Szoftware architecture

# Mi az architektúra

Struktúrák, amiket arra használunk, hogy a szoftver rendszerekről, felépítésükről és a létrehozásukról tudjunk beszélni. Mindegyik ilyen struktúra szoftver elemekből, a közöttük lévő kapcsolatokból és ezek tulajdonságaiból áll.

Egy épület tervrajzának az analógiája. Az architektúrát használjuk a szoftver rendszer megtervezésére. Konkrétabban a fejlesztési (tesztelési) project és a project management használja az elvégzendő feladatok megértésére.

Az architektúrák annak az eszközei, hogy alapvető strukturális döntéseket hozzunk, amiket később költséges megváltoztatni. Az architektúra halmazban kerülnek kiválasztásra a konkrét strukturális választások a sok lehetőség közül.

A dokumentálásuk lehetővé teszi a kommunikációt a stakeholderek között, segít korán meghozni magas szintű döntéseket és megteremti a lehetőséget az egyes komponensek újra felhasználására.

## Különböző megközelítések

* A rendszer szerkezetének leírása nagy vonalakban: Magas szintű absztrakciós szint, amely tartalmazza a komponenseket és konnektorokat, amelyek leírják az interakciókat a komponensek között
* A fontos dolgok: Azokra a döntésekre kell figyelnie, amiknek nagy hatásuk van a a rendszerre és a stakeholderekre
* Mi fontos a rendszer és környezetének megértésében
* Azok a dolgok, amiket úgy érzünk, nehéz megváltoztatni: azért is, mert az architektúrák megírása a szoftverfejlesztési életciklus elején történik, ideális jól meghozni a fontos döntéseket. Olyan karakterisztikákat tudunk bevezetni a szoftverbe, amit később nem tudunk megváltoztatni.
* A döntések kiegészítése a meghozatalukig begyűjtött információval: Nem elvágólag létezik a követelményektől, a magas és alacsony szintű részletek láncolatának része.

## Miért hasznos

* Alapot ad arra, hogy vizsgáljuk a rendszer viselkedését, mielőtt a rendszert létrehoznánk.
  + Statikus tesztelés
  + Test plan
  + Kockázatfelmérés
* Ismereteket ad arra, hogy szoftver elemeket vagy döntéseket újra felhasználjunk
* Olyan tervezési döntéseket lehet korán meghozni, amik a fejlesztésben, tesztelésben, deployment-ben és fenntartásban lényegesek lesznek.
* Segít abban, hogy a stakeholderekkel lehessen kommunikálni az igényekről és a rendszer képességeiről
* Segít a kockázatkezelésben
* Segít a költségek kezelésében

### Komponens, modul, adatmodell

A moduláris programozás azt jelenti, hogy egy szoftver funkcionalitását szétválasztjuk független és kicserélhető modulokra úgy, hogy minden modul tartalmaz mindent, ami szükséges a kívánt funkcionalitás pontosan egy aspektusának a végrehajtására.

A szoftver komponens egy szoftvercsomag, web szolgáltatás, web erőforrás vagy modul ami magában foglal összefüggő funkcionalitások vagy adatok egy halmazát.

Az adatmodell egy absztrakt modell, ami adatokat szervez és leírja, hogy azok milyen viszonyban vannak egymással és a valóságban található dolgok tulajdonságaival.

# Architektúra mintázatok (pattern)

## Model-view-control (MVC)

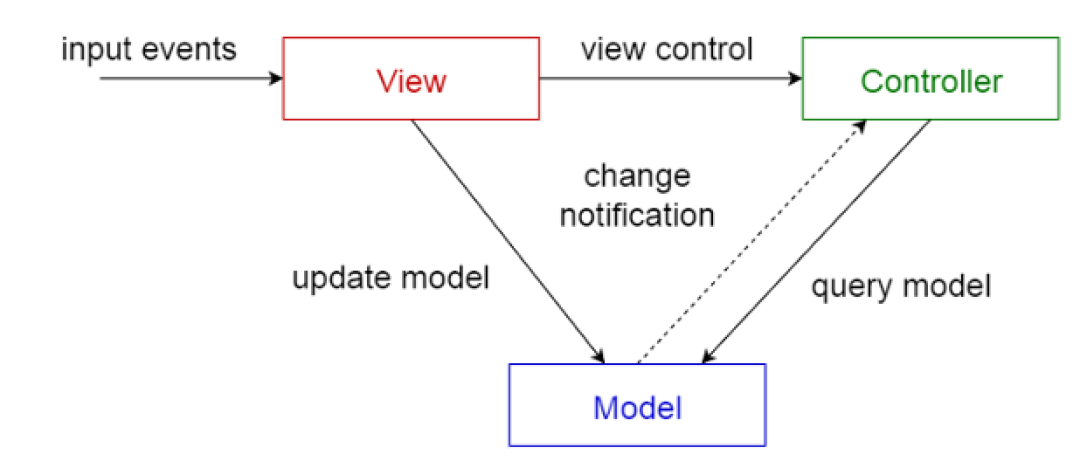
3 részre osztjuk az applikációt: model-re, view-ra és controller-re.

A model, ami a központi része a mintázatnak, tartalmazza az applikáció adatait és központi (core) szolgáltatásait. Ez a dinamikus adat struktúrája az applikációnak és ez kezeli az adatokat és a logikát. Azt nem tartalmazza, hogy az adatok hogy legyenek megjelenítve a felhasználónak.

A view jeleníti meg az applikáció adatait és ez beszél a felhasználóval. Eléri az adatokat a modellben, de nem tudja, hogy az mit reprezentál, vagy hogy azzal mi történik a logikában.

A controller kezeli a bemeneteket a felhasználótól és végzi a kommunikációt a modell és a view között. Figyel arra, hogy milyen inputok érkeznek a view-ból és megjeleníti a megfelelő outputokat. A controller beszél a modellel úgy, hogy hívja a megfelelő metódusait, hogy a megfelelő outputokat meg tudja jeleníteni.

A 3 komponens valamilyen értesítési mechanizmuson keresztül kommunikál, mint event-ek (lásd lejjebb) vagy callback-ek (amikor egy A végrehajtható kód argumentumként megkap egy B végrehajtható kódot és a A végrehajtása közben meghívja B-t). Ezek az értesítések állapotokról tartalmaznak információt (állapotváltozások), amiket továbbítanak egymásnak a komponensek, hogy tudjanak frissülni. Például a user csinál valamit a view-ban, amit az továbbít a controller-nek, ami frissíti a view-t.



Példák erre a mobil- és web applikációk.

Az MVC model lehetőséget ad a componensek szétválasztására. Ez a decouple-ing. Coupling jelöli a függőség mértékét különböző szoftver elemek között. Annak mértéke, hogy mennyire van összekötve két modul, komponens vagy más elem, vagy mennyire erősek az összefüggések közöttük.

### Coupling formái

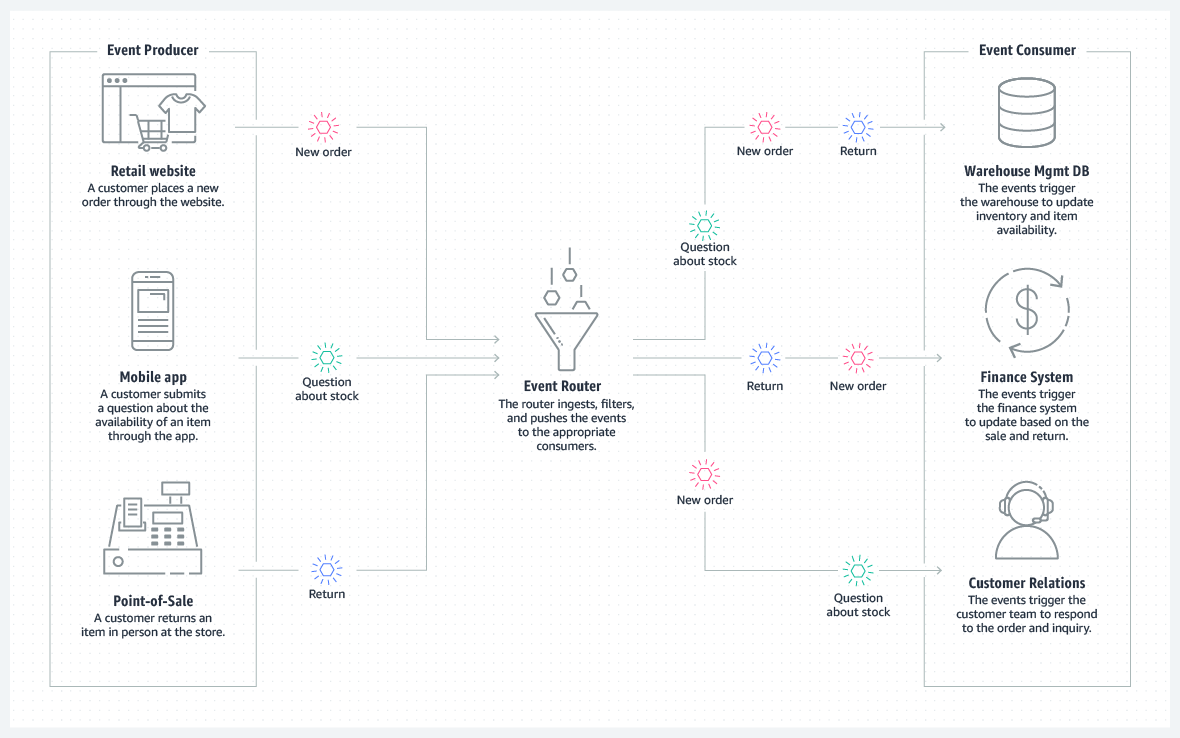
* Procedurális (subroutine: egy vagy több utasítás, aminek van neve és jobb esetben saját változói)
  + Összetevő: A modul használ kódot a B modulból
  + Közösségi: Több modul nak van hozzáférése ugyanahhoz a globális adathoz
  + Külső: Több modul közösen használ kívülről látható adat formátumot, kommunikációs protocol-t vagy eszköz interface-t.
  + Control: A modul irányítja, hogy B modul hogy fusson (control flow) (what to do flag)
  + Adatszerkezet: Több modul osztozik egy közös adatstruktúrán és csak egy részét használják. Ha változik az adatszerkezet, lehet, hogy a modulnak is kell, pedig lehet, hogy nem is használja a változtatott részt.
  + Adat: adatok megosztása pl paramétereken keresztül
* Objektum orientált:
  + Gyerek osztály: A gyerek osztály kap működést a szülő osztálytól, de fordítva nem.
  + Időbeli: Két művelet egy modulba kerül, csak mert egyszerre végezzük őket
  + Dinamikus: A rendszer működése közben (run-time) határozza meg a működését. (öröklés vagy polimorfizmus)
  + Szemantikai: Például kommentek, kód szerkesztése vagy nevezéktanbeli egyezések
  + Logikai: Mintázatok a rendszer változásainak (change, release history) történelmében. (Amikor A változott, B is stb)

Hátrányok:

* Változás egy modulban rendszerint változásokat tesz szükségessé a hozzá szorosan kapcsolódó (tightly coupled) modulokban
* A modulok összeillesztése több effortot igényel a függőségek mértéke miatt
* Egy modul nehezebben újrahasználható vagy tesztelhető, mert a szorosan kapcsolódó modulokat is bele kell venni

## Event driven architecture (EDA)

Szoftver architektúra mintázat, ami események előidézését, detektálását, felhasználását és rájuk adott válaszokat foglalja magában.



Példa eseményvezérelt architektúrára egy e-commerce oldalon. Ez az architektúra alkalmassá teszi az oldalt arra, hogy hogy reagáljon sokféle forrásból érkező változásra csúcsidőben anélkül, hogy összeomlana vagy túl sok erőforrást allokálna.

* Event producer: Az első logikai réteg ami észleli az interakciót és üzenetként továbbítja a rendszer felé
  + Pl e-mail kliens, e-commerce, fizikai detektor
* Event channel: A második logikai réteg, ez felel azért, hogy a generált üzenet eljusson a feldolgozó egységig (engine or sink).
  + TCP/IP üzenet
  + Input file (XML, e-mail, txt)
  + Több is lehet belőle
  + Ezeket, mivel egyszerre egyet akarunk feldolgozni, aszinkron olvassuk be és que-kban tároljuk
* Event engine: Logikai réteg, ami azonosítja az eseményt, majd kiválasztja és végrehajtja a vonatkozó reakciót. Lehetséges még ellenőrzések végrehajtása is.
* Downstream event-driven activity: Logikai réteg, ahol az események következményeit lehet látni.
  + E-mail üzenet
  + Warning
  + Logolás

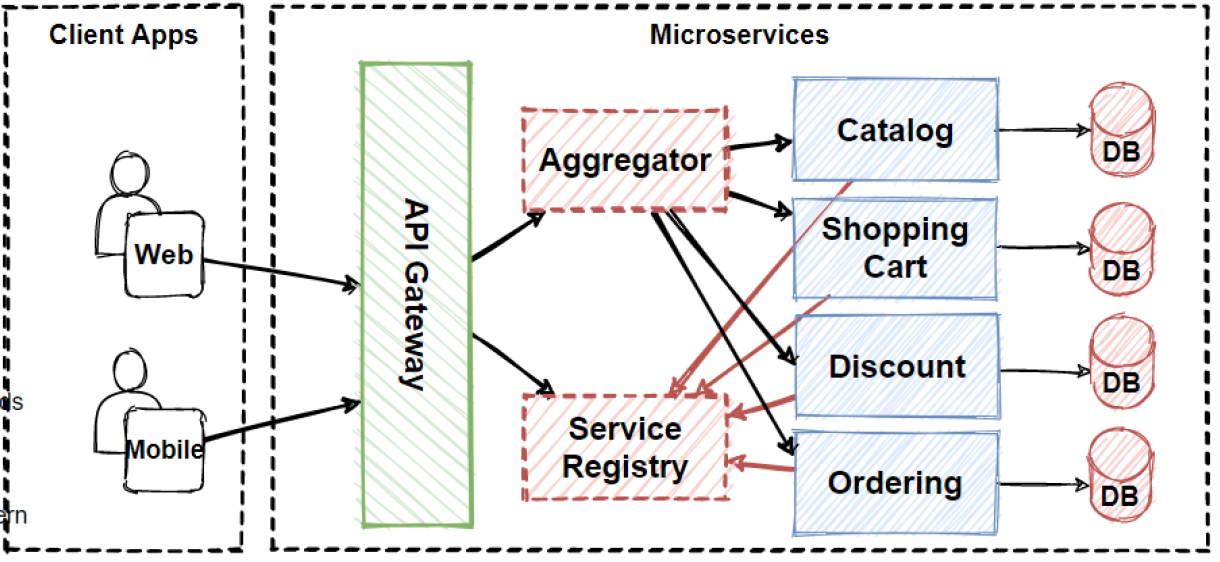
## Microservice

Ez a mintázat azon alapul, hogy több applikációt vagy microservice-t készítünk, amik egymástól függetlenül működnek. Ezeket lehetőség van egymástól függetlenül fejleszteni, viszont a működéseik egybefonódnak a többi microservice-ével.

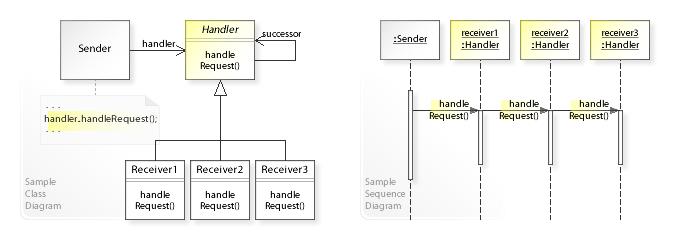
A kulcs koncepció ebben a patternben az egységek elkülönülő telepítése (deployment). Ez egyszerűen kezelhető kiadási folyamatot (delivery pipeline) eredményez, egyszerűvé teszi a microservice-ek telepítését és növeli a skálázhatóságot. Másik kulcs eleme a mintának az, hogy elosztott (distributed), ami azt jelenti, hogy a rendszer komponenseit teljesen le lehet választani a többitől és távoli elérési protocol-okkal (remote access) el lehet őket érni (Pl REST vagy SOAP)

Használják webapplikációkban és website-okon, amik kis elemekből állnak, vagy vállalati data centereknél.

Egy megvalósítás lehet az aggregátor vagy az API gateway:



További megvalósítás lehet a chain of responsibility:



Ebben az esetben a Sender osztály nem közvetlenül hív meg egy fogadó osztályt, hanem elmegy a Handler interface-hez (handler.handleRequest()). Emiatt a Sender független lesz arról, hogy melyik fogadó fogja kezelni a kérését. Reciever1, Reciever2 és Reviever3 osztályok megvalósítják a Handler interface-t úgy, hogy vagy kezelik, vagy továbbítják a kérést, a futás alatt fennálló állapot szerint. Az ábrán a reciever3 fogja kezelni a kérést.

A mintázat hátrányai a komplexitás a hálózati réteg körül illetve a teljes függetlenségből adódó tervezési feladat nehézsége.

### Elosztott rendszer

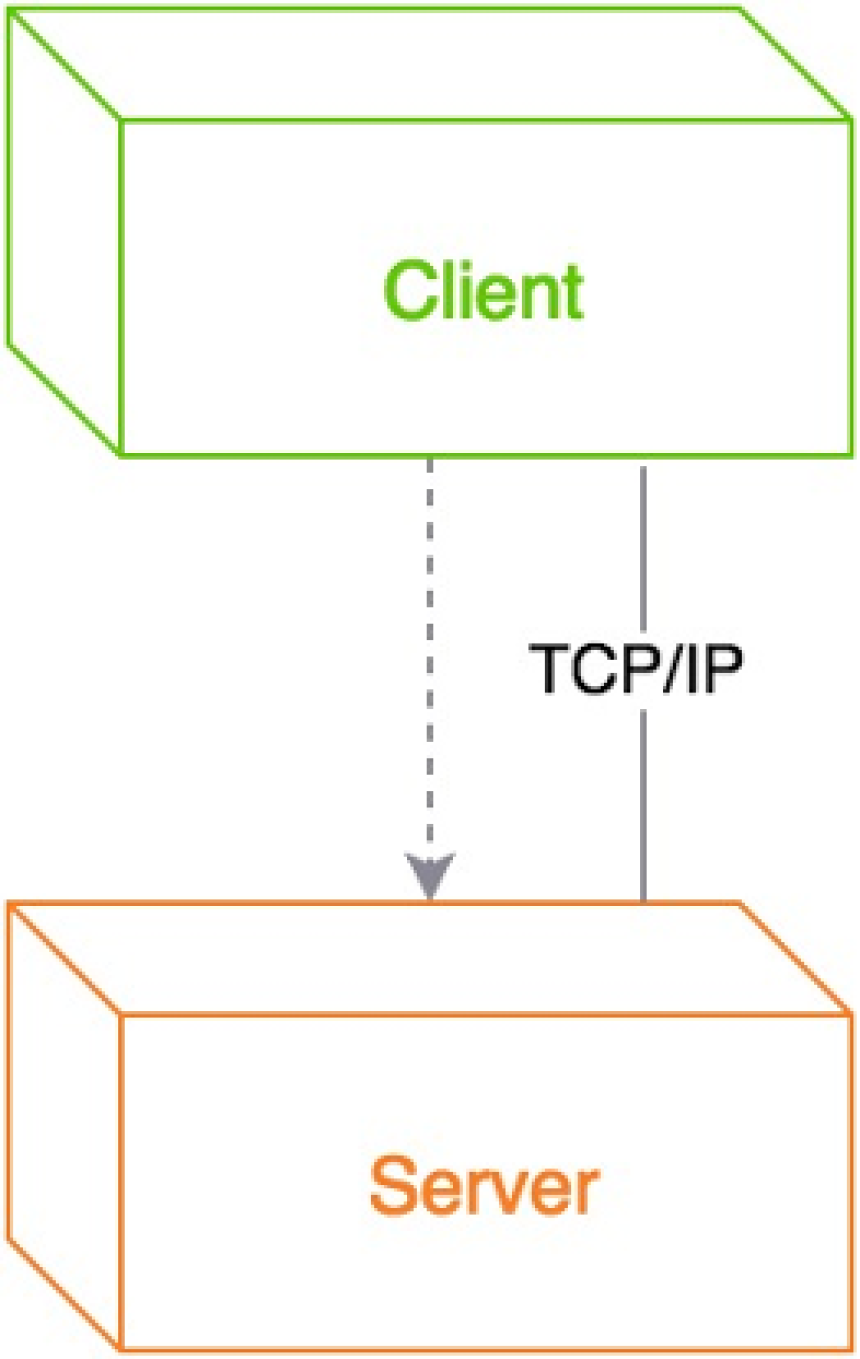
Komponensek különböző platformokon (hardware, software, infrastructure, geographic) jelennek meg és sok komponens működik együtt egy hálózaton keresztül azért, hogy egy kifejezett célt el tudjanak érni. A felhasználónak kívülről egy koherens rendszerként jelenik meg.

* Az információ feldolgozása nincs korlátozva egy gépre, hanem sok függetlenre
* A számítási feladatokat fel lehet bontani és szét lehet osztani a gépek között
* Redundanciát vagy más mechanizmust lehet használni a robusztusság növelésére
* Példa technológiák a .NET, Jenkins, Selenium Grid, IncrediBuild

## Client-server

Ennek a patternnek két fő komponense van, a kliens, aki a szolgáltatásokat kéri (requester) és a szerver, aki pedig a szolgáltatásokat adja (provider). Lehetséges az is, hogy ugyanabban a rendszerben van mind a kettő, de gyakoribb az, hogy hálózaton keresztül kommunikálnak.

A kliens meghatározott üzeneteket küld a szerver felé, hogy a szükséges szolgáltatásokat hozza létre. A kliensnek vannak portjai, amik megfelelnek az igényelt szolgáltatásoknak, a szervernek pedig vannak portjai, amit megfelelnek a nyújtott szolgáltatásoknak. A két komponenst köti össze a connector, amin mennek a kérések és válaszok (request, reply). Netes kommunikáció, file megosztás és e-mail jó példa erre.



Példa erre az online banking. Amikor a bank ügyfele eléri az online banking szolgáltatást a böngészőjében, a kliens elküld egy request-et a bank webserverére. Ebben az esetben a login adatokat használva kér a kliens adatokat a webszervertől. Az application server (backend) feldolgozza a beérkezett kéréseket az üzleti logika szerint (business logic), majd elküldi a megfelelő választ a webszervernek, aki továbbítja az ügyfélnek.

Ennek a felállásnak egy lényeges előnye az, hogy központilag dolgozzuk fel az adatokat. Az összes file, amit használunk nálunk van. Viszont egy szerver fenntartása költséges lehet.

## Controller-responder

Korábban master-slave architektúra, most inkább primary/secondary, primary/replica, parent/helper, master/replica vagy controller/responder (IEEE).

Kétféle komponense van, a controller és a responderek. A kontroller komponens elosztja a bemenetet vagy a munkát az egyforma responder komponensek között és összeállítja az eredményt a responderek válaszaiból.

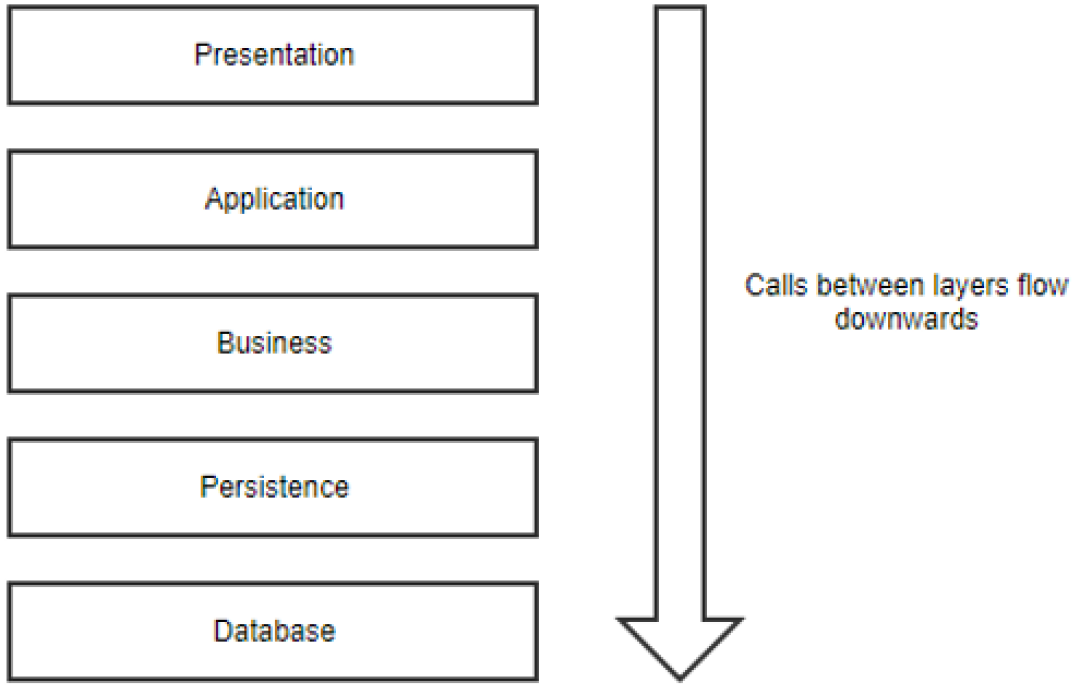
A controller őrzi az adatokat, míg a responderek erről készítenek másolatokat. Adatok írása az adatbázisba csak a controllerben történik, amit a responderek csak olvasnak. Abban különbözik a peer-to-peer felállástól, hogy itt a controller mondja meg, hogy a responderek melyik kommunikációjának mi a prioritása (meg persze a sajátjának is).

Adatok bármikor olvashatók a responderekről anélkül, hogy a controller adatbázisához hozzányúlnánk. Responderek bármikor cserélhetők, vagy újraindíthatók vagy újraszinkronizálhatók. Bajosabb, ha a controllerrel történik baj. Az applikációt újra kell indítani, vagy valamelyik respondert léptetjük elő controllerré (potenciálisan adatvesztéssel)

## Layered

Széles körben használt modell. Jól használható olyan programok tervezésére, amik számos feladatból, műveletből épülnek fel, amelyek csoportjai különböző absztrakciós rétegeken vannak. Minden ilyen absztrakciós réteg a modell egy rétegeként jelenik meg. Mindegyik réteg egy összefüggő szolgáltatás csomagot tartalmaz és szolgáltatásokat nyújt a felette lévő rétegnek (de a másik irányba nem).

Minden rétegnek van egy specifikus szerepe az alkalmazáson belül, ami kapcsolódik a többi réteg szerepéhez. Például lehet egy megjelenítő (presentation) réteg, amely kezeli a kapcsolatot a UI és a böngésző között és lehet egy üzleti logika (business logic) réteg, ami pedig a hozzá beérkező kéréseket kezeli.

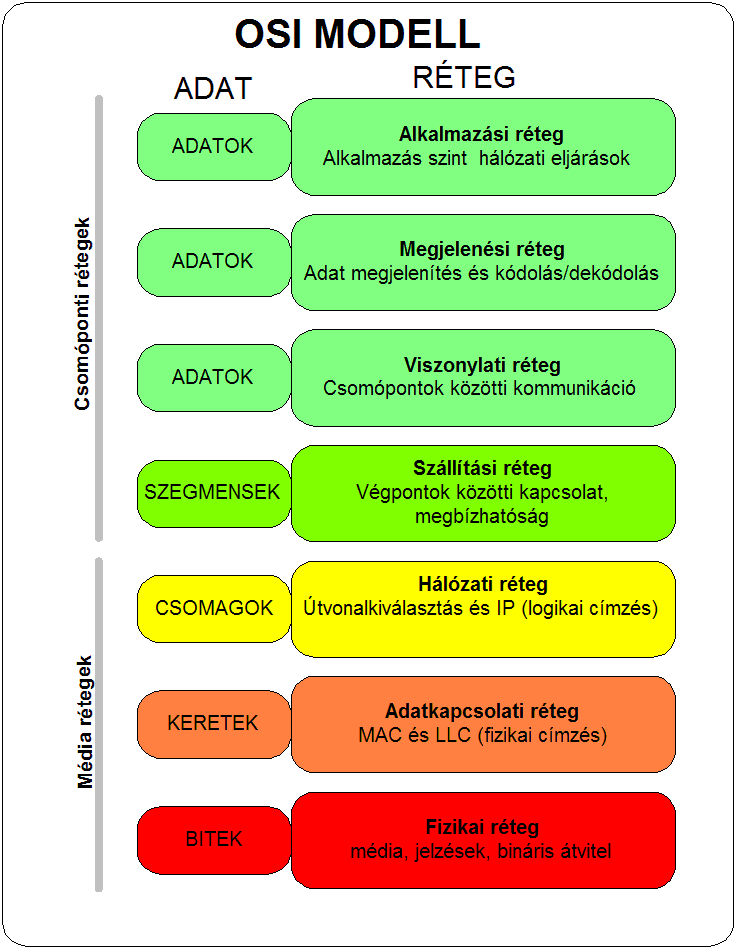


A rétegek között egyirányú interakció van, amennyiben amikor a felhasználó bevisz egy inputot, pl rákattint egy gombra, a prezentációs réteg üzeneteket küld az alatta lévő rétegnek, az applikációs rétegnek, ami pedig üzenetet küld az üzleti rétegnek, stb. Tehát a hívások lefelé történnek rétegenként.

Sok e-commerce és desktop applikáció ezt az architektúrát használja. Olyankor is hasznos, amikor gyorsan kell megépíteni egy alkalmazást. Jól tesztelhető és ellenőrizhető benne a standardnak megfelelés.

Ez a modell egyszerű lehetőséget biztosít arra, hogy egy jól szervezett applikációt írjunk. Viszont később komplex és költséges új rétegek hozzáadása vagy a meglévők felbontása.

# Hálózati referencia modellek



Az OSI-modell (angolul: Open Systems Interconnection Model) egy olyan elvi modell, amely egy távközlési vagy számítástechnikai rendszerek kommunikációs funkcióit jellemzi és szabványosítja, tekintet nélkül a mögöttes belső szerkezetre és technológiára. Célja a különféle kommunikációs rendszerek szabványos kommunikációs protokollokkal való átjárhatósága. A leírást gyakran az OSI hétrétegű modellje vagy OSI hivatkozási modell néven is emlegetik.

Az elnevezésben az Open Systems Interconnection (magyarul: Nyílt rendszerek összekapcsolása) arra utal, hogy nyílt rendszerek összekapcsolásával foglalkozik. A modell a kommunikációs rendszerben lévő adatáramlást hét absztrakciós rétegre osztja fel, a kommunikációs csatornán keresztüli bitek átvitelének fizikai megvalósításától kezdve az elosztott alkalmazások adatainak legmagasabb szintű megjelenítéséig.

## Fizikai réteg

A fizikai réteg feladata a bitek kommunikációs csatornára való juttatása. Ez a réteg határoz meg minden, az eszközökkel kapcsolatos fizikai és elektromos specifikációt, beleértve az érintkezők kiosztását, a használatos feszültség szinteket és a kábel specifikációkat. A szinten "Hub"-ok, "repeater"-ek és "hálózati adapterek" számítanak a kezelt berendezések közé. A fizikai réteg által megvalósított fő funkciók:

* felépíteni és lezárni egy csatlakozást egy kommunikációs médiummal.
* részt venni egy folyamatban, amelyben a kommunikációs erőforrások több felhasználó közötti hatékony megosztása történik. Például, kapcsolat szétosztás és adatáramlás vezérlés.
* moduláció, vagy a digitális adatok olyan átalakítása, konverziója, jelátalakítása, ami biztosítja, hogy a felhasználó adatait a megfelelő kommunikációs csatorna továbbítani tudja. A jeleket vagy fizikai kábelen – réz vagy optikai szál, például – vagy rádiós kapcsolaton keresztül kell továbbítani.

Paralell SCSI buszok is használhatók ezen a szinten. A számos Ethernet szabvány is ehhez a réteghez tartozik; az Ethernetnek ezzel a réteggel és az adatkapcsolati réteggel is együtt kell működnie. Hasonlóan együtt kell tudni működnie a helyi hálózatokkal is, mint például a Token ring, FDDI, és az IEEE 802.11.

## Adatkapcsolati réteg

Az adatkapcsolati réteg biztosítja azokat a funkciókat és eljárásokat, amelyek lehetővé teszik az adatok átvitelét két hálózati elem között. Jelzi, illetve lehetőség szerint korrigálja a fizikai szinten történt hibákat is. A használt egyszerű címzési séma fizikai szintű, azaz a használt címek fizikai címek (MAC címek) amelyeket a gyártó fixen állított be hálózati kártya szinten. Megjegyzés: A legismertebb példa itt is az Ethernet. Egyéb példák: ismert adatkapcsolati protokoll a HDLC és az ADCCP a pont-pont vagy csomag-kapcsolt hálózatoknál és az Aloha a helyi hálózatoknál. Az IEEE 802 szerinti helyi hálózatokon, és néhány nem-IEEE 802 hálózatnál, mint például az FDDI, ez a réteg használja a Media Access Control (MAC) réteget és az IEEE 802.2 Logical Link Control (LLC) réteget is.

Ez az a réteg, ahol a bridge-ek és switch-ek működnek. Ha helyi hálózat felé kell a kapcsolatot kiépíteni, akkor kapcsolódást csak a helyi hálózati csomópontokkal kell létrehozni, a pontos részleteket a „2.5 réteg” írja le.

A 2.5 réteg

Ez a réteg valójában nem része az eredeti OSI modellnek. A „2.5 réteg” kifejezés jelzi, hogy a kategóriába tartozó protokollok a 2-es és 3-as réteghez egyaránt kapcsolhatók. Ilyenek például a Multiprotocol Label Switching (MPLS) műveletek az adatcsomagokkal (2. réteg) illetve az IP protokoll címzése (3. réteg) amely speciális jelzéseket használ az útvonalirányítás során.

## Hálózati réteg

A hálózati réteg biztosítja a változó hosszúságú adat sorozatoknak a küldőtől a címzetthez való továbbításához szükséges funkciókat és eljárásokat, úgy, hogy az adatok továbbítása a szolgáltatási minőség függvényében akár egy vagy több hálózaton keresztül is történhet. A hálózati réteg biztosítja a hálózati útvonalválasztást, az adatáramlás ellenőrzést, az adatok szegmentálását/deszegmentálását, és főként a hiba ellenőrzési funkciókat. Az útvonalválasztók (router-ek) ezen a szinten működnek a hálózatban – adatküldés a bővített hálózaton keresztül, és az internet lehetőségeinek kihasználása (itt dolgoznak a 3. réteg (vagy IP) switch-ek). Itt már logikai címzési sémát használ a modell – az értékeket a hálózat karbantartója (hálózati mérnök) adja meg egy hierarchikus szervezésű címzési séma használatával. A legismertebb példa a 3. rétegen az Internet Protocol (IP).

## Szállítási réteg

A szállítási réteg biztosítja, hogy a felhasználók közötti adatátvitel transzparens legyen. A réteg biztosítja, és ellenőrzi egy adott kapcsolat megbízhatóságát. Néhány protokoll kapcsolat orientált. Ez azt jelenti, hogy a réteg nyomonköveti az adatcsomagokat, és hiba esetén gondoskodik a csomag vagy csomagok újraküldéséről. A legismertebb 4. szintű protokoll a TCP.

## Viszonylati réteg

A viszony réteg a végfelhasználói alkalmazások közötti dialógus menedzselésére alkalmas mechanizmust valósít meg. A megvalósított mechanizmus lehet duplex vagy félduplex, és megvalósítható ellenőrzési pontok kijelölési, késleltetések beállítási, befejezési, illetve újraindítási eljárások.

(A mai OSI modellben a Viszonylati réteg a Szállítási rétegbe lett integrálva.)

## Megjelenítési réteg

A megjelenítési réteg biztosítja az alkalmazási réteg számára, hogy az adatok a végfelhasználó rendszerének megfelelő formában álljon rendelkezésre. MIME visszakódolás, adattömörítés, titkosítás, és egyszerűbb adatkezelések történnek ebben a rétegben. Példák: egy EBCDIC-kódolású szöveges fájl ASCII-kódú szövegfájllá konvertálása, vagy objektum és más adatstruktúra sorossá alakítása és XML formába alakítása vagy ebből a formából visszaalakítása valamilyen soros formába.

A mai OSI modellben az Adatmegjelenítési réteg az Alkalmazási rétegbe lett integrálva. (A mai OSI ezért valójában 5 rétegű mivel a régi 7 rétegű modell 5. rétege a 4. illetve a 6. rétege a 7. rétegbe integrálódott. )

feladata:

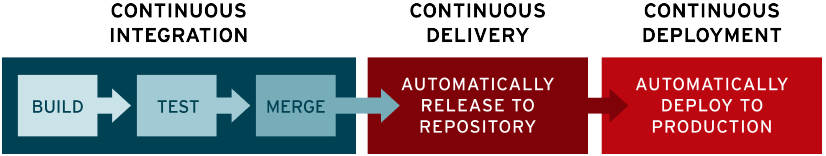
* két számítógép között logikai kapcsolat létesítése
* párbeszéd szervezése
* vezérjelkezelés
* szinkronizálás

## Alkalmazási réteg

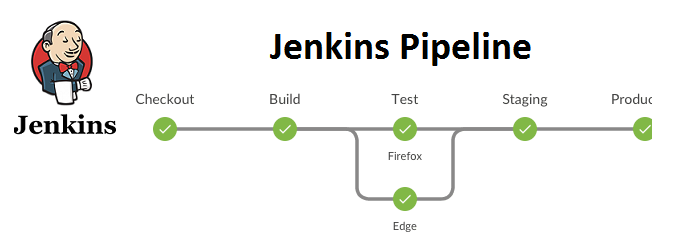
Az alkalmazási réteg szolgáltatásai támogatják a szoftver alkalmazások közötti kommunikációt, és az alsóbb szintű hálózati szolgáltatások képesek értelmezni alkalmazásoktól jövő igényeket, illetve, az alkalmazások képesek a hálózaton küldött adatok igényenkénti értelmezésére. Az alkalmazási réteg protokolljain keresztül az alkalmazások képesek egyeztetni formátumról, további eljárásról, biztonsági, szinkronizálási vagy egyéb hálózati igényekről. A legismertebb alkalmazási réteg szintű protokollok a HTTP, az SMTP, az FTP és a Telnet.

# Funkcionális csőrendszer

A pipeline egy folyamat, amely végigviszi a szoftver fejlesztését a megépítésen, tesztelésen és telepítésen. Úgy is ismert, mint CI/CD, continuous integration/continuous delivery. A folyamat automatizálásával a cél az, hogy az emberi hibákat minimalizáljuk és a (release) folyamatot konzisztensen és jól meghatározottan tartsuk. A benne résztvevő eszközök lehetnek fordítók, unit tesztek, kód analizáló toolok stb. Mostanában szeretik containerizálni. Általában a DevOps, az IT és a fejlesztés együtt csinálja.



Példa egy Jenkins pipeline-ra nagy vonalakban:



Példa egy Jenkins pipeline-ra részletesebben:

