Bevezető

Üdvözöllek a *Bevezetés a szoftverarchitektúrákba* képzésen!

A képzés elméleti és gyakorlati részekből áll. A gyakorlati részeket te is próbáld meg megcsinálni a videó alapján!

Ehhez a következő szoftverek feltelepítése szükséges:

- Chrome böngésző
- Docker Desktop
- DBeaver
- Postman
- SoapUI

Ezeket nem szükséges előre feltelepíteni, hanem elegendő az adott leckénél!

A tanfolyamhoz tartozó GitHub repository elérhető a következő címen:

https://github.com/Training360/architecture-public/

Architektúra fogalma

Ha az architektúra fogalmát szeretnénk definiálni, nehéz helyzetben leszünk, hiszen nincs rá egzakt definíció, sokan sokféleképpen értelmezik és időnként mást és mást értenek alatta. Amivel mégis legjobban közelíthetünk a fogalom megértéséhez, aszerint a szoftverarchitektúra nem más, mint egy bonyolult szoftverrendszer magas szintű struktúrájának a leírása.

Az biztos, hogy az architektúra akkor kezdett szerepet kapni, amikor megjelentek az elosztott rendszerek. Az elosztott rendszerek olyan szoftver rendszerek, amelyek nem egy számítógépre vannak telepítve, hanem több számítógépen vannak elosztva, melyek között hálózat van. Az elosztás ténye magával hozta azt az igényt, hogy pontos információ álljon rendelkezésre az egyes számítógépeken elhelyezett komponensekről, és az ezek közti kapcsolatokról. Az architektúra tehát egy magas szintű struktúra, amely nem foglalkozik a részletekkel. Elvárás az architektúra rögzítése is, melyre a diagramok használata a bevett és elterjedt módszer. Fontos a többesszám használata, mert adott szoftver architektúráját több különböző szempontból is lehet vizsgálni. Az architektúra képezi a szoftver fejlesztésében résztvevők (fejlesztők, üzleti elemzők, tesztelők, üzemeltetők stb.) közötti kommunikáció alapját, azáltal, hogy az architektúra pontos képet ad mindenki számára a szoftvert alkotó komponensekről. Az architektúra kialakítása a szoftver fejlesztésével együtt zajló folyamat, ahogy a szoftver bővül, fejlődik, úgy az architektúra változhat. Kevlin Henney mondta a következőt:

> Mikor például az architektúrákról beszélünk, erről hajlamosak vagyunk megfeledkezni: ahogy az épületek szerkezete sem csak azért van, hogy az egy helyben álljon, hanem hogy emberek éljenek és dolgozzanak bennük, úgy a

> szoftverarchitektúrák sem csak a technikai döntések halmazai, hanem emberek együttműködésének színhelyei.

Ez jól szemlélteti az architektúra összetett, és a szoftverek világában betöltött fontos szerepét.

A szoftverekkel szemben támasztott követelményeket két nagyobb halmazra oszthatjuk. Funkcionális (milyen funkciókat lásson el, azaz mit csinál a rendszer) és a nem funkcionális követelmények (milyen formában, azaz hogyan csinálja a rendszer). Az architektúra által kiszolgált igények tipikusan a nem funkcionális követelmények. Nézzünk a nem funkcionális követelményekre néhány példát:

- Futtató környezet (Linux vagy Windows operációs rendszer).
- Teljesítmény igények (válaszidő, kiszolgált felhasználók száma).
- Transzparens üzemeltethetőség (látható a rendszer belső állapota).
- Magas rendelkezésre állás és hibatűrés (folyamatos működés, minimális leállás).
- Skálázhatóság és méretezhetőség (növekvő felhasználószám esetén is elérhetővé tenni az alkalmazás módosítása nélkül).

Az architektúrának üzemeltetési feladatok szempontjából is fontos szerepe van. Az üzemeltetőnek a szoftverrendszert fel kell telepíteni az adott hardverre. Ha újabb verzió jön ki, akkor frissíteni kell. Állandó monitorozással megfigyelni, ellenőrizni. (Egy magasabb szintet képvisel, ha a rendszer már maga képes automatikus üzeneteket küldeni a felmerült hibákról.) Meghibásodás esetén sürgősen elhárítani a hibát. Folyamatos adatmentéssel biztosítani az adatok meglétét, az elveszett adatokat mentésből helyreállítani, visszatölteni.

Manapság már nemcsak fizikai számítógépekről beszélünk, hanem virtuális számítógépekről is. Ez azt jelenti, hogy egyetlen fizikai számítógépen akár több szoftveres, úgynevezett virtuális számítógép is futhat. A virtuális számítógépek használata azért hasznos, mert a szoftvereket továbbra is ugyanúgy izolálva, egymástól elválasztva tudjuk futtatni, mint több számítógép esetén, azonban a futtatásukhoz elegendő egy számítógép is, így a hardveres erőforrásokat hatékonyabban tudjuk kihasználni.

A komponensek egymástól való elkülönítése, elosztása meghibásodások esetén válik kulcsfontosságúvá, hiszen egy meghibásodott komponens nem okozza a további komponensek meghibásodását. A különböző komponensek üzenetekkel kommunikálnak egymással. Az üzenetek felépítése standardokat követ. Többnyire van egy feladó és egy cél komponens, egy fejléc (információk az üzenet típusáról, melyik rendszer küldi melyik rendszernek stb.), illetve egy törzs (a konkrét adat).

A hálózat alappillére az elosztott rendszereknek, viszont a hálózatokról elég sok tévhit él az emberek fejében. A hálózat megbízható? Nem, a hálózatban nem lehet teljes mértékben megbízni. Előfordulhatnak kisebb-nagyobb meghibásodások, melyek a hálózat belassulását vagy akár a megszakadását is okozhatják. Nincsen késleltetés? Sajnos van, például kontinenseken átnyúló számítógépes hálózatoknál konkrét fizikai határokba ütközünk. Néhány milliszekundum mindenképpen szükséges, hogy az adatátvitel teljesüljön. Végtelen a sávszélesség? Nem, hiszen nagy adat mennyiség esetén a teljes hálózat bedugulhat, és ez a hálózat lassulását okozhatja. Biztonságos? Nem. Az adat olyan csomópontokon is keresztül

halad, mely csomópontokért nem mi vagyunk felelősek, nem mi üzemeltetjük, nem a mi birtokunkban van, így a biztonságosságát sem tudjuk megítélni. Ilyen esetekben célszerű titkosítást alkalmazni. A hálózat topológiája nem változik? Dehogynem! A hálózaton belül a számítógépek száma, azok címei, illetve a kapcsolódás módjai folyamatosan változhatnak. Homogén? Nem, rendkívül változatosak. A legkisebb eszköztől (pl. óra, szenzor, kártya méretű számítógép), akár egy szuperszámítógépig minden rá van manapság csatlakoztatva egy hálózatra. Az eltérő rendszerek különböző kapacitással és erőforrásokkal rendelkeznek, más és más sávszélességgel kapcsolódnak egymáshoz. Egy adminisztrátor van, azaz egy valaki felelős azért, hogy a hálózat működjön? Nem, hiszen már egy kisebb cégnél is több adminisztrátorra, rendszergazdára lehet szükség. Az átvitel ingyen van? Nem, mert a hálózatokat folyamatosan karban kell tartani, üzemeltetni, amelynek természetesen költsége is van.

Az elosztott rendszerek fő előnye, hogy olyan komponensekre van bontva, melyeknek jól meghatározott feladatuk van, így azokat olyan számítógépekre lehet telepíteni, melyek az adott feladat elvégzésére hatékonyabbak (nagy számítási igény esetén több bennük a processzor, vagy sok adat tárolása esetén nagy tárterülettettel rendelkeznek). Amennyiben egy komponensnek további kapacitásra van szüksége, elegendő csak az adott komponenst futtató számítógép kapacitását növelni. Ezt horizontális (ugyanarra a feladatra több számítógépet használnak) vagy vertikális (egy számítógépbe több memóriát, processzort, háttértárat tesznek) skálázhatósággal szokták megoldani. Ezzel a megoldással valósítják meg tehát az elosztott rendszerek a nem funkcionális követelményeket. Az elosztott rendszereknek természetesen az előnyök mellett vannak hátrányai is. Mivel több komponensből állnak, ezért ezek a rendszerek bonyolultabbak. Nehezen átláthatók, nehezebben érthetők, nehezebben fejleszthetőek, az üzemeltethetősége és a tesztelhetősége is jóval bonyolultabb lehet.

Az elosztott rendszereknél kiemelt a biztonság kérdése, melyhez elengedhetetlen az autentikáció és authorizáció fogalmak megismerése. Autentikáció, azaz magyarul bejelentkezés. Én állítok magamról valamit, hogy ki vagyok, és ezt kell bizonyítanom. Klasszikus eset a felhasználónév és jelszó megadása, de vannak bonyolultabb formái is, mint pl.: a belépőkártya, ujjlenyomatos belépés vagy a két faktoros azonosítás (jelszó + sms). Authorizáció, azaz magyarul jogosultság kezelés. A számítógépes rendszer ellenőrzi, hogy milyen funkciókat vehetek igénybe. Például termék törlése a webshop kínálatból, vagy kizárólag a termék vásárlás. A felhasználókhoz az egyes jogosultságok nem önmagukban, hanem szerepkör alapján vannak hozzárendelve. Ilyenek lehetnek például, mint termékmenedzser, vásárló. Egy természetes személy akár több szerepkörrel is rendelkezhet. Az egyszerűsítés érdekében célszerű a felhasználókat csoportokban kezelni, így könnyebb az egyes adminisztrációs feladatok elvégzése.

Standalone konzolos alkalmazás

Még ma is használunk standalone konzolos alkalmazásokat, melyek története régre nyúlik vissza. Egy számítógépre feltelepített és kizárólag ezen a gépen futó alkalmazást jelöl a standalone jelző. Hozzáférést tekintve egyszerre csak egy felhasználót szolgál ki.

Az alkalmazás használatba vételéhez az első lépés a számítógépre történő másolás vagy telepítés volt. Mivel az egyszerű szoftverek is komponensekből állnak, melyek összeköttetésben vannak egymással, ezért telepítő készletbe csomagoljuk őket, így amikor feltelepítjük az alkalmazást, akkor a telepítő ezeket kicsomagolja, és a komponenseket a telepítő készletbe elhelyezett utasítások (telepítő program) szerint telepíti.

A konzolos jelző azt jelenti, hogy ezek az alkalmazások konzolból, azaz parancssorból vezérelhetőek. Azaz tehát karakteres (betűk, számok és speciális karakterek bevitelére szolgáló), és nem grafikus felületük van.

Ennek elterjedt neve a Command Line Interface (CLI) felület, azaz a parancssoros interfész. A felhasználó karaktereket visz be, ezzel parancsokat ad a számítógépnek és a számítógép erre úgyszintén karakterek megjelenítésével válaszol. Itt még nem használunk egeret.

Ehhez kapcsolódó fogalom még a shell. Ez gyakorlatilag egy olyan szoftver, amely várja a felhasználótól a parancsokat, ezeket a parancsokat értelmezi és lefuttatja, és az eredményt kiírja a felhasználónak karakteres formában. Amennyiben hibát észlel, akkor hibaüzenetet ír ki a felhasználónakA shellen belül a prompt egy néhány karakteres jel, ami azt mutatja, hogy az adott program a felhasználótól inputot, interakciót vár (pl.: gomb megnyomása, parancs megadása). Ha nem jelenik meg a prompt, a számítógép, az adott szoftver éppen dolgozik, egy parancs értelmezése vagy parancs végrehajtása közben van. Hiába próbálkozunk, nem tudunk újabb parancsot kiadni.

A mindig ismétlődő parancsok kiadására létrehozhatunk szkripteket (script), amivel automatizálható egy több parancsból álló feladat. A szkript ezeket a parancsokat tartalmazza, éslehetőséget ad egyben történő lefuttatásukra. A szkriptelhetőség a parancssori felület nagy előnye. Ennek (is) köszönhető, hogy bár manapság már a grafikus felhasználói felület szinte minden szoftverrel szemben elvárt igény, ettől függetlenül bőven vannak olyan szoftverek, amelyeket kizárólag parancssori felülettel lehet igénybe venni. Különösen a Unix/Linux világban, illetve az üzemeltetők és fejlesztők körében nagyon elterjedt ez a fajta interfész. Bizonyos parancsok kiadása billentyűzeten sokkal gyorsabb tud lenni, mint egeret használva klikkelgetni és adatot megadni. A szkriptek futtatása pedig messze gyorsabb, mint a grafikus felhasználói felületen való monoton munkavégzés. A parancssori felület emellett támogatja a távoli elérést is. Távoli terminálon kiadott parancs hálózaton keresztül tud eljutni az adott számítógépre, adott szoftverhez, ahol a parancs végrehajtását követően a válasz hálózaton keresztül jut vissza a terminálra. A terminálon tehát a korábban kiadott parancs kimenete fog megjelenni. Nem kell fizikailag azon gép előtt ülni, ahol éppen kiadjuk a parancsot, hiszen hálózaton fog közlekedni a parancs, illetve a parancsnak az eredménye. Karakteres parancssori felület esetén ez nagyon kevés hálózati forgalommal jár, ugyanis csak pár karaktert kell átküldeni a hálózaton. Nem kellenek bonyolult eszközök, erős processzor, sok memória, nagy hálózati sávszélesség, így bizonyos esetekben gyorsabb feladatvégzést tesz lehetővé, mint grafikus felhasználói felületet használva.

Ha egy standalone szoftvert szeretnénk használni, akkor azt másolnunk és/vagy telepítenünk kell a számítógépre. A telepítés történhet közvetlenül, azaz az adott gép előtt ülve, vagy - mivel napjainkban már a különböző operációs rendszerek biztosítanak rá lehetőséget - központilag, távolról. Különböző szoftverek esetén, különböző kompatibilitási

problémák is előfordulhatnak. (pl.: Windows DLL hell: Ilyenkor az A szoftver használ egy bizonyos DLL-t és B szoftver is használja ugyanazt a DLL-t, tehát mindkettő használ egy különálló harmadik komponenst. Fejlesztés révén kijöhet egy 2.0-ás verziójú DLL, ami gondokat okozhat, mert az egyik szoftver az adott DLL 1.0-ás verzióját szeretné használni, míg a másik szoftver a 2.0-ás verzióját. Hogyha a számítógépen csak az 1.0 verzió van fenn, akkor az egyik szoftver nem fog működni, ha a 2.0 verzió, akkor a másik.

A szoftverek fejlődése új verziók megjelenésével jár együtt, ami viszont a szoftverek folyamatos frissítését is maga után vonja. Ez is történhet az adott számítógépnél, vagy történhet központilag, távolról menedzselt módon. Fontos megjegyezni, hogy a szoftverek általában konfigurálhatók, azaz személyre szabhatók. Megadható, hogy nálam az adott szoftver egy bizonyos módon működjön, míg egy másik felhasználónál máshogy működjön, de konfigurációban adjuk meg azt is, ha ugyanazt a szoftver két teljesen különböző környezetben szeretnénk futtatni különböző beállításokkal. A környezet valójában azt jelenti, hogy más komponensekkel van kapcsolatban, más komponensekkel akar kommunikálni és/vagy ezek a komponensek máshol helyezkednek el. A konfigurációk különböző formátumúak lehetnek. A konfigurációs állományok a leggyakrabban szöveges formátumúak, azaz a konfigurációt egy egyszerű szöveges állományba írjuk le (például az ini vagy properties állományok, amiben kulcs-érték párok szerepelnek, amik a beállításokat tartalmazzák). Ezenkívül lehetnek olyan szoftverek is, melyek a saját konfigurációjukat bináris állományban tárolják. Ezek kezelése kicsit nehézkesebb, hiszen a bináris állományt nem tudjuk egy egyszerű szövegszerkesztővel megnyitni és szerkeszteni. Általában külön felhasználói felület áll rendelkezésre az ilyen szoftvereknél, ahol beállítást követően a paraméterek bináris állományba kerülnek lementésre. Az operációs rendszerek is nyújthatnak szolgáltatást arra vonatkozóan, hogy konfigurációt tartalmazzanak. (pl.: Windowsban lévő registry, amely egy központi tárhely a különböző szoftverek konfigurációjának a tárolására).

Egy standalone alkalmazás – ami csak egyetlen számítógépen fut – is több részből áll. A leggyakrabban említett részek, állományok: maga a futtatható alkalmazás, ez tipikusan már le van fordítva. Ez azt jelenti, hogy egy bináris állományunk van, amely a processzor nyelvén van megfogalmazva. Miért a processzor nyelvén? Mert a bináris formátumot a processzor sokkal gyorsabban képes értelmezni, mint egy szöveges formátumot (azaz a forráskódot). Így a parancsok nem szövegesen vannak leírva, hanem binárisan – ember számára értelmezhetetlen módon. Erőforrás állományok is lehetnek egy alkalmazáson belül. Ilyenek pl. az adatokat tartalmazó állományok, képek, felhasználói felületet leíró állományok, a felhasználói felületen megjelenő szövegek (különböző nyelveken), súgó (egy olyan dokumentum, ami az alkalmazás szempontjából a futásához nem szükséges, mégis segítheti az alkalmazás t). Amikor a telepítés történik, akkor valójában ezeket az állományokat csomagolja ki a telepítő, és teszi az operációs rendszerben a megfelelő helyre (könyvtárba, könyvtárakba).

Az alkalmazás szempontjából fontos még, hogy naplózzuk a működését. Mi is az a naplózás? Az alkalmazás a benne lévő eseményekről üzeneteket állít elő. Tipikusan egy állományról, tehát egy fájlról beszélünk, de lehetséges képernyőn (konzolon) való megjelenítés is. A konzolra naplózásnak hátránya, hogy nyom nélkül eltűnik. Ha viszont fájlba történik a naplózás akkor azt később is bármikor vissza tudjuk keresni. A napló formátuma döntően

szöveges, sorokba rendezett. Minden egyes, a szoftverben történő eseményről egy sor kerül leírásra. A sor általában tartalmazza az adott esemény idejét (timestamp), azt hogy a szoftver melyik komponense írja ki az adott naplóüzenetet, és persze magát az üzenetet is. Miért is fontos nekünk egy napló állomány? Abban az esetben, ha egy alkalmazáson belül hiba keletkezik, akkor ez gondos programozás esetén megjelenik a naplóban is, méghozzá sokkal részletesebb információkkal, mint azt a felületen láthatnánk. A naplónak nagyon fontos szerepe van tehát a monitorozásban, az alkalmazás belső működésének megfigyelésében, a hibakeresésben. Így például vissza tudjuk követni, hogy egy felhasználó mit is csinált, mikor az alkalmazásunk éppen hibára futott.

Egy külön típusa a naplózásnak az audit naplózás, amit nagyfokú biztonságos rendszerekben használnak, mint például a banki rendszerek. Ez azt tartalmazza, hogy ki mikor milyen módosítást végzett a rendszerben. Ez esetleg még személyi adatokat is tartalmazhat, így akár titkosításra is szükség lehet.

A napló igen nagy mennyiségű adat, ahol az adat ömlesztett formában található. Információ kinyeréséhez szükséges az adatok eszközökkel történő feldolgozása. A napló állomány tartalma, és hogy miről készítsen az alkalmazás napló üzenetet, a fejlesztőkön múlik, ők döntenek róla. Egy felhasználónak, egy tesztelőnek, egy üzemeltetőnek maximum annyi lehetősége lehet, hogy milyen napló üzenetek kerüljenek kiírásra. Egy modern rendszer esetén viszont ez akár az alkalmazás futása közben is módosítható. Miért szükség állítani a naplózás mértékét? Amennyiben mindig minden naplózás be lenne kapcsolva, nagyon nagy mennyiségű napló állomány keletkezhet, mely lassíthatja a rendszer működését, sőt akár a tárhely is megtelhet.

Miért szeretjük a napló állományokat? Abban az esetben, ha olyan alkalmazásunk van, amihez mi nem férünk hozzá, de fejlesztőként vagy tesztelőként meg akarjuk nézni milyen hibák fordulnak elő benne, akkor a napló állomány lehet segítségünkre. Ha sok felhasználónk van, akkor szinte lehetetlen egyszerre követni, hogy mit csinál az a sok-sok felhasználó. Viszont egy napló állományból ki lehet szűrni az adott felhasználóra vonatkozó napló üzeneteket, nyomon követhetjük az adott felhasználó tevékenységét a rendszerben. Elosztott rendszerekben, ahol egy adott rendszer több komponensből épül fel, és ezek különböző számítógépekre vannak telepítve, illetve egy olyan folyamat megy végig, amely mindegyik komponenst igénybe veszi, akkor nagyon nehéz összehasonlítani, hogy mikor, mi történt az egyes komponensekben. A napló állományok segítenek ebben is. A különböző rendszerek napló állományait felhasználva, dátum szerint összefésülve, választ kaphatunk a kérdéseinkre.

Standalone alkalmazás grafikus felülettel

Ha történetileg akarjuk vizsgálni, akkor a szoftver architektúrák fejlődésének következő lépcsőfoka, hogy a standalone alkalmazások grafikus felületet kaptak. A standalone alkalmazás azt jelenti, hogy egy számítógépre van telepítve az alkalmazás. A grafikus felület pedig azt, hogy immár nem parancssorral, hanem grafikus felületen keresztül is lehet az adott szoftverrel kommunikálni.

A szoftver és a felhasználó közötti kommunikációs felület a user interfész, azaz UI. Ez lehet karakteres vagy grafikus is. A felhasználó ezen a felületen adja meg a parancsokat a számítógépnek, a szoftver ezt értelmezi, futtatja, majd az eredményt ugyanezen a felületen adja vissza a felhasználó számára.

Amennyiben ez grafikus, szerepelnek rajta színek, különböző betűtípusok, esetleg képek, grafikus felhasználói felületnek, graphical user interface-nek (GUI) nevezzük. Itt már nem csak karakterek jelennek meg a felületen, hanem grafikus komponensek is, mint például beviteli mezők, legördülő menük stb. A grafikus felhasználói felület értelmezése és használata az átlag felhasználó számára sokkal egyszerűbb. A grafikus felhasználói felületek intuitívak, jó esetben nem kell a súgót elolvasni, azonnal használható. A grafikus felületek tipikusan ablakozó rendszeren belül jelennek meg. Mind a Windows, mind a Linux grafikus felületei már ablakokat jelenítenek meg. Az alkalmazások különböző ablakokban futnak és ezek az ablakok egymáshoz képest tetszés szerint elrendezhetők. Jelentősen különbözik abban a CLI megoldástól, hogy a felületeket grafikusan vezérelni tudjuk, de ehhez a billentyűzet mellett, az egeret és6vagy az ujjunkat (touch screen) is használhatjuk, amivel a képernyőnek egy bizonyos pontjára tudunk pozícionálni.

A grafikus felület úgynevezett grafikus komponensekből, vagy widgetekből épül fel - több megnevezésük is ismert. Két nagy csoportba sorolhatók: az adatokat megjelenítő elemekre (címkék, szövegek, táblázatok) és adatbeviteli elemekre (legördülő menük, rádió gombok, gombok). Ez utóbbiakkal jelezni tudjuk a szoftver számára, hogy valamiféle szándékunk van. A grafikus komponensek fa struktúrában jelennek meg, ami azt jelenti, hogy van egy szülő komponens, másnéven ős komponens. Ez tipikusan az ablak, amiben az összes többi komponens elhelyezkedik. Az ablakon belül helyezkedhetnek el panelek, a paneleken belül további panelek és azokon belül akár további komponensek is, mint például egy táblázat. A táblázaton belül címkék fordulhatnak elő, sőt vannak olyan táblázatok is, amelyeken belül szövegbeviteli mezők is elhelyezkedhetnek. A komponensek hierarchikus felépítése tehát egy fa struktúrát rajzol ki.

Érdemes még a grafikus felületeknél a skinekről beszélni. Skinek használatával tehetjük a szoftver megjelenítését egyedivé. Ha nem használunk skint, akkor a szoftver megjelenése többnyire az adott operációs rendszer beépített megjelenéséhez (ablak kinézet, színek) fog hasonlítani.

Egyáltalán nem biztos, hogy egy fejlesztő vagy üzleti elemző jó felhasználói felületet tud rajzolni, tervezni vagy implementálni (megvalósítani), ezért a felhasználói felületek tervezése egy külön tudomány, ami további alszakmára bontható. Az egyik a design, mely a megjelenítéssel kapcsolatos dolgokról szól. Például milyen színeket, térközöket, betűtípusokat használjunk. Egyszerűnek hangzik, de a szépérzéken kívül nagyon sok tanulás és gyakorlás kell hozzá. Attól, hogy egy szoftver szép, még nem biztos, hogy használható is, erre van egy külön tudomány terület, a user experience (UX design), ami a felhasználói felület használhatóságáért felel. Például a szoftver legyen intuitív, ha ránézünk, akkor azonnal tudjuk, hogy hova kell klikkelni, azonnal tudjuk használni. Legyen beszédes és szellős a felhasználói felület, ne vonja el a figyelmünket sok kis apró részlet. Mindig tudjuk, hogy mit kell csinálni, milyen lehetőségeink vannak és mi lesz a következménye. Érdemes figyelembe venni azt is, hogy az egér mellett/helyett nagyon sokan szeretnek még

billentyűzetet használni. Az alkalmazás legyen felkészítve arra is, hogy a legtöbb funkciót billentyűzettel is, akár egér használata nélkül is igénybe lehessen venni. Ráadásul a különböző egyéb eszközök megjelenésével, mint például a mobiltelefon vagy tablet, ahol az ujjunkkal tudjuk vezérelni a szoftvert, megint csak máshogyan kell ehhez a témakörhöz hozzáállni. A felhasználói felületek tervezésénél figyelni kell továbbá az akadálymentesítésre, hiszen vannak látássérültek is, akik az adott szoftvert használni szeretnék. A használhatóságért nagyobb betűtípust, nagyobb kontrasztot használunk. Ám talán a legfontosabb az, hogy az adott szoftverek megfelelően támogassák a szövegfelolvasó programokat.

Több érdekes technológiai vonatkozása is van a grafikus felhasználói felületeknek. A szoftverek esetén külön technológiát alkalmazunk, amivel leírjuk a felhasználói felületet (hogyan néz ki, hogyan épül fel, milyen widget-ek vannak rajta) és külön technológiát az alkalmazás funkcionalitásának megvalósítására (adatok beolvasása, adatokat kiírása, különböző üzleti folyamatok megvalósítása). A felhasználói felületnél leíró jellegű, azaz deklaratív a technológia (valamilyen leíró, pl. egy XML, HTML formátum), a funkcionális működés esetében imperatív (valamilyen programozási nyelv).

Docker telepítés Windows 10 Home esetén

Ellenőrizzük a Windows verziót, ha nincs naprakészen tartva, frissítsünk. (Windows key + R, Futtatás ablak, `winver` parancs.) Jelenlegi verzió a telepítendő gépen: Verzió: 2004 (Build 19041.630)

A Docker Desktop letölthető a https://hub.docker.com/editions/community/docker-ce-desktop-windows/ címről. Itt válasszuk ki a stable channel lehetőséget. Töltsük le a telepítőt (450+ MB) és indítsuk el, hagyjuk jóvá az esetleges kérdéseket. A telepítés végén jelzi, hogy újra kell indítani a Windowst. A telepítés során megkezdi a WSL 2 párhuzamos telepítését is (Windows Subsystem for Linux Windowsra fordított Linux kernellel).

Az újraindítás után, a Docker indításakor jelzi, hogy a WSL 2 telepítés nem teljes, a megadott linken kövessük a letöltési utasításokat (`wsl_update_x64.msi` csomag kell még kiegészítőnek). Telepítsük ezt a csomagot is, és újabb újraindítást kér a rendszer.

A Windows újraindítás, a Docker első indítása lassan megy, legyünk türelmesek. Komoly rendszer módosítások és frissítések történnek közben.

Próbáljuk ki az image letöltés és futtatás menetét, akár a letöltési felületen megadott `docker run hello-world` paranccsal.

Központi adatbázis

A architektúrák fejlődése terén a következő lépés a központi adatbázis megjelenése volt. A standalone szoftverek az adataikat az adott számítógépen tárolták, egy szöveges vagy bináris állományban. Ennek megvolt az a veszélye, hogy ha az adott számítógép esetleg megsérül, akkor az adatok is elvesznek. További hátránya az volt, hogy csak az adott

szoftver tudta használni az adatokat, csak azon a számítógépen, amin telepítve volt. Ebből adódóan egyszerre csak egy felhasználó tudott hozzáférni és tudta használni az adott szoftvert, ami ezt a fájlt használta.

Amikor megjelentek az adatbázisok, ezek a problémák megszűntek. Az adatbázisok olyan szoftverek, amelyek adatok perzisztens tárolására, lekérdezésére és módosítására alkalmasak. A perzisztens szó itt azt jelenti, hogy hosszútávú, állandó. Ez azt jelenti, hogy nemcsak a számítógép memóriájában maradnak meg az adatok, hanem hosszabb távon, akkor is, amikor a számítógépet kikapcsoljuk. Ennek hátterében az áll, hogy az adatok kiírásra kerülnek valamilyen háttértárra (winchester vagy SSD), amelyek megőrzik az adatokat kikapcsolt állapotban is.

Az adatbázis klasszikusan CRUD (Create, Read, Update Delete) operációkat enged az adatokon. Azokat az alkalmazásokat, melyekben nincs se üzleti logika, se bonyolult folyamat, gyakran CRUD alkalmazásnak is hívjuk, ezek csak adat felvételre, lekérdezésre, módosításra és törlésre valók. Egy ilyen alkalmazásnak a grafikus felülete főként táblázatokból áll, melyben a sorok tartalmát tudjuk szerkeszteni, új sort felvenni vagy meglévő sort törölni.

Az adatbázisok két fő csoportba oszthatók: a legelterjedtebbek a relációs adatbázisok, amelyek az adatokat úgynevezett táblákban, strukturáltan tárolják. Van hozzájuk egy szabványos nyelv is amellyel az adatokat lehet olvasni, lekérdezni, módosítani vagy törölni. Ez az SQL nyelv. Az SQL nyelv szerencsére szabványos, ami azt jelenti, hogy bármelyik relációs adatbázis elé kerüljünk is, ha ismerjük az SQL nyelvet, akkor tudunk adatot beszúrni vagy lekérdezni, nem kell megtanulni az adott adatbázisnak a speciális nyelvét. Ezen kívül léteznek úgynevezett NoSQL adatbázisok is, látható, hogy ezek a nevükben is az SQL-t tagadják, ezek speciális igények mentén jöttek létre. Akkor használjuk őket, ha az adatok táblákban való tárolása nem megfelelő a számunkra.

Felmerülnek az alábbi kérdések is: Hogyan épülnek fel az adatbázisok? Hogyan lehet őket használatba venni? Léteznek beágyazott adatbázisok, ezek az alkalmazás mellett vannak, ugyanazon a számítógépen futnak, nem igazán lehet kívülről hozzájuk férni. Gyakorlatilag ez azt jelenti, hogy az alkalmazáson belül van az adatbázis, az alkalmazás önmagán belül ezt az adatbázist használja. Különálló adatbázis esetén jobb a helyzet. Nevének megfelelően ez egy önálló szoftver, amit külön kell telepíteni, és ehhez kapcsolódik az alkalmazás. Ez lehet az alkalmazással azonos gépen, de eltérő gépen is, mert az adatbázisokkal hálózaton keresztül is fel lehet venni a kapcsolatot. Ha egy külön számítógépen helyezzük el az adatbázist, legnagyobb előnye, hogy nemcsak egy szoftver tud hozzá kapcsolódni, hanem tetszőleges számú szoftver el tudja érni. Akár mi, felhasználók is meg tudjuk így nézni, hogy az adott szoftver milyen adatokat tárol.

Milyen jellemzői vannak az adatbázisoknak? Az adatbáziskezelő rendszerek (rational database management system - RDBMS) is fájlba dolgoznak, azaz az adattárolás fájlokban történik. A fájlok pedig perzisztens táron (pl.: winchesteren vagy SSD-n) tárolhatóak. Az adatbázisok egyszerre több alkalmazást is kiszolgálhatnak, ráadásul az adatokat el lehet egymástól szeparálni. Különböző adatbázisokat, más néven sémákat lehet létrehozni az adatbáziskezelő rendszeren belül a különböző alkalmazásoknak és így nem kavarodnak össze az alkalmazások adatai. Az adatbázisok a fájlkezelés, illetve a hálózat kezelés

problémáit leveszik a fejlesztők válláról, illetve lehetővé teszik azt, hogy az adatokat valamiféle rendezett struktúrába szervezzék. Fontos, hogy az adatok nem ömlesztve szerepelnek az adatbázisban, hanem struktúrába szervezetten. Az adatbázis hasonló, mint egy excel állomány, vannak sorok, oszlopok, akár több lap is. Egy időben egyszerre nem csak egy felhasználó tudja használni, és a szoftverek is tudnak adatot tárolni benne. Az adatbázis lehetővé teszi a CRUD műveleteket (adatok beszúrása, lekérdezése, módosítása, törlése). A lekérdezés a megfelelő nyelv segítségével történik. Relációs adatbázisoknál tipikusan az SQL nyelven, SQL utasításokkal kell megadni a különböző módosításokat és lekérdezéseket. Ezeket az utasításokat az adatbázis értelmezi, ellenőrzi szintaktikailag, azaz megfelel-e a nyelvtani szabályoknak. Ha értelmezni tudja, akkor lekérdezi a megfelelő adatokat és visszaadja a felhasználónak, vagy módosítja azokat. Ha nem tudja értelmezni, mert pl. szintaktikai hiba van a lekérdezésünkben, azaz nem jól illeszkedtünk az adott lekérdező nyelv nyelvtanához, akkor hibaüzenetet fog adni, hogy a lekérdezést nem jól fogalmaztuk meg.

Hogyan történik az adatbázishoz való hozzáférés, hiszen az adatbázis akár egy másik számítógépen is lehet? Az adatbázisok interfészt biztosítanak az alkalmazások számára.

Az alkalmazások számára biztosított interfész maga az úgynevezett API (application programming interface). Ez valójában nem más, mint egy felület, ahol két számítógépes rendszer, vagy ugyanazon rendszer két komponense tud egymással beszélgetni. A felhasználók számára is ad természetesen hozzáférési lehetőséget egy adatbázis, ez viszont vagy CLI (parancssoros interface) vagy pedig GUI (grafikus interface). Az adatbázis lehetővé teszi, hogy egyszerre több alkalmazás is műveleteket (lekérés, módosítás stb.) hajtson végre vagy több felhasználó is tudjon közvetlenül kapcsolódni, tehát az utasításokat párhuzamosan képes végrehajtani. Hálózaton keresztül biztosított az adatbázisok távoli elérése is. Az adatbázisok nemcsak adatokat képesek tárolni, hanem programokat is tudnak futtatni. E programok dedikált elnevezése a tárolt eljárás, melyek az adatbázison belül futnak, az adatokon végeznek el műveleteket, átalakításokat.

Egy szoftver a publikált API-n keresztül tud egy adatbázishoz hozzáférni. Viszonylag bonyolult fogalom, magában foglalja a protokolokat, tehát a különböző szabályrendszereket, a funkciókat és különböző eszközöket is. Az API teszi lehetővé azt, hogy két szoftver, szoftver komponens egymással kommunikálni tudjon. A cél az, hogy amit az egyik szoftver biztosít (funkcionalitást) ahhoz egy másik szoftver hozzá tudjon férni a lehető legkevesebb költséggel, programozási tudással vagy programozási erőforrás befektetéssel. Amikor azt mondjuk, hogy az adatbázis egy interfészt biztosít a szoftverek számára, ezzel azt mondjuk ki, hogy az adatbázisnak van valamilyen funkcionalitása - adatok tárolása, lekérdezése, módosítása, törlése - és ezen funkciókhoz hozzáférést enged más szoftverek számára is. Azaz egy másik szoftver is igénybe tudja venni az adatbázis funkcionalitását, tud műveleteket végezni az adatbázisban. Ehhez kevés tanulás, kevés szakértelem szükséges, hiszen egy API-t úgy érdemes felépíteni, hogy a lehető legegyszerűbben használatba tudja venni a programozó. Fontos, hogy ezt a programozó veszi használatba, hiszen az API nem más, mint szoftver és szoftver közötti interfész, ahol a hívó szoftvert maga a fejlesztő implementálja, valósítja meg. Egy nagyon egyszerű párhuzamot is tudunk vonni. Amíg a UI egy interfész a felhasználó és a szoftver között, addig az API egy interfész a szoftver és a szoftver között.

Egy párhuzam a való életből az, mikor elektromos áramot használunk. A szolgáltatást az áramszolgáltató adja, és egy elektromos eszköz veszi használatba. A kettő közötti szabványos interfész, azaz API a konnektor.

Egy másik párhuzam egy étterem működése. Az ételt a szakácsok készítik el, és a vendégek fogyasztják el. A kettőjük közötti interfész, azaz API a pincér.

Ahhoz, hogy az adatbázisokra nagyvállalati, azaz komoly kereskedelmi szoftvereket is tudjunk építeni, fontos jellemzőkkel kell rendelkezniük. Külső forrásból, akár egy fájlból tudjuk betölteni az adatokat, ez az importálás., későbbi feldolgozásra másik helyre tudjuk kimenteni az adatokat, ez az exportálás. Az adatokat az adatbázis automatikusan titkosítani is tudja. (A GDPR miatt ez különösen fontos, hiszen nem lehet natív formában tárolni a természetes személyazonosító adatokat, ezeket védeni, titkosítani kell. Ha esetleg hozzá is férnek az adatbázishoz, akkor se tudják ezen adatokat elolvasni.)

Egy adatbázis naplózzon, írja ki milyen műveletek történtek benne, ki mikor mit csinált benne. Legyen benne archiválási lehetőség, azaz ki tudjuk menteni az adatokat, ha probléma van az adatbázisban, esetleg egy program vagy felhasználó elrontotta az adatokat, véletlenül kitörölt valamit, akkor az archívból ezeket vissza tudjuk állítani. Legyen magas rendelkezésre állású az adatbázis, azaz leállás nélkül tudjuk üzemeltetni, a nap 24 órájában működjön.

A felhasználó és jogosultság kezelés kiemelten fontos, azaz az adatokat csakis authorizáció, bejelentkezés után tudjuk használatba venni, kiolvasni. Adat szinten meg kell tudjuk mondani, hogy ki, mikor, mit olvasott ki, módosított (audit naplózás). Menedzselhető, üzemeltethető legyen az adatbázis, azaz meg tudjuk nézni, milyen állapotban van, milyen válaszidőkkel rendelkezik, párhuzamosan hány felhasználó vagy szoftver csatlakozik, ezek kapcsolódása megszakítható legyen (például, ha valaki egy erőforrásigényes lekérdezést indít el, akkor az üzemeltető ezt le tudja állítani erőforrás felszabadítása céljából). A teljesítménye vizsgálható legyen, illetve a különböző parancsokat optimalizálni lehessen. Az optimalizálás alatt azt értjük, hogy át lehessen fogalmazni ezeket, hogy a lehető legkevesebb erőforrást használjon fel, tehát kevesebb CPU vagy memória igénye legyen egy-egy lekérdezésnek, ezáltal pedig az adatbázis sokkal gyorsabban tudja a többi felhasználót és szoftvert is kiszolgálni. Legyen nagy teljesítményű a milliós nagyságrendű adattárolási igény folytán. Nem utolsó sorban tudjon különböző műveleteket elvégezni az adatokon (feltétel alapján szelektálni, más formátumba átkonvertálni az adatokat, aggregált műveleteket futtatni, mint például összegzés, átlag, vagy éppen minimum, maximum számítás).

A relációs adatbázisok erős matematikai háttérrel (ún. relációs algebra) rendelkeznek. Alapfogalma a tábla, mely adatokat tartalmazza. Egy adatbázis a gyakorlatban több táblából áll, és ezek a táblák adják meg az adatbázis struktúráját. Egy tábla oszlopokból áll, ezen kívül az adatok sorokban jelennek meg. Összességében nagyon hasonló tehát egy táblázathoz. Az oszlopoknak típusa van, pl. egész szám, lebegő pontos szám (tört), szöveges érték, dátum, bináris. Minden egyes skalár (egyszerű, nem összetett) adat azonosítható a tábla, az oszlop és a sor megadásával, ezzel egyedileg azonosítunk egy konkrét adatot. Ha nem ismert egy adat, mert nem tudjuk valami oknál fogva azonosítani, akkor erre egy

speciális értéket, a null-t szoktunk használni. Ez azt jelenti, hogy adott sorban és oszlopban szereplő adat, adott cella, értéke nem ismert.

Központi adatbázis (gyakorlat)

Portütközés

Mielőtt a MariaDB adatbázist elindítod Docker konténerben, győződj meg arról, hogy nincs MySQL/MariaDB telepítve a gépeden! Ha van, el kell távolítani, mert mind a MySQL és a MariaDB a 3306-os portot használja, így portütközés lesz (Docker hibaüzenet: port conflict, valamilyen megfogalmazással). Meg lehet próbálni, hogy átírod a parancsban a portot, például 3309-re, de ebben az esetben az összes további elérés esetében is ezt kell használni! Ha foglalt a port, kísérletezni kell tovább. A konténer ismételt létrehozása (a parancs ismételt lefuttatása) előtt a korábbi konténert el kell távolítani (pl. Docker Desktop felületen a konténer törlése). Ennek hiányában ismét Docker hibaüzenet lesz, miszerint a konténer név már foglalt.

A konténer törölhető parancssorból is:

```
docker rm locations-dbclient-mariadb
```

DBeaver

A DBeaver a https://dbeaver.io/download/ címről tölthető le. Válaszd ki a Windows 64 bit telepítőt és töltsd le! Indítd el az telepítőt, és fogadd el a javasolt beállításokat, licenszet! Hozzd létre parancsikont az asztalon!

SQL nyelv

Az SQL nyelv a relációs adatbázisok nyelve. SQL nyelven különböző utasításokat lehet megfogalmazni, mely utasításokat a relációs adatbázis értelmez, végrehajt és az eredményt visszaadja a felhasználónak. Klasszikusan CRUD műveleteket lehet kiadni. Create – létrehoz, Read – kiolvas, Update – módosít, Delete – töröl. Az SQL nyelv ezen kívül további is lehetőségeket is nyújt, mint például táblák létrehozása, törlése, felhasználók felvétele, stb. Az SQL nyelvnek van nyelvtana, szintaktikája, azaz szabályrendszere, melyben azok a szabályok foglalnak helyet, amik alapján egy SQL utasítást össze lehet állítani. Ha a parancs, amit kiadunk, ennek a szabályrendszernek nem felel meg, akkor szintaktikailag hibás, le se fogja futtatni az adatbázis, hanem azt fogja visszajelezni, hogy ez egy hibás utasítás, ki kell javítani. Az SQL (Structured Query Language) nemcsak lekérdezésre való. Ez egy DSL (Domain-Specific Language), azaz olyan speciális nyelv, amely csak egy részfeladatra fókuszál, közelebbről az adatok kezelésére. Tehát nem egy általános programozási nyelv. Ez a nyelv részben procedurális, tehát lépésekben tudjuk megadni, hogy mit csináljon az adatbázis, részben deklaratív, azaz mi csak megfogalmazzuk mire vagyunk kíváncsiak, a lépéseket és az eredményhez vezető utat maga az adatbázis határozza meg. Az SQL nyelv ma már szabvány, így bármelyik relációs adatbázissal akarunk dolgozni, ugyanazt az SQL nyelvet tudjuk használni. Azonban különböző dialektusai vannak, mely azt jelenti, hogy az

SQL nyelvnek a különböző adatbázisokban eltérő kiegészítései vannak. Ha ilyet használunk, az csak az adott adatbázison fog működni.

Az SQL további részekből áll. DDL: adatbázis táblák és különböző adatbázis objektumok létrehozása. DML: CRUD utasítások kiadása, adatokat kezelése. QL: lekérdezések írása, DCL: adatbázishoz kapcsolódó egyéb karbantartó műveleteket megfogalmazása (adatbázis/séma létrehozása, felhasználók felvétele, felhasználói jogosultságok beállítása).

Példa SQL utasítás tábla létrehozására:

```
CREATE TABLE questions (
    id bigint auto_increment,
    question_text VARCHAR(200),
    constraint pk_question primary key (id)
);
```

Az összetartozó adatokat egy kulcs jellemzi, az elsődleges kulcs azon oszlop, vagy oszlopok összessége, melyek soronként egyediek. Akkor beszélünk összetett elsődleges kulcsról, amikor több oszlop együttes értéke ad egy egyedi azonosítót. Az elsődleges kulcs való arra, hogy a sort egyedileg azonosítsuk. A relációs adatbázisokban a sorok sorrendje nem definiált. Többnyire rendezetlen, ahogy éppen az adatbázis el tudja őket tárolni. Így a különböző sorokat nem tudjuk a beszúrás sorrendjében lekérdezni, kizárólag valamelyik oszlop szerint átrendezve. Az elsődleges kulcs nem mindegyik adatbázis-kezelőben kötelező, de nagyon javasolt a használata, hiszen csak ez alapján tudunk egy sort egyedileg azonosítani. A legtöbb esetben az elsődleges kulcs nem más, mint egy generált vagy léptetett egész szám. Ezzel nagyon egyszerűen biztosítható az egyediség, ugyanis az újonnan bejött/felvett soroknak egy új egyedi számot ad az adatbázis.

Példa a beszúrásra, módosításra és törlésre:

```
INSERT INTO questions (id, question_text) VALUES
    (1, 'Mi az 5 + 5 kifejezés eredménye?');

UPDATE questions SET
    question_text = 'Mi az 5 + 8 kifejezés eredménye?' WHERE id = 1;

DELETE FROM questions WHERE id = 1;
```

Példa a lekérdezésre:

```
SELECT id, question_text FROM questions;

SELECT * FROM questions;

SELECT question_text FROM questions WHERE id = 1;
```

Táblák közti kapcsolatokat is lehet definiálni, leggyakoribb a szülő-gyerek kapcsolat. Példaként a kérdések és a hozzájuk kapcsolódó válaszok két külön táblában vannak tárolva. Valahogy meg kell feleltetni, hogy melyik kérdéshez melyik válasz tartozik. Ezt külső kulcs segítségével oldhatjuk meg. Külső kulcsnak nevezzük azt az oszlopot vagy oszlop halmazt

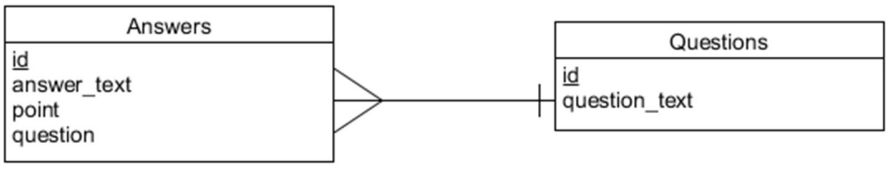
(mert több oszlop is lehet), amely egy másik tábla elsődleges kulcsának értékeit veheti csak fel. Grafikus ábrázolás is lehetséges ER (Entity Relationship) diagram segítségével. Általában egy téglalap a tábla, amelynek felső részében található a tábla neve, alsó részében az oszlop nevek, az elsődleges kulcs alá van húzva, a két téglalap közti kapcsolat is külön jelölést kap (tyúkláb). Tipikus az egy-több kapcsolat, azaz egy kérdéshez több válasz is tartozhat.

Külső kulcs megléte (FK) az összekapcsolásnak, a join műveletnek az alapja. Amikor nem egy táblából szeretnénk lekérdezést végezni, hanem több táblából, és ezek a táblák kapcsolatban állnak egymással akkor join műveletet kell kiadni.

Táblák közötti kapcsolatok

```
CREATE TABLE questions (
    id bigint auto_increment,
    question_text VARCHAR(200),
    constraint pk_question primary key (id)
);

CREATE TABLE answers (
    id bigint auto_increment,
    answer_text VARCHAR(200),
    point int,
    question bigint,
    constraint pk_answer primary key (id),
    constraint fk_answer_question
            foreign key (question)
            references questions(id)
);
```



Entity Relationship Diagram

Maga a join művelet:

```
SELECT question_text, answer_text
    FROM answers
    INNER JOIN questions ON answers.question = questions.id;
```

Adatbázis-kezelésben nagyon fontos a tranzakciókezelés. A tranzakciókezelés egy eszköz arra, hogy a párhuzamos műveleteket kezelni tudjuk és a hibákat ezzel ki tudjuk szűrni. Példaként egy banki rendszerben a banki átutalás valójában két műveletből áll: forrásszámla-levonás, célszámla-jóváírás. Ezt a két műveletet egyszerre kell elvégezni, de felmerülhetnek hibák, melyek hatására valamelyik művelet nem hajtódik végre. Azt várjuk,

hogy a kimenet csak ugyanaz lehet, azaz sikeres-sikeres vagy sikertelen-sikertelen. Ezeknek a hibáknak a kezelésére, találták ki a tranzakció fogalmát. Tranzakció esetén több műveletet vonunk egybe, ahol vagy minden művelet végrehajtásra kerül (commit) vagy egyik sem (rollback). Így tehát a tranzakcióban szereplő műveletek megbonthatatlanok, azaz atomiak.

A tranzakcióknak jellemző tulajdonságai vannak, melyeket az ACID betűszóval jelöljük.

- Atomicity: a tranzakcióban szereplő műveletek megbonthatatlanok, azaz atomiak.
- Consistency: a tranzakció előtt és a tranzakció után is konzisztens (ellentmondásmentes, azaz nincs ellentmondás az egyes táblákban tárolt adatok között) állapotban van az adatbázis.
- Isolation: a párhuzamosan, azaz ugyanabban az időpillanatban futó tranzakcióknak nincsen egymásra hatása, egymás működését nem befolyásolhatják, elszigeteltek egymástól. Az egymás mellett futó tranzakcióknak úgy kell működnie, mintha egymás után kerültek volna kiadásra, nem zavarhatják meg egymás munkáját.
- Durability: egy tranzakció eredményének tartósnak kell lennie, azaz perzisztensen meg kell maradnia. Ha kikapcsoljuk a számítógépet, az eredményül kapott adatnak akkor is meg kell maradnia.

A tranzakció után az adatot nem tárolhatjuk memóriában, hanem mindenképpen tartós tárban (winchesteren vagy SSD-n) kell tárolni, ugyanis csak ez biztosítja, hogy az adat a számítógép kikapcsolása után is, hosszú távon megmaradjon.

NoSQL adatbázisok

Bár az SQL adatbázisok nagyon elterjedtek, vannak azonban olyan speciális igények, feladatok, amelyre nem alkalmazhatóak. Ezen feladatoknak a megoldására jöttek létre a NoSQL adatbázisok. Ezek többfélék lehetnek. A kulcs-érték adatbázisok rendkívül egyszerű adatbázisok. Kulcs érték párokat tartalmaznak, viszont kulcs alapján nagyon gyorsan képesek előkeresni a hozzá tartozó értékeket. Léteznek sok oszloppal rendelkező úgynevezett oszlop alapú adatbázisok. Egy relációs adatbázis esetén az oszlopoknak a bővítéséhez magát a táblát kell módosítani, amely nagyon gyakran nem is engedélyezett az alkalmazások számára. Egy oszlop alapú adatbázis esetén viszont az oszlopokat pontosan úgy bővíthetjük, mint a sorokat. Ezenkívül ismertek úgynevezett dokumentum alapú adatbázisok. Bizonyos esetekben nem lehet adatainkat táblákba rendezni, mert jóval bonyolultabb strukturáltság van egy adaton belül (ilyen lehet például egy bonyolult és összetett adásvételi szerződés). A dokumentum alapú adatbázisok közül a legelterjedtebb a MongoDB adatbázis, talán ez a legismertebb NoSQL adatbázis. A NoSQL adatbázis következő típusa a gráf alapú adatbázis. Sajnos a relációs adatbázisok nem tudnak megfelelően gráfokat, fa struktúrákat kezelni, nem tudnak lekérdezéseket hatékonyan futtatni gráfokra vonatkozóan. Viszont a gráf alapú adatbázisok pont erre lettek kitalálva. Tipikusan gráfokat, hálókat, és így kapcsolati hálókat tudnak kezelni, a gráfokon pedig különböző műveleteket tudnak elvégezni (például Facebook kapcsolatok).

A NoSQL adatbázisok akkor kezdtek megjelenni, mikor az elosztott működés egyre elterjedtebb lett, egyre széleskörűbbé vált. A legfontosabb példája ennek, hogy egy adatbázis esetén lényeges a magas rendelkezésre állás (0-24 működés). Ezt viszont viszonylag nehéz megoldani, hiszen képzeljük el, ha egyetlen számítógépre van feltelepítve az adatbázis és az a számítógép meghibásodik, akkor onnantól kezdve leáll a teljes rendszer, tehát nem biztosítható a magas rendelkezésre állás. A magas rendelkezésre állás kizárólag további számítógépek bevonásával oldható meg. Mindezt úgy kell megvalósítani, hogy az adott adatbázist minden egyes számítógépre telepíteni kell és az összes adatnak minden számítógépen jelen kell lennie. Ekkor, ha az egyik számítógép megsemmisül, akár egy rövidzárlat, akár egyéb hardveres vagy szoftveres hiba miatt, a többinek még mindig tartalmaznia kell az adatokat, hogy ne legyen adatvesztés. Természetes lépés volt tehát az elosztott működés, azaz az adatbázisok több hardveren való működtetése. Ezzel kapcsolatban került megfogalmazásra a CAP tétel, ami azt mondja ki, hogy egy elosztott rendszer a három alapvető képesség közül legfeljebb kettőt tud megvalósítani. Az egyik képesség a konzisztencia, az ellentmondás mentesség. A konzisztencia tranzakció kezelésre vezethető vissza, tehát hogy tud-e az adott adatbázis tranzakciót kezelni. A következő a rendelkezésre állás, azaz ne legyen meghibásodás, az idő 100 százalékában elérhető legyen az adatbázis. A harmadik képesség pedig a particionálás-tűrés, ami azt jelenti, hogy amennyiben szétesik a hálózat (pl.: két számítógép között megszűnik a hálózati kapcsolat), akkor is működőképes maradjon az adatbázis.

Többrétegű alkalmazások

Ha történetileg nézzük, akkor a központi adatbázishoz kapcsolódó alkalmazások után a következő alkalmazás architektúra a többrétegű alkalmazás. Amikor csak egy központi adatbázisunk volt, akkor két rétegű alkalmazásról beszélünk. Valójában volt egy központi adatbázisunk, az volt az egyik réteg és ehhez csatlakozott egy szoftver, maga az alkalmazásunk, ami a másik réteg. Amikor azonban megjelentek a többrétegű alkalmazások, akkor a két rétegen felül még további rétegek jelentek meg. Valójában annyi történt, hogy bizonyos funkciókat az alkalmazásból átmozgattunk egy központi helyre, egy új rétegbe.

A kommunikáló felek szerint megkülönböztetünk szerver alkalmazást és kliens alkalmazást. Szerver alkalmazás az, ami szolgáltatást nyújt, szolgáltatást biztosít, valamilyen funkciót tesz elérhetővé - természetesen egy API-n keresztül, hiszen itt szoftver beszél szoftverrel. A másik szereplő pedig a kliens, ami a szolgáltatást igénybe veszi. A kliens a szolgáltatást, azaz valójában a szervert API-n keresztül hívja meg. A többrétegű alkalmazások megjelenésekor a kliens oldalon, a felhasználónál futó szoftverekből kiemelték az üzleti logikát, a működést és áthelyezték egy külön rétegbe. Ezt a réteget egy központi, nagy teljesítményű számítógépen helyezték el, és így gyakorlatilag szétválasztották magát az alkalmazást kliensre és szerverre. A kliens oldalon maradt a felhasználói felület, amely közvetlenül a felhasználót szolgálja ki. Szerver oldalra pedig átkerült az üzleti logika, azaz a bonyolult számítási műveletek, az algoritmusok, az üzleti szabályok. A kliensen gyakorlatilag semmi más nem maradt, mint a felhasználói felület - ami hálózaton keresztül csatlakozott a szerver alkalmazáshoz - , így ott a számítási igény jelentősen lecsökkent, hiszen felületet megjeleníteni, gombnyomást lekezelni, karakterek leütését figyelni viszonylag egyszerű

feladat. A szerver alkalmazáson viszont természetesen megnőtt a teljesítmény igény, ugyanis ott futottak le a különböző algoritmusok és üzleti logikák.

Ha grafikusan ábrázolni akarjuk, akkor hogyan is néz ki mindez? Van a szerver node, ami egyetlen számítógép és ezen a számítógépen fut a központi adatbázis, illetve az alkalmazás szerver oldali része. A felhasználóknál lévő számítógépek pedig a kliens számítógépek, melyeken a kliens alkalmazást futtatják, ahol csak a felhasználói felület van.

Amikor a felhasználó lenyom egy gombot, vagy billentyűt, akkor a kliens alkalmazás ezt észleli és hálózaton felveszi a kapcsolatot a szerver alkalmazással. A szerver alkalmazás ennek a hatására valamilyen számolást végez, gyakran valamilyen műveletet végez el az adatbázisban és az onnan visszaérkező adatot hálózaton küldi vissza a kliens számára.

Nézzük meg az egyes rétegek tulajdonságait! Általában a szerver alkalmazás egy relációs adatbázishoz kapcsolódik. A különböző rétegek nem feltétlenül ugyanazon a számítógépen jelennek meg. Az egyes rétegek között hálózati kapcsolat van, tipikusan TCP/IP-n keresztül kommunikálnak egymással. Az eddig megismert háromrétegű alkalmazások két gépen helyezkedtek el. Egy szerver gépen (adatbázis, szerver alkalmazás), illetve egy kliens vagy több kliens gépen (kliens alkalmazások). Azonban ezt tovább lehet strukturálni, méghozzá úgy, hogy az adatbázist kitesszük egy teljesen különálló számítógépre. E felállással marad az alkalmazásunk három rétegű, de a három réteg mindegyike különböző számítógépen helyezkedik el.

Milyen üzemeltetési vonatkozásai vannak a háromrétegű architektúrának? Mivel az alkalmazás legbonyolultabb és leginkább számításigényes részét átvittük szerver oldalon egy közös helyre, ezért azt központilag lehet menedzselni. Ha probléma van, központilag lehet beavatkozni. Egy helyen kell adatokat menteni és egy helyen kell újabb verziót kitenni a szerver alkalmazásból. Ha kétrétegű architektúránk van, tehát a kliens gépeken fut a teljes alkalmazás minden egyes üzemeltetési feladatnál (frissítés, hibaelhárítás) a kliens számítógépen kell munkálkodni.

Persze a háromrétegű architektúránál is egy réteg még mindig ott van a felhasználónál. Ha abban kell valamit javítani, frissíteni, ugyanúgy a kliens gépen kell megtenni. Ez továbbra is fennálló üzemeltetési nehézség, bár egy hatalmasat léptünk előre, hiszen a problémák javarészt szerver oldalon szoktak felmerülni a nagy számítási igény és az összetett algoritmusok miatt. Azzal, hogy a funkciók nagy részét átmozgattuk szerver oldalra, valójában egy úgynevezett “single point of failure”-t vezettünk be a rendszerbe. Ha ez a pont, a szerver, meghibásodik, akkor gyakorlatilag azonnal minden leáll, működésképtelen lesz a teljes alkalmazás, a hálózaton kapcsolódó kliensek, adatbázis egyaránt. Természetesen hasonló a helyzet a kétrétegű alkalmazásoknál, ha az adatbázis meghibásodik, az alkalmazásaink magukban nem használhatóak. A megoldás a redundancia biztosítása. A hibalehetőségek kivédésére több példányt kell futtatni a szerver oldali alkalmazásból és több példányt kell elindítani az adatbázisból is, különböző számítógépeken.

Webes alkalmazás

A háromrétegű alkalmazások után megjelentek a webes alkalmazások is. A kliens alkalmazások kliens számítógépekre való telepítése, majd karbantartása és frissítése nagy üzemeltetői munkával jártak. Erre nyújtott megoldást a webes alkalmazások megjelenése. A webes alkalmazásoknál elegendő a kliens számítógépen csak egy böngészőt telepíteni, ezt követően tulajdonképpen ebben a böngészőben fut a kliens oldal, tehát a böngésző jeleníti meg az alkalmazásnak a felhasználói felületét. Ez ugyanúgy egy többrétegű alkalmazás, mert külön van az adatbázis, külön van a szerver oldal és külön van a kliens oldal. Viszont a kliens oldalt már nem kell külön telepíteni, csak egy böngészőt kell indítani. A böngésző már mindegyik operációs rendszerben megtalálható, a felhasználók már rutinosan kezelik, nem kell külön megtanulni a használatát. A webes alkalmazások nagyon sok üzemeltetői problémát, feladatot tettek feleslegessé, innentől kezdve kliens oldalon már csak arra kellett figyelni, hogy legyen valamilyen böngésző a gépen (Google Chrome, Microsoft Edge, Mozilla Firefox, mindegyik ingyenesen letölthető és szabadon használható). Mivel böngésző várhatóan minden számítógépen van, ezért megszűnnek a telepítési és a frissítési problémák. A webes alkalmazást szokták vékony kliens alkalmazásnak is nevezni, utalva arra, hogy kliens oldalon nem kell bonyolult telepítési munkálatokat elvégezni, egyszerűen böngészővel használható az alkalmazás. A vékony klienst más kontextusban is szokták használni. Vékonynak nevezzük azt a klienst is, amiben nincs üzleti logika, nincsen benne algoritmus, nincsen benne bonyolultabb számítás, mert valójában minden szerver oldalon történik. Egy belső, cégen belüli hálózaton, intranet rendszeren még elképzelhető az, hogy az üzemeltető körbemegy és az összes kliensre telepíti vagy frissíti az adott kliens alkalmazást, de az interneten keresztül, akár több országban is használt alkalmazás esetén ez elképzelhetetlen. A böngészős alkalmazások esetén a használt technológia a HTTP. A HTTP protokoll az a hálózati protokoll, amelyen a böngésző és a szerver alkalmazás kommunikál egymással. A formátum, ami leírja a tartalmat ami megjelenik a böngészőben, az a HTML formátum. Ami pedig a megjelenést határozza meg, hogy milyen színek legyenek, milyen betűtípusok, mekkora térközök, azt a CSS formátum biztosítja. Ezenkívül ott van még a JavaScript programozási nyelv is, ami kezdetben egy olyan programozási nyelv volt, amivel böngészőben, kliens oldalon futó apró scripteket lehetett írni. Mára viszont a JavaScript egy általános célú programozási nyelvvé lépett elő, és akár szerver alkalmazásokat is lehet vele működtetni. (Nevével ellentétben nincs köze a Java programozási nyelvhez.)

A HTTP kapcsán ismerjük meg a legfontosabb fogalmakat A teljes web, és a webalkalmazások alapja a HTTP protokoll. Mivel a felhasználói felület, azaz a böngésző, HTTP-n kommunikál a szerver oldallal, ezért tisztában kell lennünk, pontosan mi is történik a háttérben. Egyrészt nagyon fontos az URL (Uniform Resource Locator) ismerete. Ez egy szöveg, amit a böngésző címsorába kell beírni, és egyedileg azonosít egy oldalt. Az interneten található erőforrások, oldalak, képek, hangfájlok, videó fájlok mindegyikének van adott url-je és ez abszolút egyedi. Az URL a protokoll, aztán a tartománynév, majd a port és végül az elérési út négyeséből épül fel. Például a training360-nak a weboldalán a hírlevél az elérhető a http://training360.com/hirlevél oldalon és ebből a protokoll a http, a tartománynév a training360.com. A portot többnyire nem írjuk ki, mert tudja a böngésző,

hogy HTTP esetén alapértelmezetten a 80-as porttal kommunikálunk, majd az elérési út a hírlevél.

A HTTP protokoll egy 1999-ben kiadott szabvány szerint épül fel. Pontosan az RFC7540-es szabvány definiálja. A W3C szabványügyi szervezet tartja karban a leírását. 2015-ben jelent meg a 2.0-ás verziója. A HTTP protokoll jellemzően egy kliens szerver kommunikációt tesz lehetővé, tehát mindig van egy kliens, ami a kapcsolatot kezdeményezi, - a legtöbb esetben ez egy böngésző -, a szerver pedig a HTTP kérésekre válaszol. Röviden összefoglalva először elmegy egy kliens által kezdeményezett kérés a szerverhez, majd erre a szerver egy válasz üzenetet küld vissza. Jól látszik, hogy a HTTP protokoll kérés-válasz alapú protokoll. Továbbá alapvetően szöveges protokoll, legegyszerűbben szövegeket tudunk rajta küldeni és fogadni. A HTTP protokollban üzenetek jönnek-mennek, a kérésnél is egy üzenet utazik a klienstől a szerver felé és a válaszban is egy üzenet utazik a szervertől a kliens felé. Az üzenet két részből áll, fejből és törzsből, a fejléc kulcs-érték párokat tartalmaz, a törzs tartalmazza magát az adatot (tartalmi részt). A HTTP protokoll állapotmentes, ami azt jelenti, hogy a kérések között semmiféle információ nem maradhat meg szerver oldalon, azaz a kéréseknek egymáshoz semmi köze nincs. Két egymás utáni kérés teljesen független egymástól. A HTTP protokoll önmagában nem titkosított, viszont kombinálni lehet más protokollokkal. Manapság a TLS protokollt szokták használni. Ezzel valójában a HTTP(S) használható, ami egy HTTP protokoll a TLS-en felül, ahol a TLS végzi a titkosítást. A HTTP kéréseket nagyon egyszerű monitorozni, minden egyes böngészőből egy fejlesztő eszköztárat lehet előhívni, melyben látható, hogy milyen kérések mentek a szerver felé a böngésző irányából és erre a böngésző milyen válaszokat kapott vissza.

A HTTP kérés egy fejből és törzsből álló szöveges üzenet, ahol a fejben csak kulcs érték párok szerepelnek. GET metódusú kérés esetén a böngésző szeretne a szervertől információkat lekérdezni. (POST metódus esetén a böngésző adatot küld fel a szerver oldalra.)

```
Accept: text/html,application/xhtml+xm…plication/xml;q=0.9,*/*;q=0.8
Accept-Encoding: gzip, deflate
Accept-Language: hu-HU,hu;q=0.8,en-US;q=0.5,en;q=0.3
Cache-Control: max-age=0
Connection: keep-alive
Cookie: _ga=GA1.2.1967894445.149906973…yis3b41advsb3cwc3b3rk; _gat=1
Host: training360.com
Upgrade-Insecure-Requests: 1
User-Agent: Mozilla/5.0 (Windows NT 10.0; …) Gecko/20100101 Firefox/57.0

GET http://training360.com/ HTTP/1.1
```

A szerver erre egy szintén fej és törzs részre bontható szöveges üzenetet küld válaszként. A fejrészben szerepel többek között a státusz kód, pl. 200, ami azt jelenti, hogy a szerver sikeresen válaszolt. A válasz törzsében egy HTML dokumentum található, amely leírja az adott weboldal tartalmát.

```
HTTP/1.1 200 OK
Cache-Control: no-cache
Pragma: no-cache
```

```
Content-Type: text/html; charset=utf-8
Content-Encoding: gzip
Server: Kestrel
Set-Cookie: Training360.Session=CfDJ8...s5; path=/; secure; samesite=lax;
httponly
Date: Fri, 09 Aug 2019 11:45:51 GMT

<!DOCTYPE html>
<html lang="hu">
<head>
  <!-- ... -->
</head>
<body>
  <!-- ... -->
</body>
</html>
```

Azonban egy weboldal nem csak szövegből áll, hanem különböző egyéb állományokból is. Például képekből, a megjelenítést elősegítő CSS állományokból, valamint a működést leíró JavaScript állományokból. A kérésre visszakapott válaszban jön tehát a HTML oldal. Ez az oldal a szöveges weboldal tartalom mellett külső hivatkozásokat is tartalmaz további állományokra. A böngésző a választ feldolgozza, a külső állományokra vonatkozó hivatkozásokat külön HTTP kérésben kérdezi le, azaz ugyanolyan szöveges kérésben, mint előzőleg, hiszen ezeket a fájlokat további URL-ek azonosítják. Végül a böngésző megjeleníti a HTML oldalt, úgy hogy összekombinálja a HTTP tartalmat, képeket, stb.

A HTTP szabvány szerinti státusz kódok a következők:

- 200-as státusz kódok: sikeresen visszatért a szerver a válasszal.
- 300-al kezdődő státusz kódok: sikeresen visszatért ugyan, de a böngészőnek valami egyéb dolgot is kell csinálnia, pl. átirányítania, stb.
- 400-as státusz kódok: a böngésző rossz kérést küldött el a szervernek és a szerver ezzel nem tud mit kezdeni. Böngésző oldalon kell valamit változtatni.
- 500-al kezdődő státusz kódok: szerver oldalon probléma van. Szerver oldalon kell valamit változtatni.

A 400-asok közül a legismertebb státuszkód a 404-es, azaz a Not found státusz kód. Ha egy nemlétező URL-t írunk be a böngészőbe, azaz olyan URL-t, amely mögött nincs semmiféle erőforrás, akkor a 404-es státusz kódot fogjuk visszakapni.

Amennyiben lekérünk egy erőforrást, ezt paraméterezni is lehet, méghozzá URL paraméterekkel. Leggyakrabban keresőknél szoktuk ezt használni. Ha a https://training360.com oldalon szeretnénk keresni a kurzusok között, vagy egy használtautó portálon szeretnénk az autók között keresni, illetve egy ingatlan hirdetéseket tartalmazó oldalon szeretnénk az ingatlanok között keresni, akkor a keresési feltételeket tipikusan URL paraméterben szoktuk átadni. Az URL-ben a path után kérdőjellel elválasztva

kell megadni a paramétereket. A paramétereket egymástól & jellel kell elválasztani. A paraméterek kulcs-érték párok, ahol a kulcs egyenlőségjellel van elválasztva az értéktől.

A HTTP nagyon bonyolult dolgokat is deklarál, például a *cache*-lést (gyorsítótár). A cache a böngésző által kezelt tárhely a kliens gépen, ahol a már letöltött oldalakat és erőforrásokat tudja tárolni. Miért van erre szükség? Azért, mert ha a felhasználó felkeresi újra és újra ugyanazt az oldalt, akkor ismételten ne kelljen letölteni a fájlokat. A böngésző a cache-ben eltárolja ezeket az állományokat és legközelebb nem kell letölteni a szervertől. Ezzel tehermentesíti egyrészt a szervert másrészt pedig hálózati forgalom sincsen. A szerver a header-ben (cache-control fejléc) tudja megmondani a böngészőnek, hogy mennyi ideig tárolhatja a cache-ben ezeket az állományokat. Ha letelik az idő, akkor a böngészőnek mindenképpen újra le kell kérni az állományt. Ha manuálisan töröljük a cache-t, persze akkor is. Hasonló böngészőbe beépített mechanizmus az is, amikor a kliens küld a szervernek egy kérést, hogy egy oldal meghatározott ideje/legutóbbi lekérdezés óta változott-e? (if-modified-since fejléc) A böngésző megjelenítheti a tartalmat a cache-ből, ha azóta nem történt módosítás. Ezt a szerver 304-es státusz kóddal (not-modified) tudja a kliensnek jelezni. Vagy visszaadhatja a szerver, hogy történt módosítás, ekkor természetesen a böngésző lekéri az új dokumentumot és azt jeleníti meg.

Ahogy láttuk a HTTP segítségével nem csak HTML oldalakat, hanem CSS fájlokat, JavaScript fájlokat, valamit képeket, videókat, illetve hang állományokat is le lehet tölteni. Azonban a böngészőnek meg kell mondani, hogy ez pontosan micsoda, erre való a *MIME type*. A HTTP protokollban nem a kiterjesztés szolgál erre, mint az operációs rendszerek esetén, hanem a _MIME type. A _MIME type-ot a `Content-Type` headerben adja meg a szerver. Ezek közül néhány példa: a `text/html` úgy érzékeli, hogy ez egy html állomány vagy text/javascript és így tovább. Amikor mondjuk a https://training360.com oldalról letöltünk egy Word dokumentumot, akkor ugyan igaz, hogy a kiterjesztése `.doc` vagy `docx`, azonban ez a HTTP szabványban nincs benne. Helyette a szerver oldal a `content-type` headerben leküldi a kliensnek, hogy ennek a MIME type-ja `application/msword` és abból tudja a böngésző azt, hogy ez egy Word állomány és a Word alkalmazást kell megnyitni ahhoz, hogy szerkeszteni lehessen ezt az állományt.

A GET metódus szolgál arra, hogy szerver oldalról adatokat, például egy weboldalt kérjünk le. A POST metódus szolgál arra, hogy adatokat vigyünk fel, például, ha van egy adatokkal kitöltött űrlapunk, melynek adatait fel szeretnénk küldeni a szerver oldalra. Ekkor a POST metódust használjuk és ugyanúgy egy szöveges HTTP kérés megy fel szerver oldalra. Annyi a különbség, hogy a HTTP kérésnek a törzsében szerepelnek majd az űrlapban megadott értékek név-érték párok formájában, nem az URL-ben. A POST metódus hatására a szerver oldalon többnyire valamiféle változás történik (például banki átutaláskor kitöltjük, hogy honnan hova szeretnénk átutalni, és akkor a szerver oldalon is változni fognak adatok, hiszen megtörténik maga az átutalás.)

```
POST /contact.aspx HTTP/1.1
Host: http://training360.com
User-Agent: Mozilla/5.0 (Windows NT 10.0; …) Gecko/20100101 Firefox/57.0

UserName=demo@example.com&Password=password
```

A sütik (cookies) szintén a HTTP protokollban definiált elemek. A sütik valók arra, hogy a szerver oldal a böngészőben valamiféle adatot el tudjon tárolni. Képzeljünk el például egy olyan webshopot, ahol meg tudjuk mondani, hogy milyen nyelven jelenjen meg a felület, illetve milyen pénznemben írja ki az árakat. Ekkor a legegyszerűbb, hogy ha ezeket a választott értékeket, tehát a kiválasztott nyelvet és a kiválasztott pénznemet sütikben tárolja el a szerver oldal. A süti egy kis tárterület a böngésző oldalán és minden egyes HTTP kéréskor elküldi a süti tartalmát az adott szervernek. A süti tehát a böngészőben (kliens oldalon) kerül tárolásra és onnantól kezdve minden egyes kérésnél elküldi a böngésző, hogy milyen nyelv és milyen pénznem lett kiválasztva. A szerver oldal ennek alapján tudja visszaadni a megfelelő oldalt. A sütiknél esetében van egy nagyon lényeges biztonsági szempont, mégpedig az, hogy minden egyes sütit csak oda lehet visszaküldeni, ahonnan a böngésző először megkapta. Ez fontos, mert nem lenne túl szerencsés, hogy ha a banki alkalmazás által visszaküldött sütit egy más alkalmazás is ki tudná olvasni.

A HTTP protokollban a HTTP kérések teljes mértékben függetlenek egymástól, azaz nem lehet értékeket átvinni a két kérés között, másként fogalmazva nincs kapcsolat a két kérés között. Azonban egy webalkalmazás fejlesztésénél hamar belefutunk abba, hogy erre szükség lenne. Ha egy felhasználó bejelentkezik egy webshopba és indít egy kérést, akkor mi tudni akarjuk, hogy a kérést melyik felhasználó küldte. Természetesen erre is van lehetőség amellett, hogy a HTTP továbbra is állapot mentes kell legyen. A következő trükk van a háttérben. A szerver oldalon fenntartanak egy memória terület, ami tárolja az adott felhasználóhoz tartozó adatokat. Ezt hívják _sessionne_k vagy munkamenetnek. Amikor a session létrejön, vele együtt generálódik egy egyedi azonosító és ezt az egyedi azonosítót leküldi a szerver a böngészőnek sütiben. Ha a böngésző egy újabb kérést indít a szerver felé, akkor elküldi ezt a sütit, benne az egyedi azonosítót, a szerver pedig a sessionből ki tudja venni az adott felhasználóhoz tartozó adatokat. A sessionnél, mindig van egy *timeout*, azaz, ha a felhasználó bizonyos ideig nem intéz kérést a szerver felé akkor a session ideje lejár, a memória terület megszűnik az adott felhasználó számára a server oldalon (ez történik akkor is, ha a felhasználó kijelentkezik adott oldalról), és a sessionben tárolt adatok elvesznek.

A következő jellemzője a HTTP protokollnak, hogy létezik egy úgynevezett átirányítási mechanizmus is. Mikor praktikus az átirányítást használni? Nézzük a következő működést! Egy banki alkalmazásban megjelenik egy átutalási űrlap, majd az adatok megadása után megnyomjuk az átutal gombot, és utána kapunk egy új oldalt, amin ott az értesítés, hogy az átutalás sikeresen elvégzésre került. Nézzük meg, hogy ez hogy is néz ki a HTTP protokollban. Először elmegy egy GET kérés a szervernek, aminek hatására a szerver visszaküld egy üres űrlapot. Majd a felhasználó kitölti az űrlapot és megnyomja az átutal gombot. Ekkor a böngésző a szerver oldalnak elküld egy POST-ot és a HTTP kérés törzsében elküldi az űrlap tartalmát. A szerver elvégzi az átutalást és a válaszban visszaküldi azt, hogy maga az átutalás sikeresen elvégzésre került. Igen ám, de ha ebben az esetben megnyomjuk a vissza gombot vagy megnyomjuk a refresh gombot vagy esetleg könyvjelzőzzük az adott oldalt, akkor onnantól kezdve működési anomáliákba fogunk belefutni. A legjobb esetben csak annyi történik, hogy a böngésző megkérdezi: biztosan újra akarod küldeni az adott űrlapot? Rosszabb esetben egy üres oldalt kapunk, amin nem látszik semmi, vagy egyszerűen össze-vissza fog működni az alkalmazás. Hogy lehetne ezt helyesen kezelni?

Valójában úgy, hogy amikor elküldünk egy POST-ot akkor el kell végezni az átutalást, de nem szabad azonnal visszatérni a válasszal, hanem egy úgynevezett átirányítást kell végezni. Ilyenkor a böngésző átirányításra kerül egy eredmény oldalra, amely eredmény oldalt a böngésző a GET művelettel fogja majd lekérni. Ennek hatására megjelenik az oldal egy külön HTTP kérés eredményeképpen. Ha itt nyomjuk meg a frissítés gombot, vagy itt használjuk a böngészőnek a vissza gombját, akkor nem lesz ebből probléma, mert a frissítés hatására egyszerűen újra fogja küldeni a GET kérést és ekkor megjelenik újra az eredmény oldal. Ha pedig a vissza gombot nyomjuk meg, akkor az űrlap fog üresen megjelenni. Azt, hogy a POST után átirányítjuk a böngészőt "redirect after POST"-nak nevezzük, de más néven is elterjedt ez a módszer.

Web formátumai: HTML és CSS

A webet eredetileg arra találták ki, hogy tudósok különböző tudományos publikációkat tudjanak közreadni, és közöttük hivatkozásokat tudjanak elhelyezni. Ez volt a HyperText, a szövegben más szövegre történik a hivatkozás. Ennek a formátuma a HTML nyelv (HyperText Markup Language). Arra utal, hogy tartalmakat tárol és a tartalmak között hivatkozások vannak. A dokumentumokat biztonsági szempontokat szem előtt tartva, különböző szervereken akarták elhelyezni, hogy ne egy központi helyen történjen a tárolás, mert támadás, probléma esetén a tárhely megszűnhet a rajta lévő dokumentumok pedig eltűnhetnek. A dokumentumok és szerverek azonosítására az URL-t használták, ezeket lekérni HTTP protokollon lehet. A HTML a W3C által karbantartott szabványos formátum. Alapvetően egy szöveges formátum, de nemcsak a natív szöveget tartalmazza, hanem a szövegnek elrendezést is ad egy fa struktúrában. Egy HTML dokumentumot fel lehet bontani fejre és törzsre (head és body), a törzset is fel lehet bontani különböző paragrafusokra, illetve címekre, a paragrafusokon belül is különböző egyéb szövegrészeket lehet elhelyezni, mint például listákat vagy táblázatokat. Történelmi okokból a HTML állomány lehetővé tette, hogy formázást is meg lehessen adni benne, de manapság ez már nem javasolt, erre egy külön formátum, a CSS létezik.

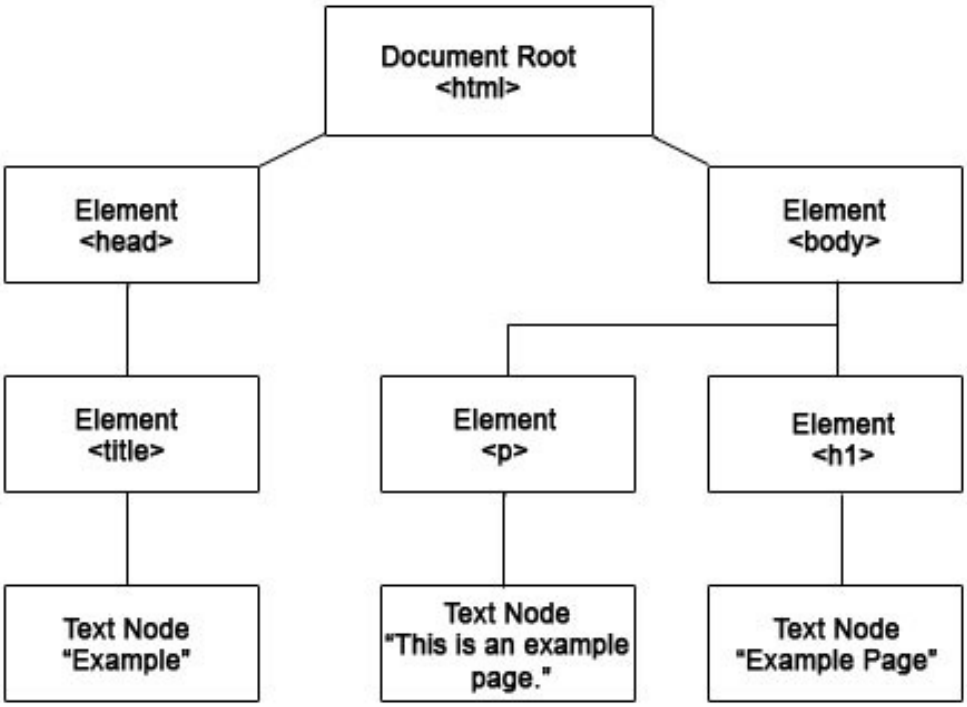
Nagyon fontos a HTML esetében a karakterkódolás. Alapvetően a számítógépek nem karaktereket, hanem számokat tárolnak, és minden egyes karakterhez egy-egy szám van hozzárendelve. Ezt maga a karakterkódolás, amikor megmondjuk, hogy melyik karakterhez milyen szám tartozik. Az angol ábécében persze nincs probléma. A problémák ott kezdődnek, ahol a különböző nemzetek speciális karaktereit kell megjelenítenünk, például a magyar ékezetes karaktereket. Gondoljunk bele, a korai verziókban egy karaktert 1 bájton ábrázoltak, így csak 256 különböző karaktert lehet ábrázolni. Megoldásként találták ki az UTF karakterkódolást. Ez egy olyan karakterkódolás, amellyel minden nemzet speciális karaktereit ábrázolni lehet. Ezért fontos, hogy ahol ékezetes karakterek is megjelennek, tipikusan a weben, ott az UTF karakterkódolást és ennek is egy speciális változatát, az UTF-8 karakterkódolást használjuk. Ez kevesebb helyen tudja tárolni az ékezetes karaktereket (angol ábécé betűit egy bájton, viszont a többit több bájton), mint egy klasszikus UTF-16 karakterkódolás (mely minden karaktert két bájton tárol). A HTML 5-ben már figyeltek erre, ennek megfelelően az alapértelmezett karakterkódolás az UTF-8. Korábbi verziókban

explicit módon kellett meghatározni, hogy az adott dokumentum milyen karakterkódolásban van.

Példa HTML dokumentum:

```
<!DOCTYPE html>
<html>
    <head>
        <title>Example</title>
    </head>
    <body>
        <h1>Example Page</h1>
        <p>This is an example page.</p>
    </body>
</html>
```

Egyszerű a HTML felépítése, fa struktúrában, tagekből épül fel. Jelölő nyelv, lehet benne szöveges tartalmat megadni, és lehet benne jelöléseket elhelyezni arra, hogy a szöveges rész hogyan struktúrált (pl. megadható cím, bekezdés, stb.). Ezeket a jelöléseket a tagekkel adjuk meg. A tageket relációs jelek közé írjuk, a HTML dokumentumot magát is úgynevezett `<html>` tagek közé kell zárni, ez a fa struktúrának a gyökere. Van egy nyitó és egy záró `<html>` tag. A záró a nyitótól egy perjellel tér el. Ez önmagában is struktúrát alkot, de ezen belül lehet további tageket elhelyezni. Két tag van közvetlenül a `<html>` tagen belül, `<head>` és `<body>`. A szöveg a tagek között szerepel, mintegy a tag értékét adja. Ezt a felhasználó is el tudja olvasni, míg a tagek a számítógépnek mondják meg, hogy mik ezek a szövegek, struktúrálisan hova tartoznak. A li tag például lista formátumot biztosít. A böngésző értelmezi a tageket, minden taghez alap megjelenési forma tartozik. Ez sokszor eléggé csúnya: fehér alapon fekete karakterek jelennek meg, alap betűtípussal (`<h1>`, `<p>` például más méretű). Ennek további formázására alkalmas a CSS, tehát nem itt a HTML-en belül formázzuk - itt csak a struktúra jelenik meg - és a szöveg stílusformázása nem. A HTML abból kifolyólag, hogy egyetlen gyökér tagnek kell lenni, a tagek között nem lehet átfedés, ill. egy tagen belül egy másik taget lehet elhelyezni, hierarchikusan fa struktúrába rendezett, ez grafikusan is jól ábrázolható.



DOM

CSS (Cascading Style Sheets) segítségével lehet a HTML dokumentumokat formázni, ez is egy W3C szabvány. A CSS-ben megadhatók (sok egyéb mellett): színek, méretek, elhelyezkedések, betűtípusok. Fontos hangsúlyozni, a tartalomnak és a formának el kell válnia. A tartalmat és a struktúrát a HTML, a formát pedig a CSS közvetítse (hasonlóan a Word-höz, ahol van a szöveg, és stílusokat lehet hozzá adni). Azért is hasznos, ha CSS-t alkalmazunk, mert ha át szeretnénk formázni a dokumentumot, újraformázni a weblapunkat, akkor nem kell a HTML struktúrát és tartalmat módosítanunk, elegendő csak a CSS-t változtatnunk. (pl.: más színnel jelenjen meg a címsor) A CSS-ben meg kell mondanunk, hogy a HTML oldalon belül mi az, aminek szeretnénk változtatni a formátumát. Legegyszerűbben ezt úgy adhatjuk meg, hogy megadjuk a tag nevét (pl.: `h1`) Nemcsak a tageket lehet megadni, hanem nagyon bonyolult leírási módok is vannak a HTML elemekre, ezeket a leírási módokat nevezzük CSS selectoroknak.

```
h1 {
    text-align: center;
    color: red;
    font-size: large;
    font-family: "Times New Roman", serif;
    font-style: italic;
}
```

A CSS tartalom megadható magában a HTML-en belül beágyazva, a header részben, és akár külön állományban is.

Beágyazott CSS: a HTML-en, és ezen is belül egy tagen belül attribútumként (kulcs-érték pár) való meghatározás. Ez nem ajánlott mód, itt pont azt a lehetőséget bukjuk el, hogy ha változtatni akarjuk a formátumot, ne kelljen belenyúlni a HTML állományba.

```
<h1 style="color: red;">A CSS használata</h1>
```

Jobb megoldás, ha a CSS-t a HTML dokumentum fejlécében helyezzük el. Ha változtatni akarunk, nem kell módosítani a törzset, csupán a fejlécben elhelyezett CSS-t kell átírni.

```
<html>
    <head>
        <style>
            h1 {
                color: red;
            }
        </style>
    </head>
    <body>
        <h1>A CSS használata</h1>
    </body>
</html>
```

Azonban, ha több HTML állományunk van - és természetesen egy weboldal akár több ezer HTML oldalból is állhat - akkor mindegyiknek a fejlécébe ezt be kell tenni, ami ismételt hálózati átvitelt igényel. Érdemes ezért kiszervezni egy külön CSS állományba, amelyekre tipikusan a css kiterjesztést szoktuk alkalmazni és ezek külön URL-en érhetők el, a HTML oldalból pedig egy link taggel tudunk hivatkozni erre a CSS állományra. Sokkal jobb megoldás ez, mert külön van a HTML és a CSS, ezeket külön tudjuk változtatni, illetve nem ismétlődik a CSS, hiszen minden egyes HTML állományunk elejére be tudjuk tenni a CSS fájlra vonatkozó hivatkozást.

```
<html>
    <head>
        <link rel="stylesheet" href="styles.css" />
    </head>
    <body>
        <h1>A CSS használata</h1>
    </body>
</html>
```

Webes alkalmazas RIA felülettel

A klasszikus webes alkalmazások HTML állományokból álltak, ezeket tudtuk megnézni, és ezek között tudtunk navigálni. Oldalról oldalra lehetett lépkedni, az oldalakat viszont linkek kapcsolták össze. Ha ezek a HTML állományok mindig ugyanazok, akkor statikus HTML állományról, ha tartalmuk változik akkor dinamikus HTML állományról beszélhetünk. A dinamikus tartalmat a szerver állítja elő amikor feltesszük a kérdést és a válasz HTML dinamikusan áll össze. Időben visszatekintve nagy szakadék volt az asztali (desktop)

alkalmazások, - melyek bonyolult felhasználói interfésszel rendelkeztek (nevezhetjük ezeket vastag kliensnek is) - és a webes alkalmazások között. Ez utóbbiak esetén az oldalak közötti lépkedésen kívül nem sok mindent lehetett csinálni. Régen el sem tudtuk volna képzelni azt, hogy a böngészőben egy világtérkép jelenjen meg, vagy akár egy levelező szoftver. Erre szolgáltak az asztali alkalmazások. Ezek sokkal intuitívabbak voltak, sokkal kényelmesebben lehetett őket használni. Azonban a technológia fejlődésével a kettő elkezdett egymáshoz közeledni, pontosabban a webes alkalmazásokkal egyre több és több dolgot lehetett megoldani, és ezt az új technológiák tették lehetővé. Azokat aböngészőben elérhető webes alkalmazásokat, melyek csaknem olyan gazdag funkcionalitást tudnak nyújtani, mint egy asztali alkalmazás, RIA (Rich Internet Application) alkalmazásoknak nevezzük. Az első implementációk között vannak a Google egyes termékei (például Maps, GMail). Nemcsak egy asztali alkalmazással (például Microsoft Outlook) lehet levelezni, hanem egy böngészőben is teljes értékű levelező klienst lehet használni. Lényegében ugyanazt a funkcionalitást el lehet érni, mint egy desktop alkalmazás esetében. A háttérben a HTML, CSS és a JavaScript technológiák fejlődése áll. Ezekben létrehoztak olyan eszközöket, lehetőségeket, amivel most már ilyen bonyolult alkalmazásokat is lehet böngészőben implementálni. Különböző technológiák is megjelentek még, csak az említés kedvéért itt van néhány: Adobe Flash (játékok, videó lejátszás), Microsoft Silverlight, Oracle JavaFX, de ezek igazából fel sem futottak. Közös hibájuk volt, hogy a böngészőn kívül külön kellett őket telepíteni, illetve nem voltak túl kiforrottak. Így a HTML, CSS, JavaScript hármasa, mára ezt a terepet biztosan uralja. Biztonsági problémák is akadtak ezekkel a technológiákkal. A JavaScript esetén is nagyon sok biztonsági megoldásra van szükség - hiszen gondoljunk bele abba, hogy nem szeretnénk, ha a böngészőnk csak úgy ad-hoc módon elkezdjen hozzáférni a számítógépünk erőforrásaihoz, fájlokat kezdjen olvasgatni, letörölje a winchesterünk tartalmát, bekapcsolja a webkameránk, elkezdjen nyomtatni. Ezek a programok úgynevezett sandboxban (homokozó, amelynek pereme van, tehát nem lehet belőle kimászni) futnak, akár egy JavaScript alkalmazásról, akár bármelyik más RIA eszközről beszélünk, mint például az Adobe Flash. Ezek nem tudják elérni a számítógépünk perifériáit, nekünk kell erre engedélyt adnunk. Ha például böngészőből akarunk videókonferencia eszközt (webkamera, mikrofon) használni, akkor a böngésző meg fogja kérdezni adunk-e engedélyt a böngészőnek, hogy hozzáférjen az eszközünkhöz.

A SPA (Single Page Application) egy irányzat, a webes alkalmazásokat úgy igyekeznek felépíteni, hogy egyetlen oldalból álljon és ezen az oldalon belül a különböző részletek változzanak. A GMail-t is mondhatjuk SPA-nak, hiszen egy oldalon maradunk mindig, csak az adott kis részletek változnak (pl.: levéllista, belekattintás után adott levél, bal oldalt és felül marad a menü struktúra, stb.). Technológiailag ezek komoly kihívást jelentenek, első betöltődésük kicsit lassabb. Vannak technológiák a gyorsításra, de még mindig azt látjuk például az említett alkalmazásnál, hogy az elején picit homokórázik, és várnunk kell mire betöltődik. Nagyon nehéz az ilyen oldalakat keresőbaráttá tenni, hogy a Google beindexelje ezeket az oldalakat, és könnyebben lehessen rájuk találni. Bonyolultabb lehet a böngésző gombok kezelését megoldani (back, forward, refresh). Ezeket használva rossz esetben elromolhat az alkalmazás, jobb esetben csak visszaugrik az elejére, mintha csak most kezdtük volna el használni az alkalmazást. A felhasználók nyomonkövetése (mit csinál, milyen oldalt kérdezett le) plusz erőforrásokat igényelnek a fejlesztők oldaláról. Van

kényelmes oldala is: könnyű használni, nem kell várni mindig az új oldal betöltődésre, viszont van hátránya is: implementálásuk sokkal nehézkesebb.

Nagyon gyakori az, hogy a funkcionalitás, illetve a felhasználói felület különválik, ugyanis más technológiák, más programozási nyelvek, más eszközök alkalmasak arra, hogy egy felhasználói felületet írjunk. Manapság webes alkalmazásokhoz felületének megvalósítására a HTML, CSS, JavaScript való. Sokszor más programozási nyelvben érdemes megírni, megvalósítani az üzleti logikát, az alkalmazás funkcióit, a különböző üzleti folyamatokat. Erre persze több programozási nyelv is használható (Java, Python, stb. Szakmai elnevezés, hogy az alkalmazás felületének megjelenítését a frontend végzi, az üzleti logika futtatását pedig a backend.

Ahhoz, hogy RIA alkalmazásokat tudjunk implementálni, nagyon fontos, hogy a HTML és a CSS mellett erősen kibővült a JavaScript technológia. Ez kezdetben csak egy böngészőben futó programozási nyelv volt, szkript nyelv és a böngésző futtatta, ráadásul egy sandboxon belül, így nem is volt meg minden joga, nem tudott például fájlokat beolvasni. Manapság már sokkal elterjedtebb, egy általános célú nyelv, és már backendet is lehet benne implementálni. Ugyanakkor még mindig az első számú technológia arra, hogy a böngészőben dinamikus működést implementáljunk. Azaz egy oldalon belül történjen valami, ne kelljen átmenni egy másik oldalra ahhoz, hogy egy másik tartalmat lássunk.

A Javanak azonban semmi köze a JavaScripthez. A háttérben csak egy marketing együttműködés van, és ezért utalnak egymásra nevükben, de a Java és a JavaScript két teljesen különböző programozási nyelv, nagyon eltérő adottságokkal és felhasználási móddal.

A CSS-hez hasonlóan nem célszerű a HTML-t és a JavaScriptet keverni. Lehet beágyazott a HTML fejlécébe, de tanácsosabb kiszervezni egy külön állományba a JavaScript kódot, ebben az esetben külön tudjuk módosítani, illetve újra tudjuk használni.

Kezdetben a JavaScriptet arra találták ki, hogy az adott oldalon belül mindenféle egyszerű műveleteket lehessen végezni (pl.: felugró ablak, vagy validáció, azaz a felhasználó által beírt értékek ellenőrzésére). Ez a kis szkript, programocska csak a böngészőben futott, és ott végzett különböző műveleteket. Olyat is tudott, hogy egy HTML oldalnak megváltoztatta a tartalmát. Gyakorlatilag az történt a háttérben, hogy a HTML fát módosította, és ahogy módosította, a böngésző ezt észrevette és máshogy jelenítette meg az adott HTML oldalt. Dinamikussá tudtuk tenni tehát az oldalunkat, a HTML oldalon belül tudtunk működést implementálni. (Pl. egy elemet megjeleníteni vagy eltüntetni, a legördülő menük tartalmát módosítani, stb.)

HTML oldalba ágyazott JavaScript kód:

```
<input type="button" onclick="alert('Hello World!');"
    value="Click me!" />
```

HTML oldal fejlécébe ágyazott JavaScript kód:

```
<head>
    <script type="text/javascript">
```

```
        onload = function() {
            document.getElementById('mybutton')
                .addEventListener('click',
                    function(e) {
                        alert('Hello World!');
                    });
        }
    </script>
</head>
```

JavaScript külön állományban:

```
<!DOCTYPE html>
<html>
<head>
    <script src="myscript.js" type="text/javascript"></script>
</head>
<body>
    <input id="mybutton" type="button" value="Click me!" />
</body
</html>
```

A `myscript.js` állomány tartalma:

```
onload = function() {
    document
        .getElementById('mybutton')
        .addEventListener('click',
            function(e) {
                alert('Hello World!');
            });
}
```

Az AJAX technológia (Asynchronous JavaScript and XML) annyival több, hogy JavaScripttel fel lehet venni a kapcsolatot a szerver oldallal. Azaz tudunk a szerverrel kommunikálni oldal váltás nélkül. Erre klasszikus példa a Google kereső (legördülő menüben felajánlott találati lista a keresőszóhoz). Ezek a felugró lehetőségek nem a HTML oldalba vannak beágyazva, hanem minden egyes esetben, amikor lenyomunk egy billentyűt, akkor egy JavaScript kód lefut, felveszi a kapcsolatot a szerver oldallal, onnan letölti az adatokat, a lehetőségeket, javaslatokat, amiket a logika gondol, és utána a JavaScript kiteszi a legördülő menüt, módosítva a HTML állománynak a tartalmát, valójában a fa struktúrát (DOM fa). Ezzel komolyabb felhasználói élmény alakítható ki, mert nem kell oldalt váltanunk, nem kell várnunk a következő oldal betöltésére. Mire szokták még ezt használni? Klasszikus módon az egymástól függő legördülő menük esetén, mint például ország-város összekapcsolás. Nem kell az összes országot, várost tudni a HTML-ben. Adott ország kiválasztása után egy HTTP-kérésben AJAX-szal a JavaScript a szerverhez fordul, onnan letölti a városok listáját, és megjeleníti a városokat legördülő menüben. Így működik az összes RIA alkalmazás (pl.: a Google Maps használatakor, ahogy görgetünk, dinamikusan, mozaikszerűen jelennek meg a térkép darabkák). Azért is jó az AJAX, mert nem kell mindig az adott oldalt újra és újra letölteni, ezzel a hálózat kihasználtsága is javul, kevesebb adatot kell átvinni a hálózaton, ha

mindig csak a változó tartalmat, az oldal változó tartalmát kell letölteni. Az AJAX-szal manapság már nem XML formátumot, hanem JSON formátumot szoktunk használni adatátvitelre. Kezdetben még csak HTML darabkákat, de ma már csak az adatot visszük át. Ez aszinkron történik, nem várjuk meg amíg lejön az adat, hanem gyakorlatilag a böngésző ugyanúgy továbbra is használható marad, nem fagy le amíg a szerver választ nem ad. Amikor megérkezik az adat, akkor lesz frissítve a képernyő. Elég sokszor láthatunk ilyet, kattintunk valahova és nem történink semmi, majd hirtelen egy kis idő múlva megváltozik a felület. Ekkor AJAX van a háttérben és aszinkron módon történt a hívás.

HTTP esetén mindig a kliensnek kell kommunikációt kezdeményezni, nincs olyan, hogy a szerver úgy dönt, most leküld valamit a böngészőnek, hanem minden esetben a kliensnek kell kérnie az adott adatot: nem lenne túl jó, ha a böngésző állandóan a különböző szerverek üzeneteivel bombázna minket. Bizonyos esetekben viszont igény van arra, hogy ha történik a szerver oldalon valami, az megjelenjen a kliensnél anélkül, hogy a kliens ezt kérné. (például böngészőben futó chat alkalmazás esetén, ha a chat partnerem ír, akkor az jelenjen meg, tehát nem én kérdezem meg hogy van-e üzenet hanem ez az üzenet automatikusan jelenjen meg nálam. A böngészőt a szervernek kell tájékoztatni a változásról, de a HTTP erre alapvetően alkalmatlan. Kezdetben kerülő megoldásokat alakítottak ki, vegyük sorba őket. Polling, ami úgy működik, hogy a kliens bizonyos időközönként rákérdez, hogy kapott-e üzenetet, azaz nem a szerver küldi le a kliensnek az üzenetet, hanem a kliens kérdezget (polling), hogy van-e új üzenet. Ennek az eljárásnak már egy magasabb szintjét képviseli a long polling. A kliens kérdezi, hogy jött-e üzenet, melyre a szerver nem válaszol, csak amikor jött üzenet. Nem sok kis kéréssel bombázza a kliens a szervert, hanem hosszú kéréssel. WebSockets: a kliens és a szerver között, a HTTP protokoll átvált egy WebSocket protokollra, ami kétirányú (full duplex) kommunikációt tesz lehetővé a kliens és a szerver között. Olyan, mint egy TCP/IP - felépül egy kapcsolat és mind a két fél tud a csatornán üzenetet küldeni a másiknak - viszont ez a TCP/IP fölött helyezkedik el, egy magasabb rétegen. Server sent events: a szerver egy nyitott HTTP kérésen belül képes adatokat leküldeni a kliensnek. A kliens nem fejezi be a kommunikációt, hanem tovább vár újabb adatszeletekre a nyitott HTTP válaszban.

REST webszolgáltatások

Manapság a webesalkalmazásokat legalább két részre bontják. Az egyik az üzleti logikát végző backend, amely felveszi a kapcsolatot az adatbázissal is, illetve a felhasználói felületet megjelenítendő frontend.

Közöttük a kommunikáció nélkülözhetetlen. A leggyakoribb megoldás a frontend és a backend között kialakított RESTful webszolgáltatás. Szögezzük le, hogy a REST egy architektúrális elv, más szóval minta/pattern, és RESTful az a webszolgáltatás, amely ezt implementálja. A W3C általános definíciója szerint a webszolgáltatás hálózaton keresztüli gép-gép együttműködést támogató szoftver rendszer. Kevésbé általánosan megfogalmazva a webszolgáltatás nem más, mint egy technológia, amivel két független számítógép két szoftvere fel tudja venni egymással a kapcsolatot webes környezetben. Legtöbb esetben platformfüggetlenek a webszolgáltatások, ennek köszönhetően eltérő nyelven írt szoftverek is képesek kommunikálni egymással. A kommunikációban hangsúlyos a protokoll, azaz

hogyan megy az adat az egyik szoftverből a másikba, és az adat formátuma, mint például XML (Extended Markup Language), JSON (JavaScript Object Notation). A webszolgáltatások is kliens-szerver módban üzemelnek. Van egy kitüntetett alkalmazás szoftverrendszer, amelyik szerverként működik, ez biztosítja a szolgáltatást, amit a másik, a kliens, igénybe szeretne venni.

Bár a webszolgáltatásoknak több fajtája is van, mégis igazán csak két elterjedt típust különböztetünk meg, a RESTful, illetve a SOAP szolgáltatásokat. A SOAP szolgáltatások ugyan régóta jelen vannak, de lassan túlhaladottak lesznek. Ezzel szemben a REST webszolgáltatások modernebbek és könnyebben használhatóak. A REST (Representation State Transfer) webszolgáltatás fogalmát Roy Fielding találta ki doktori disszertációjában. Roy Fielding elég jelentős szerepet vállalt a HTTP protokoll kidolgozásában, és erre építve dolgozott ki egy kommunikációs formát arra, ahogy az alkalmazások egymással tudnak kommunikálni (REST). Minden alkalmazásra úgy kell gondolni, mint erőforrások gyűjteményére.Az erőforrások megcímezhetők, és rajtuk CRUD (Create, Read, Update, Delete) műveletek definiálhatók. Példaként nézzünk egy banki alkalmazást, ahol az ügyfelek, a hozzájuk tartozó számlák és bankkártyák tekinthetők erőforrásnak. Ezek ugyan üzleti fogalmak, de jól látszik a velük való műveletek elvégzésének (bankszámla nyitás, módosítás, megszüntetés, egyenleg lekérés) napi gyakorlata. A RESTful webszolgáltatások HTTP protokollra építenek, az átviteli formátum pedig a legtöbb esetben a JSON. A RESTful webszolgáltatásoknál az API leírására a WADL (Web Application Description Language) technológiát használták régen, de nem terjedt el igazán. Helyette az OpenAPI - korábbi nevén Swagger API – vált használatossá. Ennek az a feladata, hogy a RESTful szolgáltatásokat valamiféle nyelven szabványos módon le tudjuk írni. A RESTful webszolgáltatások térhódítása az AJAX megjelenésére, illetve a frontend és backend közti kommunikációs igényre vezethető vissza, mert erre a leginkább kézenfekvő a RESTful webszolgáltatások alkalmazása volt.

A RESTful webszolgáltatások esetén a HTTP protokoll szinte minden tulajdonságát kihasználjuk. Erőforrások egyedi címezésére az URL, műveletek elvégezésére a HTTP metódusok (GET, POST, PUT, DELETE) alkalmasak.

Tekintsük át egy klasszikus webes alkalmazás (RIA = Rich Internet Application alkalmazás) működését. Az alkalmazás először letölti a fő oldalhoz szükséges HTML, CSS és JavaScript állományokat, melyek alapján rendereli (megjeleníti) a böngésző a felületet. Felolvassa tehát a HTML állományt, felépíti a DOM struktúrát, a CSS alapján ezeket megformázza, valamint lefuttatja a különböző JavaScript programokat, egyszóval grafikusan megjeleníti az oldalt. Abban az esetben, ha a felhasználó például kitölt egy űrlapot, majd megnyomja a submit gombot (akármi legyen is a konkrét neve), akkor egy JavaScript fut le; ez AJAX-al felküldi egy GET HTTP aszinkron kérésben az adatokat a szervernek. Az aszinkronitás azt jelenti, hogy a JavaScript nem áll meg addig, míg választ kap, hanem végzi tovább a működését. A szerver erre valamikor válaszol valamit például JSON formátumban egy egyszerű HTTP response törzsében; ezt a JavaScript fogadja és a válasz alapján módosítja az adott weboldal tartalmát (pl.: kiír egy üzenetet, hogy az adott értékek sikeresen elmentésre kerültek). Látható, hogy nem történt a háttérben oldalváltás, nem változott a cím, nem villant a teljes böngésző oldal, csak az adott oldalon belül változott egy picit a tartalom - ez az AJAX lényege. A háttérben valójában volt egy HTTP kérés, de ezt a JavaScript tette fel és

küldte át a kérést a szerver oldalra, majd erre a szerver oldal válaszolt. Fontos kiemelni, hogy nem a teljes HTML tartalom ment át, hanem csak az az adat, amire épp szüksége volt a szervernek vagy a kliensnek, ezáltal még hálózati forgalmat is spóroltunk.

Az OpenAPI/ Swagger UI szolgál RESTful webszolgáltatások definiálására. Ez egy olyan eszköz, amivel nemcsak dokumentálni lehet a RESTful webszolgáltatásokat, hanem még kipróbálni is, tehát egy úgynevezett aktív dokumentációként szerepel. Az OpenAPI egy programozási nyelvtől és keretrendszerektől független technológia. Ennek segítségével meg tudok hívni egy JAVA programból például egy C++ programot. A protokoll RESTful webszolgáltatások esetén a HTTP protokoll, és az üzenetformátum JSON. A JSON-t az XML leváltására találták ki, és számos előnye van az XML-lel szemben, bár ez is szöveges formátum, amit az ember és a számítógép egyaránt olvasni, értelmezni tud. Főként RESTful webszolgáltatásnál használatos, amikor backend és frontend egymással beszélget.

Példa JSON dokumentum:

```
{
  "questions": [
        {
            "text": "Válassza ki az alábbiak közül a helyes állítást!",
            "answers": [
                {
                    "point": 0,
                    "text": "Micimackó szereti a dinnyét."
                },
                {
                    "point": 1,
                    "text": "Micimackó szereti a mézet."
                }
            ],
        }
    ]
}
```

Hogyan is épül fel egy JSON dokumentum? Kapcsos zárójelek között úgynevezett objektumok vannak. Az objektumok mezőkből állnak, amik pedig kulcs érték párok. A kulcs az értéktől kettősponttal van elválasztva. Ezen kívül még tömböket is lehet definiálni, amik arra valók, hogy több értéket is lehessen megadni. A tömböt szögletes zárójel határolja, az elemeket vessző választja el egymástól. Az objektumok és a tömbök egymásba ágyazhatók. A beágyazás miatt fastruktúrát lehet leírni a JSON dokumentummal, ugyanúgy, mint egy HTML, vagy XML esetén.

Ez JavaScripttel a következő módon kérdezhető le:

```
data['questions'][0]['text']

data['questions'][0]['answers'][0]['text']
```

JSON dokumentum leírására a JSON Schema-t használják. A JSON Schema önmaga is egy JSON dokumentum.

```
{
  "$schema": "https://json-schema.org/draft/2019-09/schema",
  "title": "Employees",
  "type": "array",
  "items": [
    {
      "title": "EmployeeDto",
      "type": "object",
      "required": ["name", "id"],
      "properties": {
        "id": {
          "type": "integer",
          "description": "id of the employee",
          "format": "int64",
          "example": 12
        },
        "name": {
          "type": "string",
          "description": "name of the employee",
          "example": "John Doe"
}}}]}
```

Amennyiben RESTful webszolgáltatásokat szeretnénk tesztelni, a Postman egy kiváló és elterjedt eszköz erre. Feltelepíthető a saját számítógépünkre, de szolgáltatásként is igénybe vehető. Két gépről használva a két gép között a beállításokat átviszi. Különböző operációs rendszereken is használható, illetve gyakorlatilag a RESTful webszolgáltatások teljes fejlesztési életciklusát támogatja. Nagy segítséget nyújt a manuális és az automata tesztelésben, valamint üzemeltetésben, monitorozásban, de a tervezésben, fejlesztésben is hasznos. Emellett magas szintű csoportmunkát is lehetővé tesz, egyszerre akár többen is tudnak ugyanazon a projekten dolgozni, sőt verziókezelés is tartalmaz.

Szerver alkalmazás webszolgáltatás interfésszel

Bár alkalmazások közti kommunikációra manapság többnyire a RESTful webszolgáltatásokat alkalmazzák, azonban elterjedt még a SOAP-os (Simple Object Access Protocol) webszolgáltatás használata is. Régebben kizárólag SOAP-os webszolgáltatások léteztek, és régebbi rendszerekben, különösen olyan rendszerekben, amelyek nem képesek gyorsan fejlődni – főként a magas biztonsági és megbízhatósági elvárások miatt -, még mindig a SOAP-os webszolgáltatásokat használják (pl.: régebbi banki rendszerek).

A webszolgáltatás itt is azt jelenti, hogy olyan szoftverrendszerről beszélünk, ami lehetővé teszi, hogy két alkalmazás egymással kommunikálni tudjon. A RESTful webszolgáltatásoknál a kizárólagos protokoll a HTTP. Habár SOAP webszolgáltatások esetén a protokoll választható, a leggyakoribb mégis a HTTP. Az átviteli adat formátuma itt nem a JSON, hanem az XML. Míg a RESTful webszolgáltatásoknál választható a formátum, de leggyakrabban JSON, addig a SOAP-os webszolgáltatásoknál kizárólag XML-t használhatunk.

A SOAP-os webszolgáltatásoknál szinte már a kezdetektől létezik az interfészt leíró dokumentum, tehát az API-t leíró dokumentum, és ez nem más mint a WSDL dokumentum.

Az XML (Extensible Markup Language) formátuma szintén a W3C szabványügyi szervezet alá tartozik. Ez egy általános célú leíró nyelv elsősorban az adatszerkezet leírására, mely kiterjeszthető és jelölő is egyben. Szöveges formátumú - ember és számítógép által is könnyen értelmezhető, illetve más formátumokhoz hasonlóan ez is fastruktúrát ír le.

Példa XML állomány:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!-- Catalog of books -->
<catalog>
    <book isbn10="059610149X">
        <title>Java and XML</title>
        <available />
    </book>
    <book isbn10="1590597060">
        <title>Pro XML Development with Java Technology</title>
    </book>
</catalog>
```

Nagyon hasonlít a HTML nyelvre, de persze vannak jelentős eltérések is közöttük. A hasonlóság hátterében az áll, hogy mind az XML, mind a HTML formátumnak ugyanaz az őse, egy SGML dokumentum formátum. Az XML és a HTML egyaránt ennek az egyszerűsített formája. A fő különbség a két formátum közt, hogy a HTML sokkal megengedőbb, nincsenek olyan kötött formai szabályai, mint az XML-nek. Az XML dokumentum adatok tárolására, adatok átvitelére való, ezért használjuk webszolgáltatásoknál is. Jelölő nyelv, azaz itt is tagek szerepelnek, a tagen belül pedig a kulcs-érték formában megjelenő attribútumok helyezkednek el. Egy gyökér elemnek mindenképpen lennie kell. Nem lehetnek átnyúló, egymást átfedő tagek. Egy tag csak egy másik tagen belül helyezkedhet el. Behúzást és sortörést alkalmazunk az XML-ben, ami szépen kirajzolja, megjeleníti a fastruktúrát, de ez csak humán "fogyasztásra" való, a számítógép enélkül is értelmezni tudja. A tagek opcionálisak is lehetnek. Önlezáró tagnek nevezzük azt, amelyikben nincs szöveg. Ezek a tulajdonságok teszik lehetővé, hogy egy könnyen olvasható, hierarchikus adatszerkezetet képes tárolni. Egy XML dokumentum első sorában szerepel az XML pragma, amely néhány általános információt ad meg (például verziószám, karakterkódolás, megjegyzés, stb.). A szöveges rész maga az adat tartalom, a tagek pedig az XML dokumentum struktúráját adják meg.

Több XML szerkesztő alkalmazás is létezik a piacon. Vannak ingyenesen (Notepad++) használható és kereskedelmi (Altova XMLSpy) termékek is. Az XML szerkesztők azért jobbak egy szövegszerkesztőhöz képest, mert ellenőrizni tudják, hogy betartottuk-e az XML szabályokat. Adott szakterületen használt XML szerkezetét (szabályait) az XML Schema írja le.

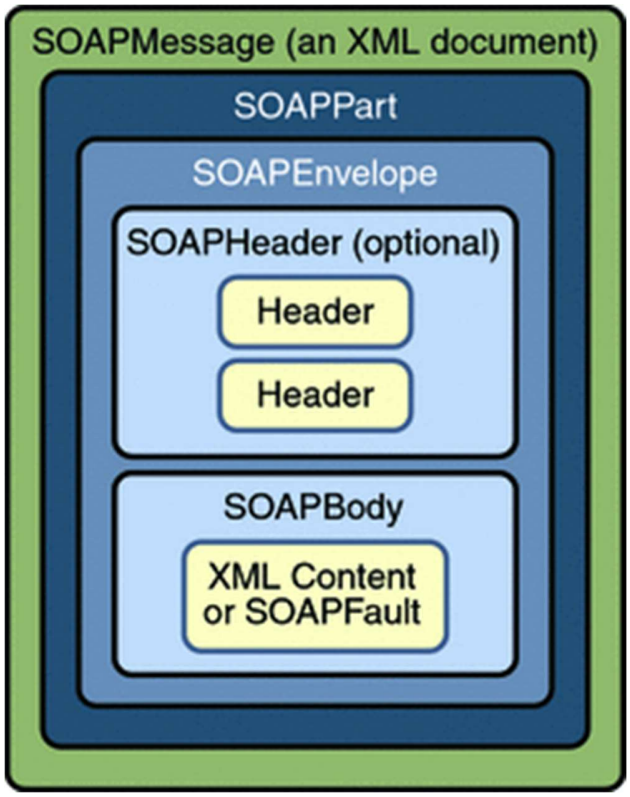
Az XML kiterjeszthető, azaz a tageket mi magunk adhatjuk meg. Ezzel kapcsolatban merült fel az az igény, hogy egy adott XML dokumentumon belül használható tagek rögzítésre is kerüljenek. Erre megoldásként az XML Schema szolgál, ami önmagában is egy XML

dokumentum, és leírja, hogy egy másik XML dokumentumban milyen tageket lehet használni, de a tagek mellett meghatározza az attribútumokat - melyik tag értékének milyen típusúnak kell lennie (pl. string, decimal, float, date), valamint, hogy mire való egy adott tag. Egy tagen belül egy másmilyen tag akár többször is szerepelhet, de attribútumból kizárólag egy lehet.

Példa XML séma:

```
<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema">
  <xs:element name="catalog">
    <xs:annotation>
      <xs:documentation>Catalog of books</xs:documentation>
    </xs:annotation>
    <xs:complexType>
      <xs:sequence>
        <xs:element name="book" minOccurs="0" maxOccurs="unbounded">
          <xs:complexType>
            <xs:sequence>
              <xs:element type="xs:string" name="title"/>
              <xs:element type="xs:string" name="available" minOccurs="0"/>
            </xs:sequence>
            <xs:attribute type="xs:string" name="isbn10" use="optional"/>
          </xs:complexType>
        </xs:element>
      </xs:sequence>
    </xs:complexType>
  </xs:element>
</xs:schema>
```

A SOAP olyan XML dokumentum, amely két nagyobb részből áll (fej és törzs), és szintén egy W3C szabvány. A SOAP kéréseket, azaz XML dokumentumokat, nem kell kézzel írni, szerkeszteni, ezt a szoftverek automatikusan állítják össze, de értelmezhetők, olvashatók emberi szemmel is. A SOAP felépítése szigorú szabályok szerint történik, formája a boríték (Envelope), melyen belül további részek a header és a body. Az egyes rendszerek közötti adat átvitel összetett. Az egyik rendszer elküld egy HTTP kérést, melynek törzsébe teszi be a SOAP kérést; majd a másik rendszer válaszként, a HTTP válaszban, ennek törzsébe illeszti a SOAP választ, azaz egy XML dokumentumot. A két rendszer tehát így kommunikál egymással, az egyik HTTP-n küld egy SOAP XML-t, a másik pedig válaszol egy SOAP XML-lel.



SOAP boríték

Példa SOAP kérés:

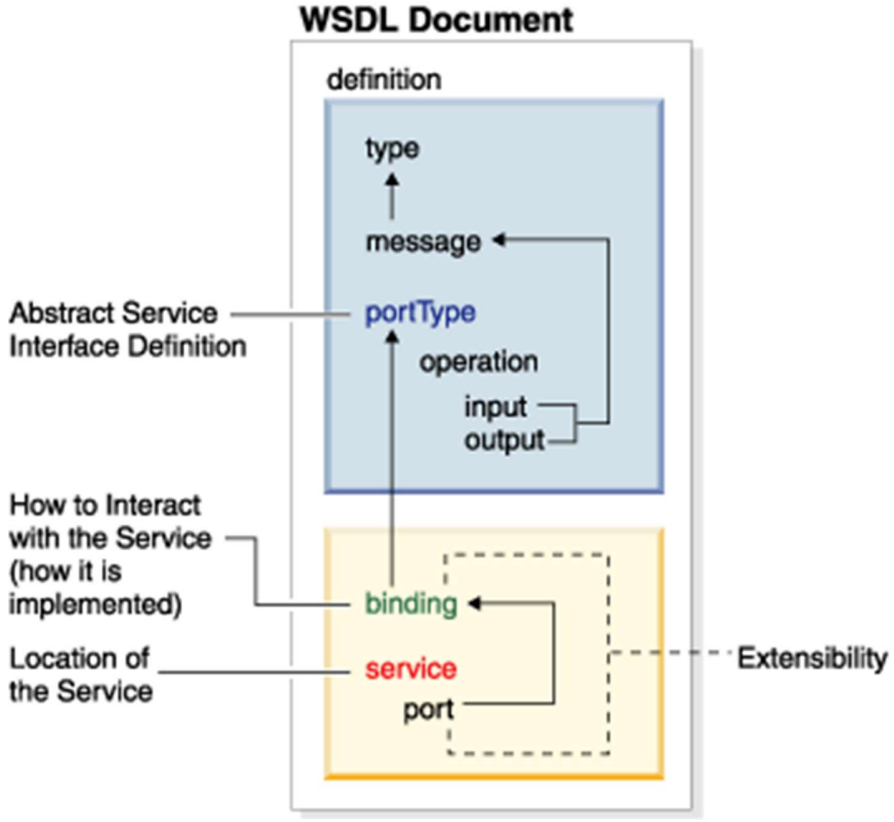
```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Header>
    <headers:RequestId xmlns:headers="http://www.example.org/stock/headers"
         env:mustUnderstand="0">93e12b33-6511-4bf4-9310-
69cd13e60b44</headers:RequestId>
  </soap:Header>
  <soap:Body>
    <m:GetStockPrice xmlns:m="http://www.example.org/stock">
      <m:StockName>IBM</m:StockName>
    </m:GetStockPrice>
  </soap:Body>
</soap:Envelope>
```

Példa SOAP válasz:

```
<?xml version="1.0"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  <soap:Body>
    <m:GetStockPriceResult xmlns:m="http://www.example.org/stock">
      <m:StockName>IBM</m:StockName>
      <m:StockPrice>13 $</m:StockPrice>
    </m:GetStockPriceResult>
```

```
  </soap:Body>
</soap:Envelope>
```

A SOAP webszolgáltatásoknál is van interfész leíró, ez a WSDL dokumentum. Nem meglepő módon ez is XML formátumban van, és kiterjedten használja az XSD-t (XML Schema Definition) - ez az ami az API-t leírja. Egy WSDL dokumentum felépítése nagyon bonyolult, értelmezésében különböző eszközök segíthenek.



WSDL

Két rendszer közti kommunikáció kialakítása számos módon történhet. Tekintsük át a legfontosabb megközelítéseket. Top down megközelítés: először is leírjuk, hogyan kommunikáljanak a rendszerek egymással, majd utána úgy módosítjuk a rendszereket, hogy megfeleljenek ennek a leírásnak. Bottom up megközelítés: a már létező rendszer leírása, azaz a kész rendszer alapján készül a leírás. Meet in the middle megközelítés: WSDL is van, rendszer is van, és a rendszert úgy próbáljuk alakítani, hogy megfeleljen az adott WSDL dokumentumnak.

Klasszikus webszolgáltatás hívás menete: Az egyik rendszer összerakja a SOAP borítékot, egy WSDL dokumentum alapján; ezt az XML dokumentumot HTTP-n keresztül átküldi a szervernek; a szerver ezt fogadja, feldolgozza; majd előállítja a válasz üzenetet, mely szintén egy XML SOAP dokumentum; majd ezt beleírja a HTTP válasz törzsébe; a kliens ezt megkapja, kiolvassa és értelmezi.

A SOAP webszolgáltatások tesztelésére, kipróbálására létezik egy igen elterjedt eszköz, a SoapUI. Ingyenes és fizetős változata is van, de már az ingyenessel is sok minden kivitelezhető (új projekt, kérés, stb.). A SoapUI képes arra, hogy a WSDL alapján legenerálja az XML dokumentumokat, amit el lehet küldeni a szervernek, majd a SoapUI megmutatja az visszakapott választ is.

Szerver alkalmazas webszolgáltatás interfesszel (gyakorlat)

A SoapUI letölthető a https://www.soapui.org/downloads/soapui/ címen. A SoapUI Open Source verziót töltsd le! Indítsd el a telepítőt, és kövesd az utasításokat!

Aszinkron üzenetküldés

A HTTP protokoll szinkron protokoll, ami azt jelenti, hogy a kliensnek meg kell várnia, amíg a szerver választ nem ad, addig nem folytathatja tovább a tevékenységét, amíg a HTTP válasz vissza nem érkezik. Legtöbb esetben ezredmásodpercek alatt visszajön a szerver válasza, de adódhatnak problémák is szinkron kommunikáció kapcsán. Lehet, hogy a hívott fél megbízhatatlan, így nem tud kiszolgálni kéréseket. Nem áll rendelkezésre, túlterheltség vagy verzió frissítés miatt, esetleg csak bizonyos időszakban működik. Előfordulhat a kommunikációs csatorna megbízhatatlansága is. A két rendszer közti hálózat lassú vagy nem elérhető, esetleg szétesik. Gondot jelent az is, ha a másik rendszer egyszerűen csak nagyon lassan – percek vagy akár órák alatt - válaszol. A webalkalmazás horizontális skálázás is probléma forrás, mert ha több számítógépen fut az alkalmazás, külön feladat a beérkező kérések elosztása. Probléma adódhat, ha két rendszer között nem találunk olyan protokollt, amelyen kommunikálni lehetne egymással. Manapság ez már ritka, hiszen a HTTP-t már a legtöbb rendszer ismeri. Régi alkalmazásoknál viszont előfordulhat, hogy nem tud HTTP-n kommunikálni. A felismert problémákra találták ki az aszinkron kommunikációt, ami gyakorlatilag azt jelenti, hogy az egyik rendszer elküld egy üzenetet a másiknak és utána fut tovább, nem várja meg a választ. Az üzenet elküldésre került és a válasz bizonytalan időben, de biztosan érkezni fog. Azonban HTTP protokollal ezt nagyon körülményes megoldani.

Erre találták ki a Message Oriented Middleware-eket (MOM), amelyek aszinkron jellegű üzenetküldést tesznek lehetővé. Ezt úgy képzeljük el, hogy van egy middleware - egy szoftver - ami mind a két alkalmazás mellé fel van telepítve. Az egyik alkalmazás csak a maga mellett lévő Message Oriented Middleware-nek küldi el az üzenetet, ami viszont a másik rendszeren futó middleware-vel áll kapcsolatban, ahonnan a másik alkalmazás kiveszi az üzenetet. Miért jó ez a fajta felépítés? Azért, mert az első alkalmazás mellett levő Message Oriented Middleware mindig rendelkezésre áll, mindig hozzá tud csatlakozni az első alkalmazás. Az első és a második middleware között helyezkedik el a hálózat. Ha a hálózat megszűnik létezni, vagy belassul, akkor a két middleware elintézi egymás között a kapcsolatot. Tehát vár addig, amíg a másik oldal helyre nem áll, vagy el nem indul. Kizárólag akkor küldi át az üzenetet, amikor a hálózat elindul. Ha a másik oldalon az adott pillanatban a másik alkalmazás éppen nem fut, akkor sincs baj, ugyanis a második alkalmazás mellett levő middleware addig tárolja az üzenetet, amíg a második alkalmazás el nem indul. Amint

elindul, onnan már ki tudja venni az üzenetet. Gyakorlatilag tehát nem nekem kell a hibát kezelni, mert az üzenet átjuttatását az egyik rendszertől a másikig a middleware végzi el. A middleware-k összefoglalóan olyan szoftver rendszerek, amelyek funkcionalitásaikat, mint például a megbízható üzenet továbbítást, elrejtik egy API mögé. Úgy gondoljunk rájuk, mint az adatbázisokra. Az adatbázisoknak az a feladata, hogy adatokat tároljon. Ezt egy API mögé rejtjük el és ezen az API-n keresztül lehet megszólítani. A Message Oriented Middleware-nél magát az üzenet küldést nevezzük a funkcionalitásnak. A Message Oriented Middleware-ek üzenetet tudnak küldeni, ezt egy API mögé rejtjük és az alkalmazások ezt az API-t hívják meg. Mindkét esetben feladatot vesznek le a vállunkról. Az adatbázis adatot tárol, a middleware üzenet továbbít. A Message Oriented Middleware-ek esetében hívják ezt „Store and forward"-nak is. A middleware eltárolja az adott kérést és amikor helyre jön a hálózat, vagy elindul a másik rendszer, akkor továbbítja magát az üzenetet a második rendszer felé.

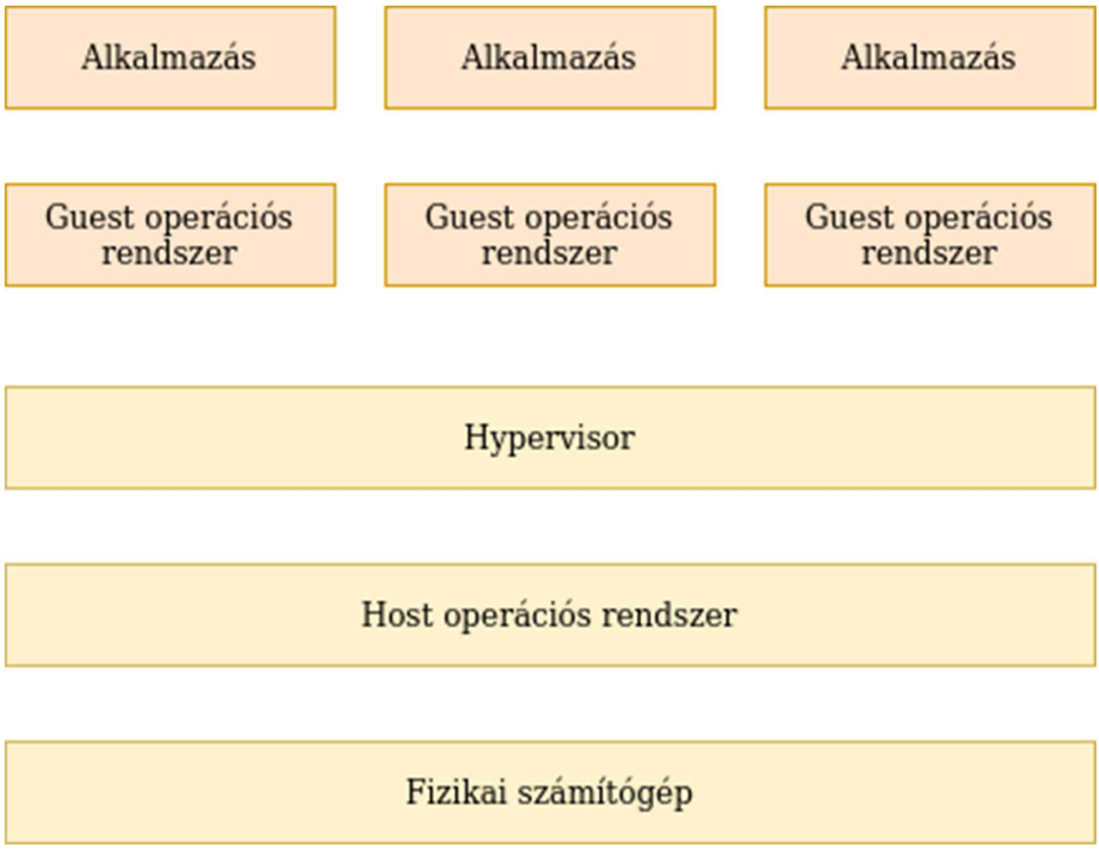
Kétféle modell van az üzenet küldésben, a Point to point és a Publish and subscribe. A Point to point kommunikáció folyamata a következő. Egyrészt van az A és a B alkalmazás, két külön rendszer. Az A alkalmazás szeretne üzenetet küldeni a B alkalmazásnak. Ez úgy történik, hogy a Message Oriented Middleware-be üzenet küldés történik, a legtöbb esetben egy XML vagy egy JSON, vagy egy hasonló formátumban. (Fontos tudni, hogy a Message Oriented Middleware-kbe definiáltak úgynevezett sorokat, melyekbe az üzenetet lehet berakni és kivenni.) Az A alkalmazás létrehozza az üzenetet, majd bepakolja a queue-ba, a B alkalmazás a queue-ból pedig kiveszi. Az egyik üzenetet küld, a másik pedig üzenetet fogad. Bár a rendszerek viszonylag össze vannak kapcsolva egymással, nagy előnye a Message Oriented Middleware-nek, hogy nem kell mindig rendelkezésre állnia, nem kell mind a kettőnek futni, sőt még a hálózatnak sem kell állandóan működnie kettejük között. Az üzenet a kézbesítés lehetőségéig a sorban tárolódik. Amikor megjavul a hálózat, amikor elindul a második rendszer, az ki tudja venni a sorból az üzenetet. A Message Oriented Middleware gondoskodik arról, hogy amíg nem ért célba az üzenet, addig magánál tárolja.

A másik modell a Publish and subscribe. Itt gyakorlatilag az történik, hogy deklarálunk egy topic-ot és az egyes alkalmazás erre a topic-ra küld üzenetet. A topic-ra fel tudnak iratkozni további alkalmazások. Amennyiben erre a topic-ra üzenet érkezik, akkor azt minden egyes feliratkozó alkalmazás megkapja. Itt a küldő alkalmazás nem is ismeri a többi topic-ra csatlakozott alkalmazást, ő csak a topic-ot ismeri,így nem csak egy alkalmazásnak, hanem ilyen formán több alkalmazásnak is tud üzenetet küldeni. Ha analógiát akarunk hozni akkor a Point to point egy egyszerű levélküldéshez hasonlít. A mailszerver tölti be aqQueue szerepét, az tárolja az üzenetet, amíg a címzett fél meg nem kapja. A Publish and subscribe pedig a levelezési listának felel meg. Amikor a felhasználó küld egy levelet a levelezési listára, akkor az összes többi felhasználó, aki erre a levelezési listára fel van iratkozva, megkapja ezt a levelet.

Virtualizáció

Virtualizáció esetén szoftveresen valósítunk meg egy olyan dolgot, ami fizikailag létezik.

Van egy fizikai számítógép, erre feltelepítünk egy operációs rendszert, ezen belül létrehozunk egy virtuális számítógépet. Az így létrehozott virtuális gép olyan, mint egy igazi számítógép, van processzora, memóriája, tárhelye. Erre is telepítünk egy operációs rendszert - mely akár eltérő is lehet, mint a fizikai gépé -, majd különböző alkalmazásokat. A fizikai számítógépre telepített operációs rendszer a host, a virtuális számítógépre telepített operációs rendszer a guest. A guest operációs rendszer, amikor telepítésre kerül, nem „tudja“, hogy ő virtuális környezetben fut, azt „gondolja”, hogy tényleg valós, igazi számítógépen fut. Ez azt is lehetővé teszi, hogy egymás mellett többet is elindítsunk, például egy Windowson akár 2-3 Linuxot is el tudunk indítani. A virtualizáció azt is jelenti, hogy a számítógép részegységei (processzor, memória, stb.) is szoftveresen vannak emulálva. Természetesen ennek vannak hátrányai is. A guest operációs rendszer lassabb, mint a host operációs rendszer, mert a hívások sok közbülső rétegen mennek át.



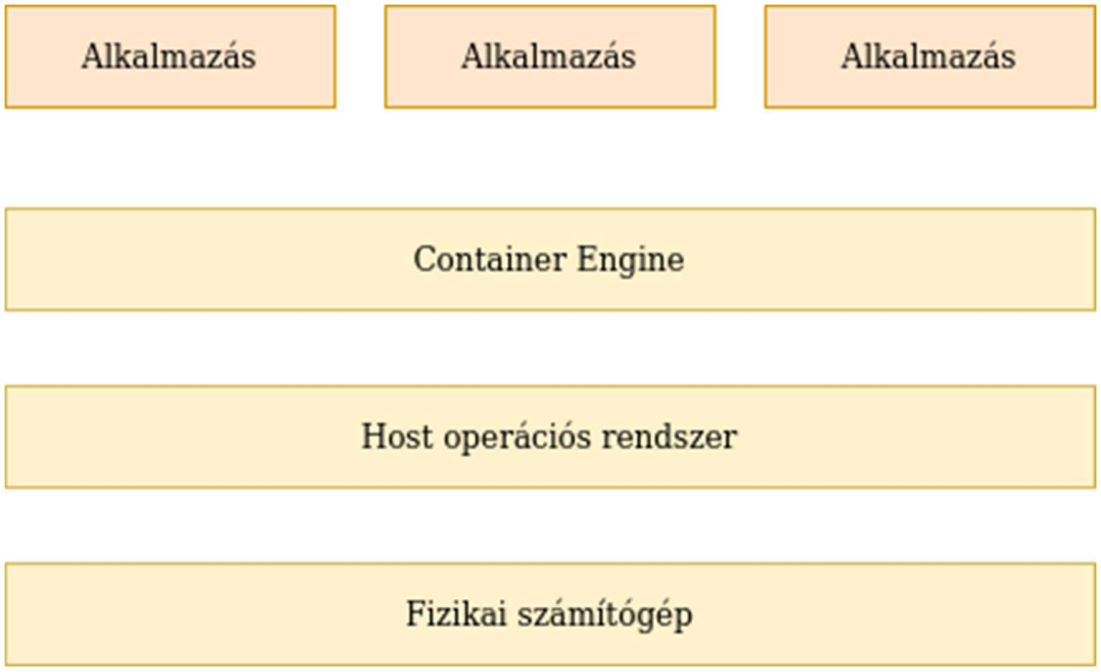
Virtualizáció

A Hypervisor feladata az, hogy fizikai számítógépként mutassa magát, szoftveresen emulálja a számítógépet. Erre lehet feltelepíteni a guest operációs rendszert (rendszereket), amelyek egymással párhuzamosan futnak. A guest operációs rendszerek teljes értékűek, nemcsak egy, hanem több alkalmazást is futtathatunk rajtuk.

A virtualizáció ideális technológia a számítógépek optimális kihasználásához. Így például három elavult, ritkán használt alkalmazáshoz - melyek különböző operációs rendszereken futnak, három különböző fizikai számítógépen - nem kell egy-egy teljes fizikai gép, ami futtatja, elegendő egy fizikai gépen három virtuális gép. A másik felhasználási terület, hogy kipróbálhatunk másik operációs rendszert a saját meglévő operációs rendszerünk mellett párhuzamosan. A kipróbálás után fokozatosan térhetünk át, vagy egyszerűen vissza is léphetünk. Ez alkalmazás fejlesztése során is hasznos. Tökéletes arra is, hogy úgy próbálhassunk ki alkalmazásokat, hogy a saját számítógépünket ne érhesse kár, azt ne veszélyeztessük egy kevésbé ismert és nem feltétlenül megbízható alkalmazás telepítésével. Ha esetleg valamilyen kárt okoz az alkalmazás, akkor csak a virtuális környezetet teszi tönkre, ugyanis a virtuális gépből nem tud kinyúlni az alkalmazás. Tesztelhetünk vele architektúrát is. Példaként, ha van egy komplex szoftverrendszerünk, amely több szoftverből áll, ezek különböző gépekre vannak telepítve, és ezek hálózaton keresztül kommunikálnak egymással, akkor eddig csak annyi lehetőség volt, hogy feltelepítettük jó néhány számítógépre az alkalmazást, és kipróbáltuk működik-e. Innentől kezdve akár egy számítógép is elegendő lehet erre a tesztelésre. A környezet replikálására is kiváló. Példaként, ha sok felhasználó ugyanazt a környezetet használja, konkrét feltelepített szoftverekkel, akkor nem szükséges minden felhasználó gépére külön-külön egyesével telepíteni mindent, hiszen elegendő egy virtuális gépre feltelepíteni ezeket, majd csak a virtuális gépet másolni a felhasználók gépére. Lokáció függetlenséget is meg lehet valósítani, mert a technológia lehetővé teszi a működés közbeni költöztethetőséget. Futás közben lehet az operációs rendszert egyikről a másik számítógépre áttelepíteni, melyből a felhasználó csak egy kismértékű lassulást érzékel, mert tizedmásodpercek alatt kivitelezhető. Miközben a felhasználó folyamatosan dolgozik az első számítógépen, addig első lépésként nagyobb darabokban másolódik át az operációs rendszer a másik számítógépre, de közben a virtualizációs megoldás csak a változásokat kezdi átvinni a másik virtuális számítógépre. Ezek egyre kisebb darabok lesznek, és amikor már egészen pici, akkor leállítja az első számítógépet, hogy azon a felhasználó már ne tudjon dolgozni, átviszi azt a pici darabkát, amit utoljára módosított a felhasználó, ami gyakorlatilag a memória tartalma. Végül áttereli a felhasználót a másik számítógépre.

A legelterjedtebb virtualizációs megoldások a következők: VMware, Oracle VirtualBox (ingyenesen használható), Vagrant (virtuális gépek menedzselésére, kezelésére).

Jelentősen előbbre vitte az IT világot a Docker megjelenése, ugyanis sokkal kevesebb erőforrással lehet virtualizációt megvalósítani, de kiemelendő, hogy ez nem virtualizációs technológia, hanem konténerizáció.



Konténerizáció

A Cloud computing alapja is a virtualizáció, hiszen különböző virtuális gépeken futnak az egyes ügyfelek különböző számítógépei. De mi is az a Cloud computing? Több száz vagy több ezer darab számítógépből álló számítástechnikai infrastruktúra biztosítása szolgáltatásként cégek számára. Gyakorlatilag a fizikai erőforrás bérlése, ahol a bérleti díj összege az éppen adott szükséglethez igazodik. Klasszikus példa az erőforrás kiszervezése, de cloudot a cégen belül is lehet építeni, ekkor beszélünk privát/belső felhőről (private cloud).

Több fajtát is megkülönböztetünk. Az Infrastructure as a Service (IaaS): a szolgáltató virtuális számítógépeket biztosít, az operációs rendszer telepítése és frissítése mellett a saját alkalmazások feltelepítése is a bérlőnek a feladata. A hardveren kívül mindent a bérlő ad, ez adott esetben hátrány, de előny is speciális elvárásoknál. Magasabb szint a Platform as a Service (PaaS): az operációs rendszert a szolgáltató menedzseli és tartja karban, az alkalmazást a bérlő telepíti. Még magasabb szint a Software as a Service (SaaS): a szoftvert kapja meg és használhatja online az igénybe vevő.

A Docker felpörgette a virtualizációs megoldásokat, mert operációs rendszer szintű virtualizációt tesz lehetővé. Pontosan ugyanarra használható, mint a virtualizáció, csak gyorsabb, kevesebb erőforrást igényel, ugyanis itt az operációs rendszer van belül feldarabolva szeparált részekre. Az alkalmazásokat az operációs rendszeren belül párhuzamosan lehet futtatni az operációs rendszerre való feltelepítés nélkül, mivel az alkalmazások egy kis elkülönített részben futnak az adott operációs rendszer belül. Valójában itt nem áll rendelkezésükre egy teljes operációs rendszer, hanem a host operációs rendszer kerneljét (az operációs rendszer magja, ami a hardvert vezérli) használják.

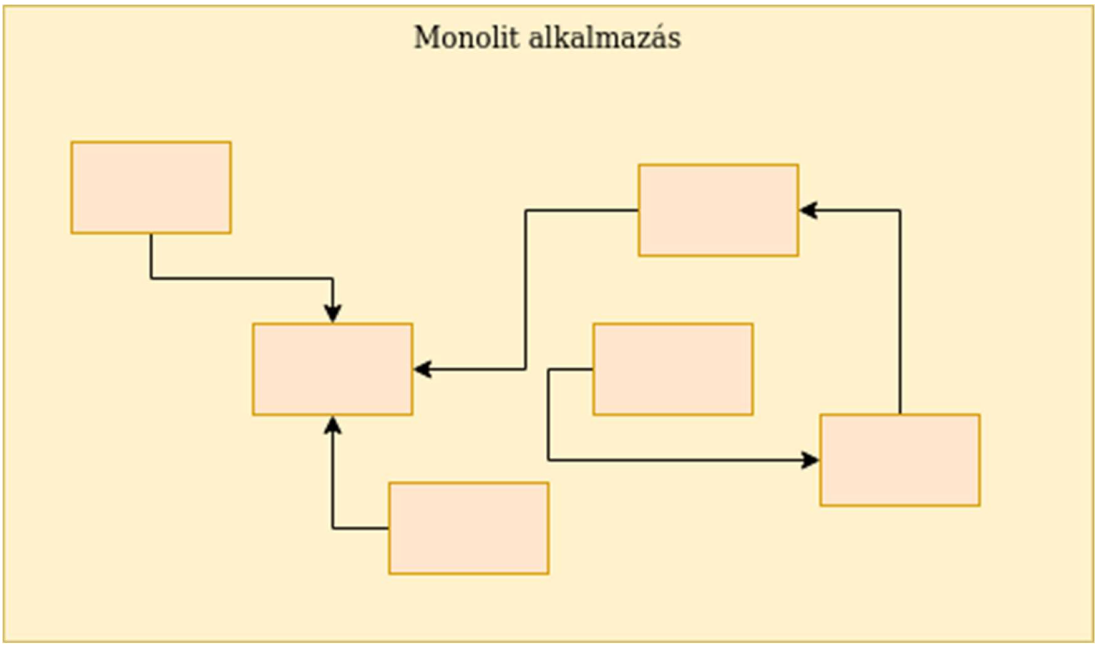
A host operációs rendszer felett egy Container Engine van - ezért hívják konténerizációs technológiának - majd ezeken belül nem egy másik operációs rendszer van, hanem közvetlenül az alkalmazások egy elkülönített térben, és ezeket az alkalmazásokat a host operációs rendszer kernelje futtatja. Összefoglalva, a konténerizáció egy gyorsabb virtualizációs technológia, célja az elszeparált környezetben, de ugyanazon a számítógépen való alkalmazás futtatás.

A Docker azért vált nagyon elterjedté, mert a virtualizációs megoldás biztosítása mellett további plusz komponensekkel rendelkezik. Egyik a Docker Hub. Ez egy online szolgáltatás, ahová a „virtuális gépeket" lehet fel- és letölteni. A Docker Compose egy külön egység a Dockeren belül, amivel egyszerre több Docker konténert lehet vezérelni. A Docker Swarm képes arra, hogy magas rendelkezésre állást biztosítson, egy konténerből akár több példányt indítson el, és ezek között terheléselosztást végezzen. A Docker Machine távoli számítógépekre telepített Docker vezérlésére alkalmas.

A Docker két alapfogalma az image és a konténer. Az image tartalmazza az adott alkalmazáshoz tartozó és az operációs rendszerhez tartozó fájlrendszert tömörítve, ezt a teljes fájlrendszer futtató példányt hívjuk konténernek. Ugyanabból az image-ből akár több konténert is el lehet indítani, melyek a virtualizációnak köszönhetően teljesen függetlenek lesznek egymástól.

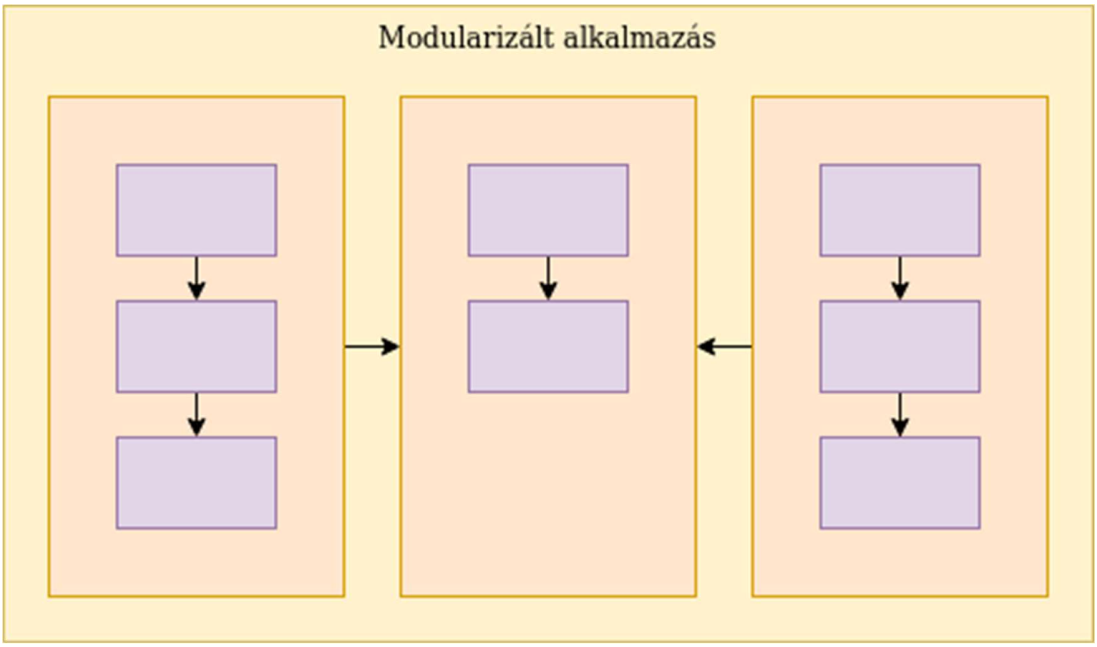
Alkalmazások és alkalmazás rendszerek felépítése

Azokat az alkalmazásokat hívjuk monolitikus rendszereknek, amelyek túl nagyra nőttek, nem rendelkeznek belső struktúrával, és így a fejlesztésük nagyon nehézkes, mert nehéz hozzájuk új funkciókat implementálni. Többnyire a monolitikus alkalmazások is kezdetben átlátható, könnyen fejleszthető kisebb alkalmazások voltak, azonban idővel, ahogyan újabb funkciók kerültek bele szoros határidőkkel és kevésbé kidolgozva, oda vezetett, hogy az alkalmazás belső struktúrája átláthatatlan lett. Az alkalmazást alkotó komponensek össze-vissza kapcsolódnak egymáshoz - spagetti struktúra - , melynek az a veszélye, hogy ha az alkalmazás egyik részét módosítják, bővítik, sokszor a másik részén lévő funkciók is elromlanak. Ez az oka, hogy a monolitikus alkalmazások módosítása, tesztelése, egy új verzió létrehozása rendkívül időigényes.



Monolit

Az egyik megoldás a fenti problémára az alkalmazás modulokra bontása. Napjainkban – jó esetben - már úgy kezdenek el egy alkalmazást fejleszteni, hogy rögtön modulokra bontják. Az egyes modulokban pedig jól meghatározott struktúrában komponensek helyezkednek el. A modulok általában funkcionalitásuk alapján vannak szétválasztva, viszonylag függetlenek egymástól, csak kis kapcsolódási pontjuk van, így elkerülhető a spagetti hatás. (Egy webshop lehetséges moduljai például: termékek nyilvántartása, vásárlók nyilvántartása, kosár kezelése).



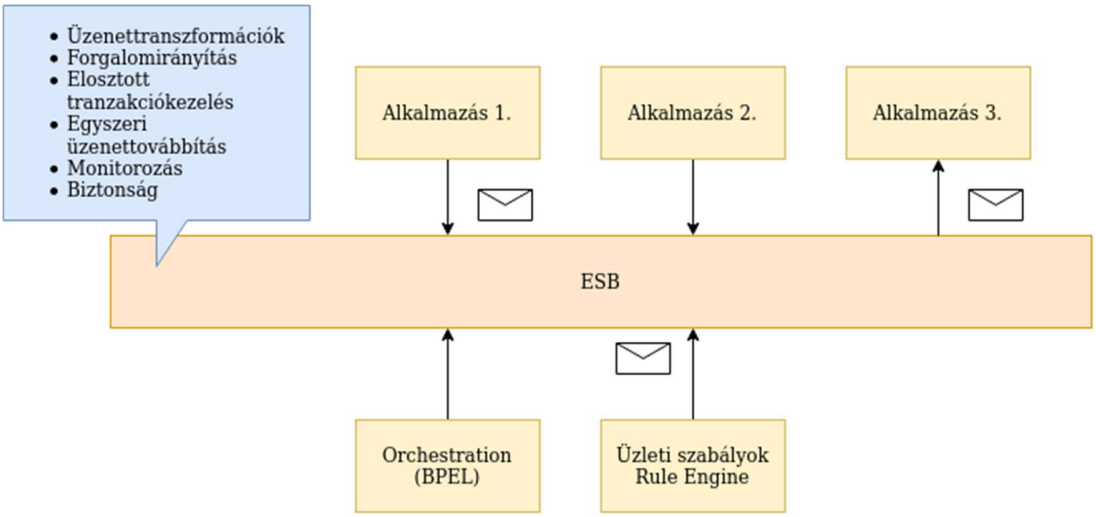
Modularizáció

A modularizáció az önmagában álló alkalmazásokra alkalmazható, de a modern alkalmazásokra nem. A mai modern alkalmazások már nem elszigetelt rendszerek, hanem további más alkalmazáshoz kapcsolódnak. Ennek kialakulása hasonló az önmagában álló monolitikus alkalmazások kialakulásához. Folyamatosan merült fel az igény két, majd három, végül az architektúra bővülésével akár több alkalmazás összekapcsolására is, mely megint csak nem lett strukturált, inkább egy valóságos pókháló. Erre találták ki a SOA (Service Oriented Architecture) architektúrát, mely az alkalmazások között biztosít átláthatóbb felépítést. A SOA architektúrában minden egyes alkalmazásnak egy jól meghatározott feladata van (például termék nyilvántartás), és ezt a funkciót, mint szolgáltatást biztosítja, melyet a többi alkalmazás igénybe vesz. Tehát minden alkalmazásra szolgáltatásként kell tekinteni a SOA architektúrában. A SOA esetén SOAP-os webszolgáltatásokat szoktunk használni, ahol az interfészt WSDL dokumentummal lehet leírni. Az alkalmazások között a kommunikáció vagy szinkron HTTP hívás vagy pedig aszinkron kommunikáció egy Message Oriented Middleware-en keresztül.

A SOA architektúrába fontos szereplő az Enterprise Service Bus (ESB), mely egy külön software, egy külön middleware, azaz egy köztes réteg. Az architektúra közepén helyezkedik el az Enterprise Service Bus, melyhez közvetlenül és egyesével kapcsolódnak az alkalmazások. Azaz tehát nem egymáshoz vannak össze-vissza kötve, hanem csak egy kapcsolat van az Enterprise Service Bus felé. Az Enterprise Service Bus valósítja meg alkalmazások között a kapcsolatot, így az alkalmazások az ESB-n keresztül tudnak egymással kommunikálni, klasszikusan üzenetekkel.

Amellett, hogy az ESB központi szerepet tölt be, még számos feladatot el tud végezni. Egyrészt tud üzenet transzformációkat, azaz a küldő alkalmazás üzenetét képes úgy formázni, hogy a fogadó alkalmazás azt értelmezni tudja. Továbbá képes még forgalomirányításra, mely azt jelenti, hogy az ESB a beérkező üzenet fejléce alapján képes

eldönteni, melyik alkalmazásnak továbbítsa. Tud tranzakció kezelést is, úgynevezett elosztott tranzakciót. Ez azt jelenti, hogy például a tranzakció az A alkalmazásban indul és a B alkalmazásban ér véget. Az ESB-k képesek pontosan egyszeri üzenet továbbítására is, így biztosak lehetünk abban, ha egyszer feladunk egy üzenetet, akkor az célba fog érni, lehet nem azonnal, de előbb-utóbb mindenképpen. Az ESB-n keresztül lehet monitorozni is a rendszereket, mely alkalmazások működnek, melyek nem, milyen az alkalmazások közti kommunikáció, stb. Az ESB még a biztonságért is lehet felelős. Akár az üzenetek titkosításáért, vagy akár az üzenetek elektronikus aláírásáért. A központi szerepe miatt az Enterprise Service Bus egy Single point of failure. Azaz ha tönkremegy, akkor onnantól kezdve az összes alkalmazás elszigetelt rendszerként működik tovább, nem tudnak egymással kommunikálni. Ezért célszerű az ESB-t erősen túlbiztosítani, redundanciát alkalmazni.

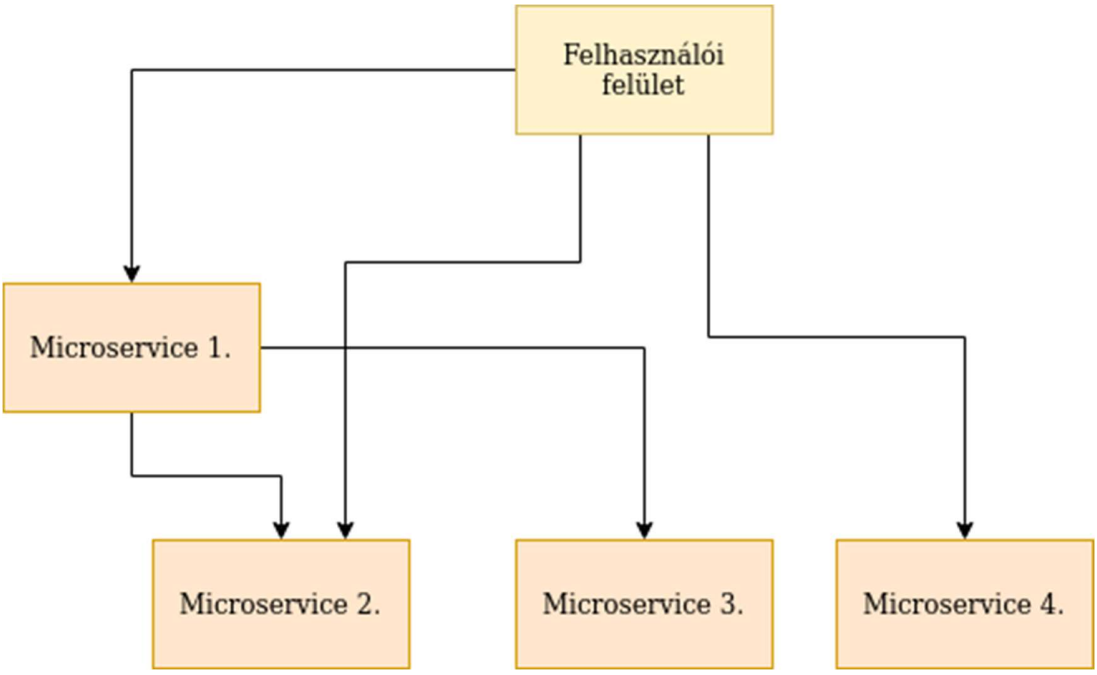


ESB

Két további SOA komponenst is érdemes megtárgyalni. Az egyik egy úgynevezett BPEL Engine, ami az orchestrationt végzi. Több alkalmazáson keresztül vezető üzleti folyamatok esetén, mint karmester szerepel, ez mondja meg az egyes alkalmazásokat milyen sorrendben kell meghívni. További komponens a Rule Engine, mely az üzleti szabályokat tartja nyilván. Ennek megalkotásakor fontos szempont volt, hogy programozói tapasztalat nélkül is megfogalmazhatók legyenek az üzleti szabályok, illetve az üzleti szabályokat akár egy üzleti felhasználó is módosíthassa (gyorsan változó üzlet, üzleti titkok).

Napainkban nagyon trendi architektúra a Microservices, arra való, hogy az alkalmazást felbontsuk különálló pici alkalmazásokra. Nem modulokra, hanem még ennél is tovább megyünk, olyan alkalmazásokra, amelyek külön üzemeltethetőek, külön fejleszthetőek, külön frissíthető a verziójuk stb. Tehát valójában olyan apró alkalmazásokra bontjuk fel a nagy alkalmazást, amelyet például egy hatfős csapat is képes átlátni és fejleszteni. Egy ilyen kis alkalmazással szemben elvárás, hogy egy ember képes legyen átlátni az alkalmazáson belüli összefüggéseket, így biztosítva azt, hogy egy hiba ne okozzon további hibát. Minden egyes alkalmazásnak van egy apró feladata és ezt a feladatot hibamentesen biztosítja, melyet más alkalmazások API-n keresztül tudnak meghívni. Egymáshoz pehely súlyú

protokollokkal kapcsolódnak, RESTful webszolgáltatásokon vagy aszinkron módon egy Message Oriented Middleware-n keresztül. A különböző apró alkalmazások különböző technológiával valósíthatóak meg, a fejlesztők szabadon választhatnak, így minden feladatra a legmegfelelőbb eszközt tudják alkalmazni.



Microservices

A microservice architektúra esetén a felhasználói felület mögött nem egy nagy monolitikus alkalmazás áll, hanem sok-sok microservice. (ilyen például a Spotify, ahol külön alkalmazás a kedvenc zenék, az ajánló, az ismerősök nyilvántartása, stb.). Az egyes kis alkalmazások külön fejleszthetők, frissíthetők, melyből a felhasználó nem érzékel semmit, hiszen ő csak az egységes felhasználói felület látja.

Microservice esetén a nem funkcionális követelmények is könnyebben teljesíthetők, mint például a magas rendelkezésre állás és a skálázhatóság is sokkal egyszerűbben megvalósítható. Ide kapcsolódó fogalom az elasztikusság, elasztikus működés. Az infrastruktúra önállóan érzékeli a változó terhelést, és ennek megfelelően automatikusan, üzemeltetői beavatkozás nélkül reagál, dinamikusan horizontálisan skálázza az adott microservice-t (több példány indítása adott microservice-ből terhelés növekedésekor, futó példányok számának csökkentése terhelés visszaesésekor). A microservice-ket nem valós számítógépen, hanem Cloud infrastruktúrán, a virtualizációt, vagy manapság konténerizációt használva futtatjuk.

Természetesen a microservices-nek hátrányai is vannak. Az üzemeltetése, a nyomonkövetés - mi kapcsolódik mihez - , sokkal bonyolultabb, mint egy nagy alkalmazást üzemeltetni. A microservice architektúra felépítéséhez nagyon sok más komponensre is szükség van (például az elasztikusságot biztosító szoftver) és kevés jól képzett szakember van hozzá. A microservice-nél nagyon fontos, hogy minden microservice magánál tárolja a

saját adatait, tehát az adatok szét vannak osztva különböző microservicek-nél. Ez azzal jár, hogy egy bonyolultabb lekérdezéshez nem használható egy központi adatbázis, hanem sok helyről kell a szükséges adatokat összegyűjteni. Emellett a microservice-nél a tranzakciók kezelése is nehézkes. Problémás azt megoldani például, hogy az egyik microservice-ben elinduljon egy tranzakció, majd ehhez csatlakozzon további néhány microservice, végül az utolsó microservice lezárja a tranzakciót.