szemben.

**azurerm\_public\_ip és azurerm\_lb**

resource "azurerm\_public\_ip" "load\_ip" {

name = "load-ip"

location = azurerm\_resource\_group.app\_grp.location

resource\_group\_name = azurerm\_resource\_group.app\_grp.name

allocation\_method = "Static"

sku = "Standard"

}

resource "azurerm\_lb" "app\_balancer" {

name = "app-balancer"

location = azurerm\_resource\_group.app\_grp.location

resource\_group\_name = azurerm\_resource\_group.app\_grp.name

frontend\_ip\_configuration {

name = "frontend-ip"

public\_ip\_address\_id = azurerm\_public\_ip.load\_ip.id

}

sku="Standard"

depends\_on = [

azurerm\_public\_ip.load\_ip

]

}

Erőforrás deklaráció és konfigurációi:

* + - * + *azurerm\_public\_ip*: Egy statikus nyilvános IP-címet hoz létre load-ip néven. Ez az IP-cím a terheléselosztóhoz kapcsolódik, és lehetővé teszi az internetes felhasználók számára, hogy elérjék az alatta futó alkalmazásokat vagy szolgáltatásokat. A statikus kiosztási módszer biztosítja, hogy az IP-cím állandó maradjon, ami fontos a kiszolgáló infrastruktúra stabilitása és elérhetősége szempontjából.
        + *azurerm\_lb:* Egy app-balancer nevű terheléselosztót hoz létre, amely a nyilvános IP-címhez kapcsolódik a frontend\_ip\_configuration beállításon keresztül. Ez a terheléselosztó felelős a beérkező kérések fogadásáért és azok egyenletes elosztásáért a rendelkezésre álló erőforrások között, ezáltal növelve a rendszer megbízhatóságát és skálázhatóságát. A terheléselosztó "Standard" SKU-ja magasabb szolgáltatási szintet és fejlettebb funkciókat biztosít, mint az alapmodell.

Összefoglalva, ez a konfiguráció kardinális a nagy rendelkezésre állású, skálázható webes alkalmazások és szolgáltatások infrastruktúrájának kialakításában. A statikus nyilvános IP-cím stabil elérhetőséget biztosít, míg a terheléselosztó optimalizálja a hálózati forgalmat és javítja a teljesítményt, lehetővé téve, hogy az alkalmazások hatékonyan kezeljék a különböző terheléseket és forgalmi csúcsokat.

**azurerm\_lb\_backend\_address\_pool\_address**

resource "azurerm\_lb\_backend\_address\_pool\_address" "appvm1\_address" {

name = "njevm01"

backend\_address\_pool\_id = azurerm\_lb\_backend\_address\_pool.pool\_a.id

virtual\_network\_id = azurerm\_virtual\_network.app\_network.id

ip\_address = azurerm\_network\_interface.app\_interface1.private\_ip\_address

depends\_on = [

azurerm\_lb\_backend\_address\_pool.pool\_a

]

}

resource "azurerm\_lb\_backend\_address\_pool\_address" "appvm2\_address" {

name = "njevm02"

backend\_address\_pool\_id = azurerm\_lb\_backend\_address\_pool.pool\_a.id

virtual\_network\_id = azurerm\_virtual\_network.app\_network.id

ip\_address = azurerm\_network\_interface.app\_interface2.private\_ip\_address

depends\_on = [

azurerm\_lb\_backend\_address\_pool.pool\_a

]

}

A kód két azurerm\_lb\_backend\_address\_pool\_address erőforrást definiál, amelyek hozzárendelik a virtuális gépek IP-címeit a terheléselosztó backend címmedencéjéhez. Minden erőforrás egy-egy virtuális gépet képvisel a hálózaton belül, amelyeket a terheléselosztó fog szolgálni. A fontos részletek a következők:

* + - * + *azurerm\_lb\_backend\_address\_pool\_address*: njevm01 és njevm02, amelyek az egyes virtuális gépeket azonosítják a terheléselosztó konfigurációjában.
        + *backend\_address\_pool\_id*: Mindkét cím az azurerm\_lb\_backend\_address\_pool.pool\_a.id backend címmedencéhez van hozzárendelve, ami a terheléselosztó egy csoportját jelenti, ahol a kérések eloszthatók.
        + *virtual\_network\_id*: Mindkét cím ugyanahhoz a azurerm\_virtual\_network.app\_network.id virtuális hálózathoz tartozik.
        + *ip\_address*: Az egyes virtuális gépek hálózati interfészének (azurerm\_network\_interface.app\_interface1 és app\_interface2) privát IP-címei, amelyeket a terheléselosztó használ.

Fontosságuk a terheléselosztásban

Ezek az erőforrások nélkülözhetetlenek egy hatékony terheléselosztó infrastruktúra létrehozásában, mivel lehetővé teszik, hogy a terheléselosztó dinamikusan eloszthassa a hálózati forgalmat az elérhető szerverek között. Az IP-címek hozzárendelése a backend címmedencéhez biztosítja, hogy a terheléselosztó pontosan tudja, melyik szerverekhez irányítsa a forgalmat. Ez hozzájárul:

* + - * + *Megbízhatósághoz*: A terhelés egyenletes elosztása javítja az alkalmazások és szolgáltatások rendelkezésre állását, mivel a hibás gépek kiesése esetén a terhelést át lehet irányítani az egészséges gépekre.
        + *Skálázhatósághoz*: Lehetővé teszi, hogy a rendszer könnyen bővíthető legyen több szerver hozzáadásával a címmedencéhez, így a növekvő forgalmat is képes kezelni.
        + *Teljesítmény* *optimalizálásához*: A forgalom intelligens elosztása javítja a válaszidőket és csökkenti a túlterhelést az egyes szervereken.

Ez a konfiguráció tehát képes alkalmazkodni a változó forgalmi igényekhez.

**azurerm\_lb\_probe és azurerm\_lb\_rule**

resource "azurerm\_lb\_probe" "probe\_a" {

loadbalancer\_id = azurerm\_lb.app\_balancer.id

name = "probe\_a"

port = 80

depends\_on = [

azurerm\_lb.app\_balancer

]

}

resource "azurerm\_lb\_rule" "rule\_a" {

loadbalancer\_id = azurerm\_lb.app\_balancer.id

name = "rule\_a"

protocol = "Tcp"

frontend\_port = 80

backend\_port = 80

frontend\_ip\_configuration\_name = "frontend-ip"

backend\_address\_pool\_ids = [ azurerm\_lb\_backend\_address\_pool.pool\_a.id ]

probe\_id = azurerm\_lb\_probe.probe\_a.id

depends\_on = [

azurerm\_lb.app\_balancer,

azurerm\_lb\_probe.probe\_a

]

}

Az utolsó két erőforrás, az azurerm\_lb\_probe és az azurerm\_lb\_rule, lényeges szerepet játszanak a terheléselosztók hatékony és megbízható működésében.

Erőforrás deklaráció és konfigurációi:

* + - * + *azurerm\_lb\_probe*: Ez egy egészségellenőrző mechanizmust definiál a terheléselosztó számára. Ez a szonda ellenőrzi a backend szerverek állapotát, hogy azok képesek-e fogadni a forgalmat. Ebben az esetben a szonda a 80-as porton ellenőrzi a szervereket, ami a HTTP alapértelmezett portja.

A probe nélkülözhetetlen az infrastruktúra megbízhatóságának biztosításában, mert automatikusan kizárja azokat a szervereket, amelyek nem válaszolnak vagy hibás állapotban vannak, ezzel megakadályozva a hibás válaszokat vagy a szolgáltatás kimaradását.

- *azurerm\_lb\_rule*: Egy forgalomirányítási szabályt definiál, ami meghatározza, hogy a beérkező kérések hogyan routolódnak a backend szerverekre. A TCP protokollt használja, és mind a frontend, mind a backend portokat a 80-as porton állítja be. Ez biztosítja, hogy a HTTP kérések megfelelően továbbításra kerüljenek.

- *backend\_address\_pool\_ids*: A rule az azurerm\_lb\_backend\_address\_pool.pool\_a.id címmedencét használja, ami tartalmazza a szolgáltatásra kész backend szervereket.

- *probe\_id:* Integrálja a korábban definiált szondát (azurerm\_lb\_probe.probe\_a.id), amely ellenőrzi, hogy a kéréseket csak az egészséges szerverek kapják.

- *depends\_on:* A szabály működése szintén függ a terheléselosztótól és a szondától, biztosítva a szinkronizált és hibamentes működést.

A rule definiálása elengedhetetlen a terheléselosztás finomhangolásában, mivel lehetővé teszi, hogy a terheléselosztó intelligensen oszthassa el a forgalmat a rendelkezésre álló szerverek között, figyelembe véve azok aktuális állapotát és kapacitását.

Ezzel a Terraform kód bemutatásának a végére értünk, ami a teljes Azure infrastruktúra kiépítését célozza. Minden erőforrás elkészült, amely szükséges egy kiterjedt és integrált környezet létrehozásához, beleértve a két Windows VM-et is, amelyek terheléselosztás alatt működnek. A kód tesztelés során garantálta ezeknek a virtuális gépeknek a zavartalan működését.

A projekt során számos hibával és kihívással kellett szembenéznem. Az első akadály az Azure és a Terraform közötti integráció pontos beállítása volt. További komplikációk merültek fel a storage blob konfigurációja során, ahol a helyes paraméterek meghatározása kulcsfontosságú volt. Az egyik legidőigényesebb feladat az extension script összeállítása volt, amely nélkülözhetetlen volt ahhoz, hogy a Terraform képes legyen importálni a sap-html-t a GitHub-ról és helyesen telepíteni azt a szerverre.

Külön kiemelném a loadbalancer konfigurációjának összeállítását, amely számos specifikus követelményt és lépést tartalmazott. Ennek a komponensnek a megtervezése és implementálása rendkívül bonyolult volt, figyelembe véve a szükséges biztonsági és teljesítménybeli paramétereket, amelyek garantálják a rendszer zavartalan és hatékony működését.

Mindezeket a kihívásokat sikerült leküzdenem a problémák alapos elemzésével és a troubleshooting technikák alkalmazásával. Ez a folyamat nemcsak, hogy mélyebb betekintést nyújtott az Azure platform működésébe, de jelentősen bővítette a Terraformmal kapcsolatos ismereteimet is.

Következő lépésként a Jenkins automatizálási platform bevonásával szeretném bemutatni, hogyan tudtam tovább finomítani a folyamatot. A Jenkins segítségével sikerült egy olyan rendszert létrehoznom, amely lehetővé teszi a teljes infrastruktúra kiépítését egyetlen kattintással. Ez a megoldás nemcsak, hogy jelentős mértékben csökkenti a kód implementációjának idejét, hanem növeli a folyamat megbízhatóságát és átláthatóságát is. A következő részben részletesen ismertetem az automatizálás lépéseit, a Jenkins konfigurációját, és azt a Pipeline script-et, amelyet asználtam a folyamat során.

# Jenkins

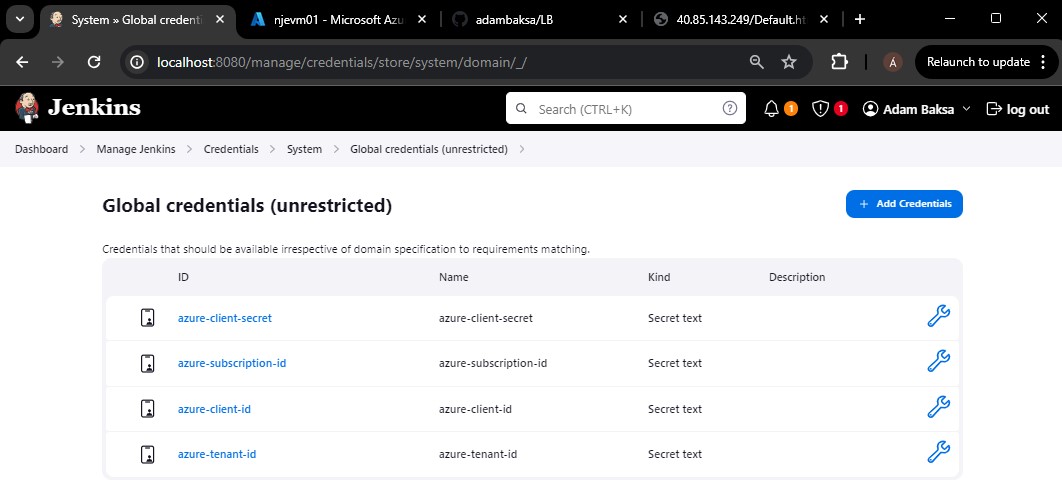
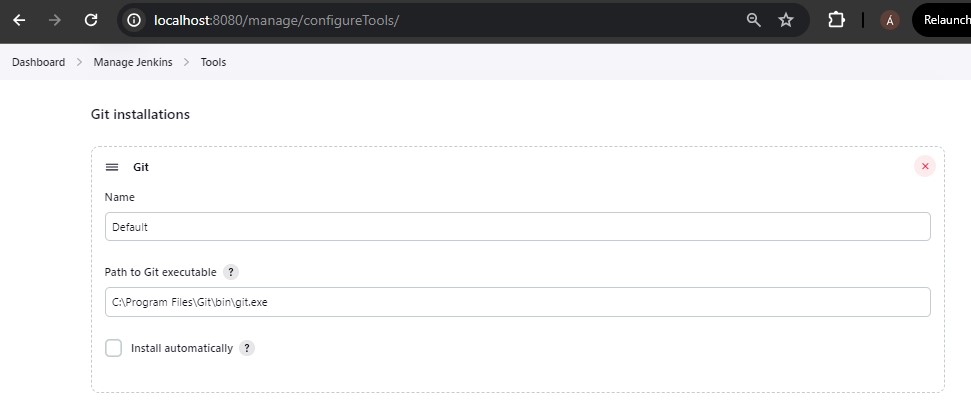
## Jenkins telepítése

1. <https://jenkins.io/download/> oldalról kiválasztjuk a Windows operációs rendszerhez készült telepítőt.
2. A telepítési varázsló utasításait követve kiválaszthatjuk az alapértelmezett telepítési helyet vagy a saját elérési utat adhatunk meg.
3. A telepítés befejeztével a Jenkins automatikusan elindul, és futtatja a szükséges konfigurációs folyamatokat.
4. Adminisztrációs hozzáférés konfigurálásához a böngésző megnyitása után a következő címet adjuk meg: <http://localhost:8080>. Itt a Jenkis kezdeti beállítási oldala jelenik meg.
5. A képernyőn található utasítások alapján a `C:\Program Files (x86)\Jenkins\secrets\initialAdminPassword` útvonalon található fájlban találjuk a jelszót az első belépéshez.
6. A kezdeti beállítások során válaszhatunk, hogy az alapértelmezett bővítményeket telepítjük, vagy kiválaszthatjuk az egyedi bővítményeket, amelyekre szükségünk van. Jelen esetben nekem telepíteni kellett a GitSCM és a terraform plugint.
   1. GitSCM plugin a Jenkins egyik leggyakrabban használt pluginja, amely lehetővé teszi a Git verziókezelő rendszer integrálását a Jenkins projektjeibe
   2. A Terraform plugin a Jenkins környezetbe integrálja a Terraformot. Támogatja a plan, apply és destroy műveleteket, amelyek lehetővé teszik az infrastruktúra automatizált telepítését, módosítását és lebontását.
7. A bővítmények telepítése után a rendszer felszólít, hogy hozzunk létre egy adminisztrátori felhasználót.
8. A folyamat végén a Jenkins átirányít a főoldalra, ahol megkezdhetjük a konfigurációk beállítását.

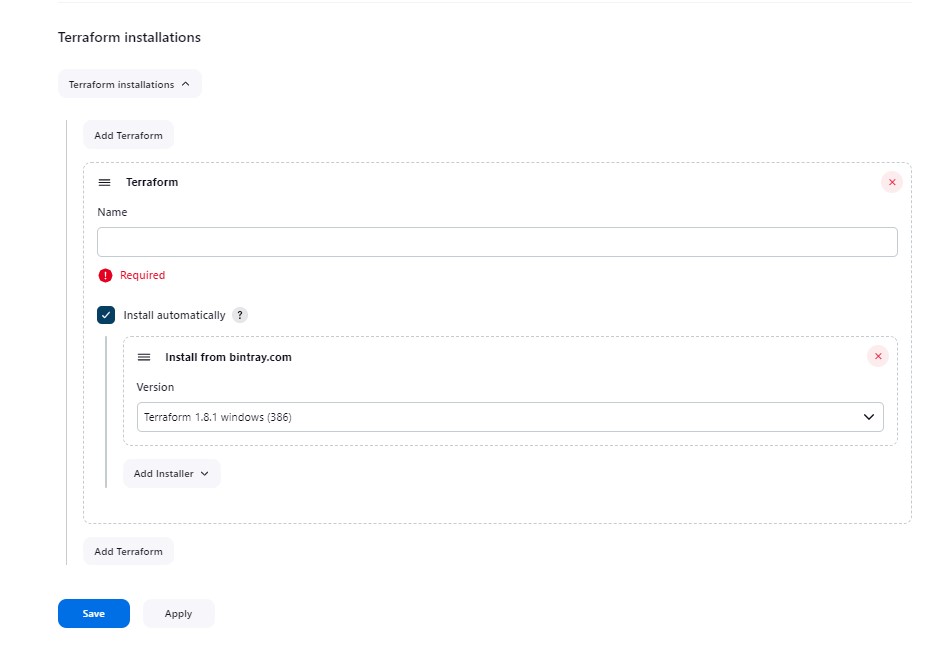
## Jenkins konfiguráció

A szakdolgozatomban bemutatásra kerülő projekt következő lépése a GitHub-ra történő feltöltés, amely magában foglalja az összes Terraform fájlt és a „scripts.zip” nevű archívumot. Ez az archívum tartalmazza azokat az HTML fájlokat, amelyek elengedhetetlenek a webalkalmazás megjelenítéséhez és működéséhez. Emellett létrehozásra kerül egy Jenkins pipeline, amelynek részletezése a későbbiekben történik meg, és ez is szintén a GitHub-ra kerül feltöltésre.

A konfiguráció további lépései közé tartozik az összes szükséges hitelesítési adat beállítása, amely a „providers.tf” fájlban is definiálva van. Ezek a hitelesítési adatok szükségesek ahhoz, hogy a Jenkins hozzáférhessen az Azure előfizetéshez. E hitelesítési adatok kezelése a Jenkins felületén, a „Manage Jenkins” menü „Credentials” almenüjében történik.

A Jenkinsben a „Manage Jenkins” menü „Tools” almenüje lehetővé teszi számos fejlesztési és build eszköz konfigurálását. Ez a funkcionalitás különösen hasznos, mert centralizáltan lehet kezelni az összes szükséges eszköz beállításait, amelyeket a Jenkins különböző job-jai és projektek igényelnek. Ennek értelmében a Git installation szekciónál megadtam a „Path to Git executable” résznél C:\Program Files\Git\bin\git.exe, ami a saját telepített GitSCM tool elérési útvonala.

Ugyanebben a menüpontban, a ’Terraform installations’ rész alatt adjuk meg a gépre telepített Terraform verzióját. Ennek pontos verzióját a parancssorban a „terraform –version” parancs kiadásával ellenőrizhetjük, és ezt követően az eredményül kapott verziószámot kell megadni a megfelelő mezőbe.

 Összefoglalva, a Jenkins konfigurációjának alapvető követelményei, amelyek lehetővé teszik a Jenkins számára a Git és az Azure szolgáltatásokkal való kommunikációt, a következő lépésekben foglalhatók össze: a szükséges pluginok telepítése, a hitelesítési adatok megadása és az eszközök megfelelő beállítása.

## Jenkins pipeline kód

pipeline {

agent any

parameters {

booleanParam(name: 'autoApprove', defaultValue: false, description: 'Automatically run apply after generating plan?')

choice(name: 'action', choices: ['apply', 'destroy'], description: 'Select the action to perform')

}

environment {

ARM\_SUBSCRIPTION\_ID = credentials('azure-subscription-id')

ARM\_CLIENT\_ID = credentials('azure-client-id')

ARM\_CLIENT\_SECRET = credentials('azure-client-secret')

ARM\_TENANT\_ID = credentials('azure-tenant-id')

}

Az „agent any” kifejezés azt jelenti, hogy a pipeline bármely rendelkezésre álló agent-en futtatható a Jenkins környezetben. Ez a beállítás biztosítja, hogy a pipeline ne legyen korlátozva egy konkrét gépre vagy környezetre.

A pipeline paramétereket definiál:

* + - * + *autoApprove:* Egy boolean típusú paraméter, amely dönti el, hogy az apply művelet automatikusan megtörténjen-e a plan generálása után.
        + *action:* Egy választási lehetőség, ahol a felhasználó kiválaszthatja az apply vagy destroy műveletet. Ez a paraméter határozza meg, hogy a Terraform milyen típusú műveletet hajt végre.

Az „Environment varables” az Azure hitelesítéséhez szükségesek, amelyek a Terraform által kezelt erőforrásokhoz való hozzáféréshez kellenek:

- ARM\_SUBSCRIPTION\_ID

- ARM\_CLIENT\_ID

- ARM\_CLIENT\_SECRET

- ARM\_TENANT\_ID

Ezeket a hitelesítési adatokat a Jenkins Credential Management rendszeréből tölti be a pipeline, amiket a Jenkins konfiguráció részben említettem.

stages {

stage('Checkout') {

steps {

dir('../LB') {

git branch: 'main', url: 'https://github.com/adambaksa/LB.git'

}

}

}

stage('Terraform init') {

steps {

dir('../LB') {

script {

bat 'terraform init'

}

}

}

}

A pipeline stages több szakaszból áll, amelyek különböző Terraform műveleteket hajtanak végre:

* + - * + *Checkout:* A szükséges Terraform konfigurációt tartalmazó Git repository klonozása. A dir('../LB') kifejezés azt jelzi, hogy a műveletek a ../LB mappában hajtódnak végre.
        + *Terraform init:* Inicializálja a Terraform munkakönyvtárat, ami előkészíti a backendet és letölti a szükséges provider pluginokat.

stage('Plan') {

steps {

dir('../LB') {

script {

bat 'terraform plan -out main.tfplan'

bat 'terraform show -no-color main.tfplan > tfplan.txt'

}

}

}

}

* + - * + *Plan:* Generál egy Terraform tervet (main.tfplan), amelyet beír egy tfplan.txt fájlba. Ez a lépés segít megjeleníteni a végrehajtandó változtatásokat.

stage('Apply / Destroy') {

steps {

dir('../LB') {

script {

// Here we ensure we're using Groovy string interpolation to construct the command

def terraformCommand = "terraform ${params.action} -input=false main.tfplan"

if (params.action == 'apply') {

if (!params.autoApprove) {

def plan = readFile 'tfplan.txt'

input message: "Do you want to apply the plan?",

parameters: [text(name: 'Plan', description: 'Please review the plan', defaultValue: plan)]

}

// Using Groovy string interpolation to pass the correct command to bat

bat terraformCommand

} else if (params.action == 'destroy') {

// Same approach here for the destroy action

bat "terraform ${params.action} --auto-approve"

} else {

error "Invalid action selected. Please choose either 'apply' or 'destroy'."

}

}

}

}

}

}

}

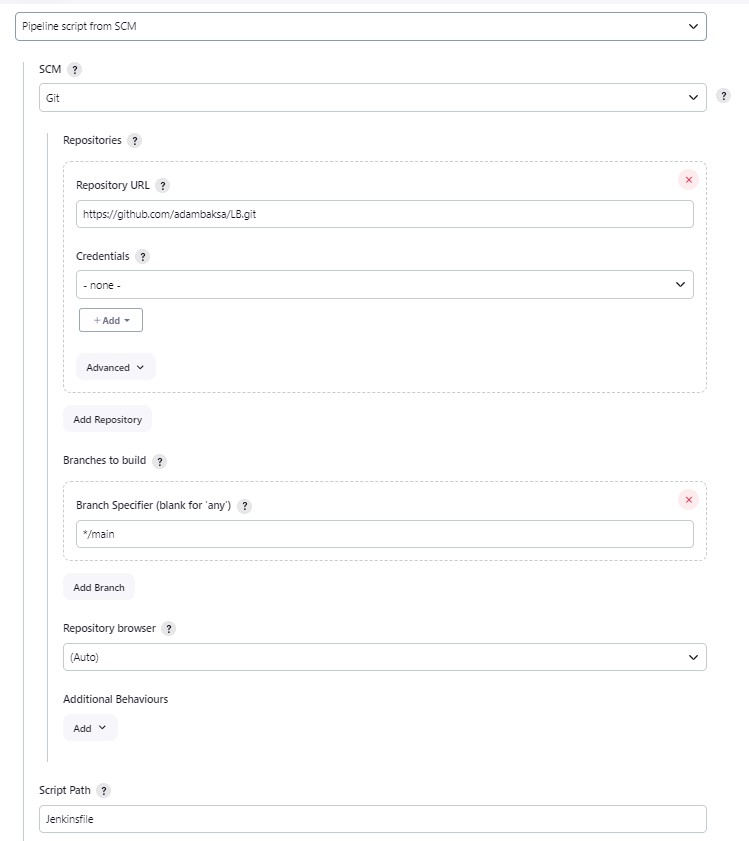
*Apply/Destroy:* Ez a szakasz végzi el az apply vagy a destroy műveleteket. A params.action változó értéke alapján dönti el, hogy melyik parancsot hajtsa végre:

- apply: Ha az autoApprove false (thick box üres), a rendszer megerősítést kér a terraform plan elkészítése után, hogy biztosan futtatni szeretnénk-e. Ha az autoApprove true (thick box kijelölve), a rendszer automatikusan végig futtatja a job-ot.

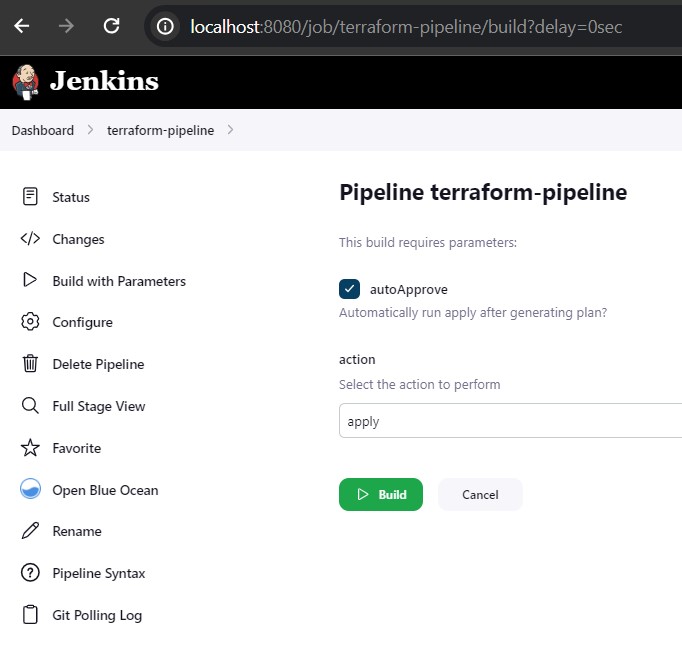
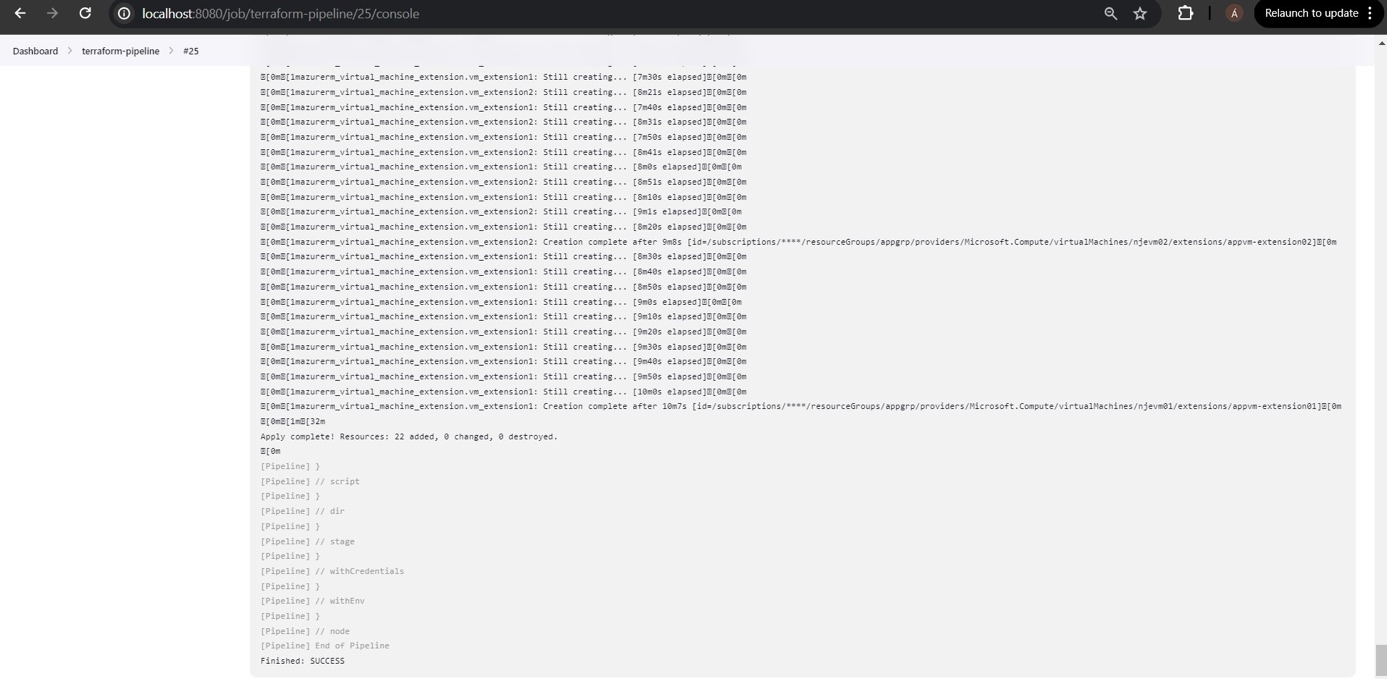
- destroy: Automatikus jóváhagyással végzi el a megsemmisítést, megkönnyítve a használt erőforrások eltávolítását.

## Jenkins pipeline létrehozása

Pipeline létrehozásának lépései a következők:

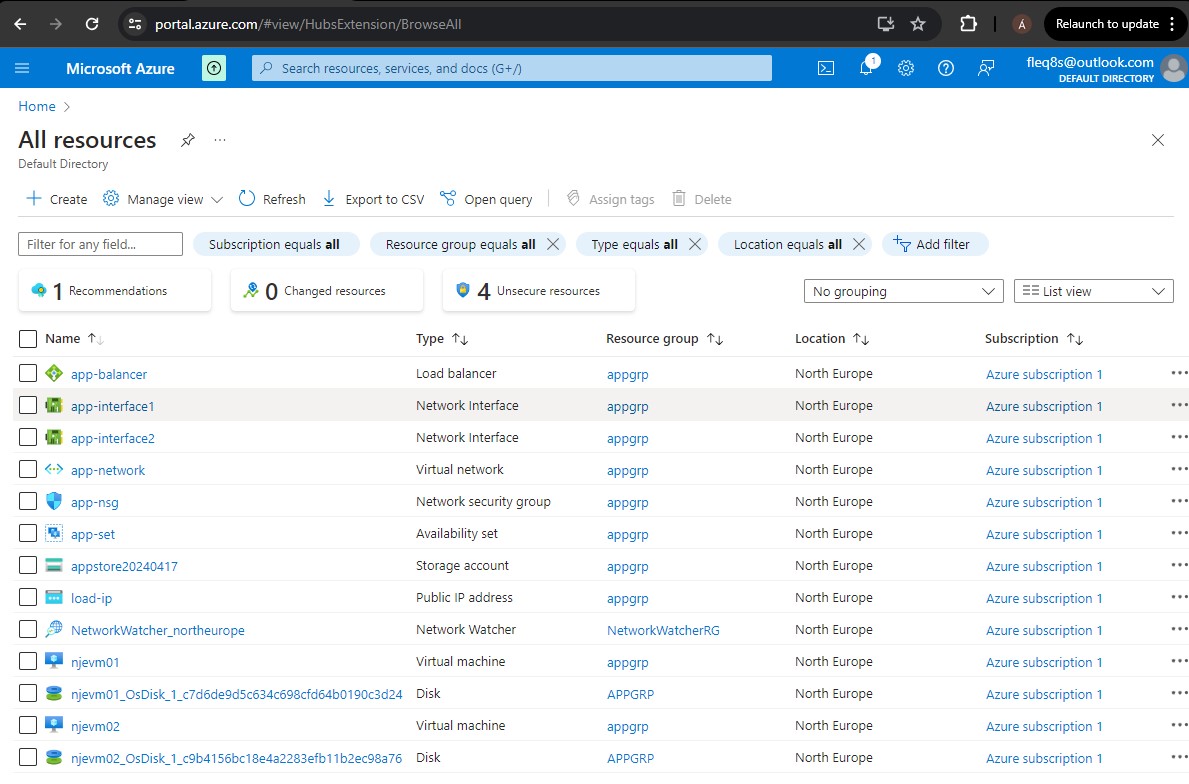
1. **New item**
2. Adjunk egy nevet a item-nek (*terraform-pipeline*), majd válasszuk ki a „Pipeline”-t.
3. Build trigger alatt jelöljük ki a **Poll SCM**-et. Ez a funkció lehetővé teszi a Jenkins számára, hogy időszakosan ellenőrizze a változásokat a forráskód-verziókezelő rendszerben, jelen esetben a Git-en.
4. A Pipeline - Definition alatt, válasszuk a „**Pipeline script from SCM**”, majd az SCM alatt, hogy **GIT**. Ekkor megjelentik a „Repository URL”, ahova illesszük be a saját git repository URL-ünket.
5. Specifikusan meg kell adnunk a Branch-et. A mi esetünkben ez „**\*/main**” lesz.
6. Továbbá meg kell adni a „Script path”-t, ami a „J**enkinsfile**”.

## Jenkins pipeline futtatása

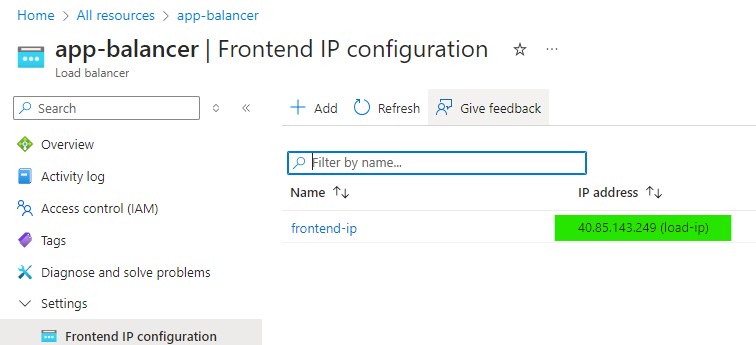
1. Válasszuk ki létrehozott pipeline-t.
2. A bal oldalon található menüsorban kattintsunk a „Build with Parameters” opcióra.
3. Válasszuk ki, hogy autoApprove, így megerősítés nélkül végig fut a job.
4. Az action résznél hagyjuk „apply”-on az az infrastruktúra kiépéítéséhez.
5. Build
6. Kattinsunk az aktuális Job-ra.
7. A bal oldalon található menüsoron található a „Consol output”-ot, ahol láthatjuk a Jenkins build vagy job futásának részletes naplózását. Ez a kimenet tartalmazza az összes információt, műveletet és eseményt, amely a job futása során történt. A console output hasznos eszköz a hibák diagnosztizálására, a build folyamatok nyomon követésére és az automatizált scriptek végrehajtásának ellenőrzésére. (Spacelift, 2024)

# A felépített infrastrukúra működése

## Azure portal



Az „*app-balancer – Settings – Frontend IP configuration”* alatt érjük el a publikus IP címet, amin keresztül bárki eléri a webapp-ot.



Az Azure Load Balancer segítségével a frontend IP-konfiguráció több szerver között osztja szét a bejövő forgalmat. Ez biztosítja, hogy ha egy szerver hiba miatt elérhetetlenné válik, a terhelés átirányítható más, még működő szerverekre.

A frontend IP-címek használata lehetővé teszi, hogy a belső hálózati erőforrások elrejtőzzenek a nyilvános internet elől, miközben továbbra is képesek fogadni a külső kéréseket. Ez csökkenti a biztonsági kockázatokat.

## Webapp

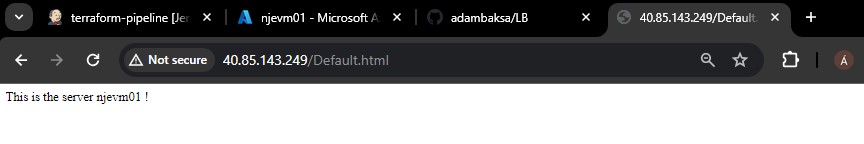
A <http://40.85.143.249/Default.html> megmutatja, hogy éppen melyik load-balancer szerver fut a webalkalmazás mögött. Ezért a scripts.zip archívumban található *IIS\_config.ps1* file felel, a webapp-hoz szükséges applikációk telepítése mellett.

Import-Module ServerManager

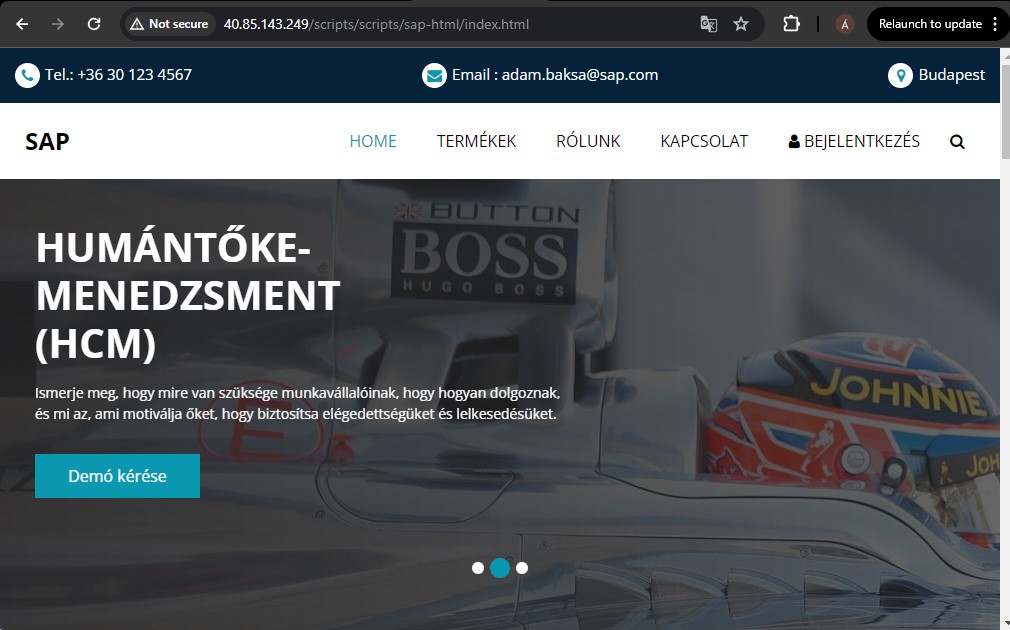
Add-WindowsFeature Web-Server -IncludeAllSubFeature

Add-WindowsFeature Web-Asp-Net45

Add-WindowsFeature NET-Framework-Features

Set-Content -Path "C:\inetpub\wwwroot\Default.html" -Value "This is the server $($env:computername) !"

Az <http://40.85.143.249/scripts/scripts/sap-html/index.html> URL címen érhető el a webalkalmazás, amely a SAP hivatalos weboldalának replikájaként hoztam létre.



# Következtetések, javaslatok

## Következtetés

Az Infrastructure as Code (IaC), Terraform és Jenkins ismerete a DevOps munkakörben azért különösen hasznos, mert ezek az eszközök és módszerek több szempontból is alapvetőek a hatékony és modern IT infrastruktúra kezelésében. Az IaC lehetővé teszi az infrastruktúra teljes körű automatizálását és hatékony kezelését, amely jelentősen csökkenti a manuális beavatkozásokból eredő hibák számát, miközben gyorsítja a telepítési és konfigurációs folyamatokat. Az automatizált infrastruktúra kezelése révén a fejlesztők és az üzemeltetők konzisztensen és ismételhető módon hozhatnak létre különböző környezeteket, mivel a konfigurációs fájlok nemcsak dokumentálják, hanem nyomon is követik az infrastruktúra változásait.

A Terraform használata az erőforrások optimalizálása és a költségek csökkentése mellett rugalmasan és gyorsan skálázható infrastruktúrát biztosít, amely képes alkalmazkodni a változó üzleti igényekhez. A Terraform konfig fájlok segítségével az erőforrások hatékonyan kezelhetők, és a felesleges költségek minimalizálhatók, mivel az infrastruktúra pontosan az igényeknek megfelelően méretezhető. Az IaC koncepció lehetővé teszi, hogy az infrastruktúra előzetesen tesztelhető és validálható legyen, ami csökkenti a termelési környezetben előforduló hibák kockázatát.

A Jenkins által nyújtott automatizált CI/CD folyamatok biztosítják a kód gyors és biztonságos bevezetését, valamint a gyakori frissítések hatékony kezelését. A Jenkins pipeline-ok automatizálják az összes fejlesztési, tesztelési és telepítési folyamatot, növelve a fejlesztési ciklus hatékonyságát és csökkentve az átfutási időt. Ez a folyamatos integráció és szállítás lehetővé teszi, hogy a fejlesztők gyorsabban reagáljanak a hibákra és a változó igényekre, minimalizálva a leállási időt és biztosítva a magas rendelkezésre állást.

Ezen eszközök és módszerek közös nyelvet és eszköztárat biztosítanak a fejlesztők és az üzemeltetők számára, elősegítve a hatékony csapatmunkát és az integrációt más DevOps rendszerekkel. Az IaC, Terraform és Jenkins használata elősegíti a DevOps kultúra kialakítását és fenntartását, amely a fejlesztés és az üzemeltetés közötti szakadék áthidalására törekszik. Végső soron növeli az egész szervezet agilitását és versenyképességét, mivel lehetővé teszi az infrastruktúra gyors és rugalmas módosítását, valamint a fejlesztési folyamatok hatékonyabbá tételét.

## Javaslatok

A jövőbeni fejlesztések között kiemelt fontosságú a meglévő Terraform konfiguráció modularizálása, amely révén növelhető az infrastruktúra kódjának újrafelhasználhatósága és karbantarthatósága, továbbá lehetővé válik az egyes komponensek független fejlesztése és tesztelése. A modulok létrehozása során célszerű különálló fájlokba szervezni az egyes erőforrásokat, mint például a virtuális hálózatot, az alhálózatot, a hálózati interfészeket, a virtuális gépeket, valamint a load balancert és a hozzá tartozó konfigurációkat, így ezek könnyebben kezelhetők és áttekinthetők lesznek.

Egy következő fejlesztési irány lehet a Terraform state fájl tárolásának és kezelésének javítása, például az Azure Storage használatával, amely nemcsak a state fájl biztonságos tárolását teszi lehetővé, hanem támogatja a csapatmunkát és a párhuzamos módosítások kezelését is. Az automatizálás további fokozása érdekében integrálható a Terraform konfig fájlok validálása és tesztelése a Jenkins pipeline-okba, ami biztosítja, hogy minden változtatás átfogó tesztelésen menjen keresztül, mielőtt alkalmazásra kerülne a termelési környezetben.

Emellett érdemes megfontolni a fejlett monitorozási és naplózási megoldások bevezetését, például az Azure Monitor és az Application Insights használatával, amelyek valós idejű adatokat és elemzéseket nyújtanak az infrastruktúra teljesítményéről és állapotáról, lehetővé téve a gyors reagálást az esetleges problémákra. A biztonság növelése érdekében bevezethetők további hálózatbiztonsági intézkedések, mint például a webalkalmazás tűzfal (WAF) és az Azure Security Center használata, amelyek elősegítik a kiberfenyegetések elleni védelmet.

Végül, a jövőbeni fejlesztések közé tartozhat a CI/CD folyamatainak továbbfejlesztése is, például a Jenkins pipeline-ok tovább bővítéséve. A jelenlegi Jenkins pipeline konfiguráció, amely az Azure Terraform kód futtatásáért felel, kiváló alapot biztosít az infrastruktúra felépítéséhez. Azonban a pipeline további fejlesztésével növelhető a rugalmasság és a testreszabhatóság, amely lehetővé teszi a felhasználók számára, hogy a Jenkins felületen keresztül paramétereket adhassanak meg, például a disk méretét és a szerver nevét. Ezt két lépésben lehetne kivitelezni; A Jenkins pipeline script módosítása úgy, hogy a felhasználók paramétereket adhassanak meg a futtatás előtt. Ez magában foglalja a „disk\_size” és a „server\_name” paraméterek bevezetését. Továbbá, a Terraform konfig fájlok módosítása úgy, hogy a Jenkins-ből kapott paramétereket használják a megfelelő erőforrások konfigurálására.

# Összefoglalás

Szakdolgozatomban részletesen bemutattam az Infrastructure as Code (IaC) fogalmát, a Terraform, az Azure DevOps és a Jenkins eszközöket. Az IaC lehetővé teszi az infrastruktúra automatizált kezelését és konfigurálását kód segítségével, ezzel növelve a hatékonyságot és csökkentve a hibalehetőségeket. Az IaC alkalmazása révén a rendszerek átláthatósága és megbízhatósága is jelentősen javul, mivel a kód alapú megközelítés lehetővé teszi a verziókövetést és a változtatások egyszerű nyomon követését. A Terraform egy nyílt forráskódú eszköz, amely lehetővé teszi az infrastruktúra deklaratív meghatározását és kezelését. A deklaratív megközelítés azt jelenti, hogy a kívánt állapotot írjuk le, amit a rendszernek el kell érnie, a Terraform pedig automatikusan gondoskodik az erőforrások létrehozásáról és beállításáról. Ez a módszer csökkenti az emberi hibák esélyét és növeli a folyamatok konzisztenciáját.

A Jenkins egy másik rendkívül hasznos eszköz a DevOps folyamatokban. Automatizálja a szoftver build, tesztelési és telepítési folyamatokat, támogatva a folyamatos integrációt (CI) és folyamatos szállítást (CD). A Jenkins segítségével a fejlesztők könnyedén beállíthatnak automatizált build folyamatokat, amelyek minden kódváltoztatás után automatikusan lefutnak. Ez lehetővé teszi a gyakori és megbízható kiadásokat, valamint a hibák korai felismerését és javítását.

A projektfeladat során két darab Windows alapú load balancer szervert hoztam létre, amelyeken egy webalkalmazás (webapp) fut. Az infrastruktúrát Azure környezetben Terraformmal építettem fel. A Terraform kódot egy Git repository-ban tárolom, és a Jenkins segítségével automatikusan futtatom, amely létrehozza és konfigurálja az összes szükséges erőforrást Azure-ben.

A projekt megvalósítása során a következő lépéseket követtem:

1. Környezet előkészítése: Létrehoztam egy Azure Active Directory (AD) alkalmazást és egy szolgáltatói fiókot, amelyet a Terraform használ az Azure erőforrások kezelésére.

2. Erőforráscsoport létrehozása: Terraformmal létrehoztam Azure-ban egy erőforráscsoportot, amely tartalmazza az összes szükséges erőforrást.

3. Virtuális hálózat és biztonsági csoportok létrehozása: A Terraformmal létrehoztam egy virtuális hálózatot és a szükséges biztonsági csoportokat. A virtuális hálózat biztosítja az erőforrások közötti kommunikációt, míg a biztonsági csoportok szabályozzák a hálózati forgalmat, növelve ezzel a rendszer biztonságát.

4. Load balancer szerverek létrehozása: Két load balancer szervert hoztam létre, amelyek egy webalkalmazás forgalmát kezelik. A load balancerek biztosítják a terhelés egyenletes elosztását a szerverek között, javítva ezzel az alkalmazás rendelkezésre állását és teljesítményét.

5. Jenkins Pipeline konfigurálása: A Jenkins segítségével beállítottam egy pipeline-t, amely figyeli a Git repository-ban bekövetkező változásokat és automatikusan lefuttatja a Terraform parancsokat, létrehozva és konfigurálva az infrastruktúrát. Ez a pipeline biztosítja, hogy a kódváltoztatások gyorsan és megbízhatóan érvényesüljenek az infrastruktúrában.

Ezek az ismeretek és lépések kulcsfontosságúak egy DevOps munkakörben, mivel az automatizált infrastruktúra kezelés és a folyamatos integráció/szállítás (CI/CD) jelentősen növeli a fejlesztési és üzemeltetési folyamatok hatékonyságát. Az IaC alkalmazása lehetővé teszi az infrastruktúra gyors és megbízható kiépítését, minimalizálva a manuális beavatkozásokból eredő hibákat. A Terraform és a Jenkins használata pedig biztosítja, hogy az infrastruktúra mindig a legfrissebb állapotban legyen, és a változtatások gyorsan és hatékonyan végrehajthatók legyenek.

A szakdolgozatomban bemutatott projekt nemcsak technikai értelemben jelentős, hanem gyakorlati szempontból is, mivel az automatizált folyamatok és az integrált eszközök alkalmazása révén a vállalatok hatékonyabban és gyorsabban tudnak reagálni az üzleti igényekre. A Jenkins pipeline automatizálása lehetővé teszi a fejlesztők számára, hogy több időt fordítsanak az új funkciók fejlesztésére, miközben biztosítják, hogy az infrastruktúra megbízható és naprakész maradjon.

Összességében a szakdolgozatom bemutatta, hogy az IaC, a Terraform, az Azure és a Jenkins hogyan segíthetnek a modern szoftverfejlesztési és üzemeltetési gyakorlatok megvalósításában. Az automatizálás és az integrált eszközök alkalmazása révén a vállalatok nemcsak, hogy hatékonyabban tudják kezelni az infrastruktúrájukat, hanem gyorsabban és megbízhatóbban tudják kiszolgálni ügyfeleiket, növelve ezzel a versenyképességüket és a piaci pozíciójukat.

# Summary

In my thesis, I provided a detailed explanation of the concept of Infrastructure as Code (IaC), as well as the tools Terraform, Az ure DevOps, and Jenkins. IaC enables the automated management and configuration of infrastructure using code, thereby increasing efficiency and reducing the potential for errors. Implementing IaC significantly enhances the transparency and reliability of systems, as the code-based approach allows for version control and easy tracking of changes. Terraform is an open-source tool that facilitates the declarative definition and management of infrastructure. The declarative approach involves describing the desired state that the system should achieve, and Terraform automatically handles the creation and configuration of resources. This method reduces the likelihood of human errors and increases process consistency.

Jenkins is another highly valuable tool in DevOps processes. It automates software build, testing, and deployment processes, supporting continuous integration (CI) and continuous delivery (CD). Jenkins allows developers to easily set up automated build processes that run every time there is a change in the code. This enables frequent and reliable releases, as well as the early detection and resolution of errors.

For my project task, I created two Windows-based load balancer servers running a web application (webapp). The infrastructure was built using Terraform in an Azure environment. The Terraform code is stored in a Git repository, and Jenkins is used to automatically execute it, creating and configuring all the necessary resources in Azure. This approach not only speeds up the creation of infrastructure but also ensures that the environment is always up to date, and changes can be implemented quickly and effectively.

The steps I followed in implementing the project were as follows:

1. Preparing the environment: I created an Azure Active Directory (AD) application and a service principal that Terraform uses to manage Azure resources. This step is critical as the AD application and service principal provide authentication and authorization for accessing Azure resources.
2. Creating resource group: Using Terraform, I created a resource group in Azure that contains all the necessary resources. Resource groups enable logical grouping of resources, making it easier to manage them and track costs.
3. Setting up a Virtual Network and Security Groups: With Terraform, I created a virtual network and the necessary security groups. The virtual network ensures communication between resources, while the security groups regulate network traffic, thereby enhancing system security.
4. Creating Load Balancer Servers: I set up two load balancer servers to handle the traffic for the web application. Load balancers ensure even distribution of traffic among servers, improving the availability and performance of the application.
5. Configuring a Jenkins Pipeline: Using Jenkins, I configured a pipeline that monitors changes in the Git repository and automatically executes the Terraform commands to create and configure the infrastructure. This pipeline ensures that code changes are quickly and reliably reflected in the infrastructure.

These skills and steps are crucial for a DevOps role, as automated infrastructure management and continuous integration/delivery (CI/CD) significantly enhance the efficiency of development and operations processes. Implementing IaC allows for the quick and reliable setup of infrastructure, minimizing errors from manual interventions. Using Terraform and Jenkins ensures that the infrastructure is always up to date, and changes can be executed swiftly and efficiently. Integrating Azure DevOps improves team collaboration and process transparency, which is essential for successful and scalable software development and operations practices.

The project presented in my thesis is not only technically significant but also practically valuable, as the use of automated processes and integrated tools allows companies to respond more efficiently and quickly to business needs. Automating the Jenkins pipeline enables developers to spend more time developing new features while ensuring that the infrastructure remains reliable and up to date.

Overall, my thesis demonstrated how IaC, Terraform, Azure DevOps, and Jenkins can aid in implementing modern software development and operations practices. By adopting automation and integrated tools, companies can manage their infrastructure more efficiently, deliver services to their customers more quickly and reliably, and enhance their competitiveness and market position.