12-Factor App módszertan

1. faktor: Codebase

A 12-Factor App módszertan első alapelve szerint egy alkalmazásnak **egy kódbázisa** legyen, amelyet **verziókezelő rendszerben** tárolunk. Ez a kódbázis lehet például egy Git repository. Ugyanebből az egyetlen kódbázisból több különböző környezetbe is történhet telepítés, például fejlesztői, teszt és éles környezetbe.

A legfontosabb gondolat tehát az, hogy **egy alkalmazás = egy kódbázis**, de ebből a kódbázisból lehet több deploy. A deploy itt azt jelenti, hogy ugyanazt az alkalmazást különböző környezetekben vagy konfigurációval futtatjuk.

Mit jelent ez a gyakorlatban?

Ha van egy webalkalmazásunk, akkor annak teljes forráskódja egyetlen repositoryban található. Ebből készülhet például:

- egy fejlesztői telepítés a programozók számára,
- egy staging rendszer tesztelésre,
- egy production rendszer a valódi felhasználóknak.

A három környezet nem három külön projekt, hanem ugyanannak az alkalmazásnak három külön futtatási példánya.

flowchart TD A[Egyetlen Git repository] --> B[Fejlesztői környezet] A --> C[Staging környezet] A --> D[Éles környezet]

Hibás gyakorlat

flowchart TD A[myapp-dev mappa] --> X[Fejlesztői futás] B[myapp-test mappa] --> Y[Teszt futás] C[myapp-prod mappa] --> Z[Éles futás]

Az első ábrán ugyanabból a kódbázisból indul a három környezet. A második ábrán már három külön másolat létezik, ami hosszú távon hibákhoz és eltérésekhez vezethet.

Jó megközelítés

- Egy alkalmazás forráskódja egy Git repositoryban van.
- A repository tartalmazza az alkalmazás összes szükséges forrásfájlját.
- Ugyanebből a repositoryból történik a fejlesztői, teszt és éles telepítés is.

Példa:

```
my-webapp/
  src/
  tests/
  package.json
  Dockerfile
  README.md
```

Ebben az esetben a `my-webapp` egyetlen alkalmazás, és egyetlen kódbázissal rendelkezik.

Rossz megközelítés

Nem jó, ha ugyanannak az alkalmazásnak több különálló, kézzel szinkronizált másolata van, például:

```
my-webapp-dev/
my-webapp-test/
my-webapp-prod/
```

Ez azért problémás, mert idővel a három változat eltér egymástól, és nem lehet biztosan tudni, hogy melyik az aktuális vagy helyes verzió.

Szintén rossz megoldás, ha egyetlen repositoryban több, egymástól független alkalmazás található úgy, hogy azok valójában külön életet élnek.

Miért fontos ez?

Ez az elv azért fontos, mert csökkenti a káoszt a fejlesztés és az üzemeltetés során. Ha egy alkalmazásnak több “félig azonos” kódbázisa van, akkor nagyon könnyen előfordulhat, hogy:

- a hibajavítás csak az egyik változatba kerül be,
- a tesztelt verzió nem egyezik meg az éles verzióval,
- nehézzé válik a verziókövetés,
- a csapat tagjai nem ugyanazzal a forráskóddal dolgoznak.

Az egyetlen kódbázis biztosítja, hogy mindenki ugyanarra az alapra építsen.

Példa 1: egyszerű webalkalmazás

Tegyük fel, hogy készítünk egy Python FastAPI alapú rendszert. A projekt repositoryja például ez:

```
invoice-service/
  app/
    main.py
    routes.py
  requirements.txt
  Dockerfile
```

Ebből ugyanabból a kódbázisból indulhat el:

- a fejlesztő gépén localhoston,
- a teszt szerveren,
- az éles Docker konténerben.

A különbség nem a forráskódban van, hanem a konfigurációban és a futtatási környezetben.

Példa 2: Node.js alkalmazás

Egy Node.js backend esetén is ugyanez az elv:

```
student-api/
  src/
  package.json
  package-lock.json
  .env.example
```

A fejlesztői, staging és production rendszer mind ugyanebből a repositoryból épül fel. Nem készítünk külön `student-api-prod` vagy `student-api-final` mappát.

Példa 3: amit sokan hibásan csinálnak

Kezdő projektekben gyakran előfordul, hogy valaki ezt csinálja:

```
projekt/
projekt_uj/
projekt_vegso/
projekt_vegso_jav/
projekt_vegleges_tenyleg/
```

Ez nem verziókezelés, hanem kézi másolgatás. A 12-Factor szemlélet szerint ezt Git repositoryval kell kiváltani, ahol a verziók commitokkal, branchekkel és tagekkel követhetők.

Kapcsolat a verziókezeléssel

A kódbázis elve szorosan kapcsolódik a verziókezeléshez. A legtipikusabb eszköz erre a Git. A fejlesztők ugyanazt a repositoryt használják, és ott külön ágakon dolgozhatnak. Ettől még maga az alkalmazás továbbra is egyetlen kódbázissal rendelkezik.

Tipikus félreértések

Sokan azt hiszik, hogy ha több microservice van, akkor az sérti ezt az elvet. Valójában nem, mert ilyenkor minden microservice külön alkalmazásnak számít, ezért mindegyiknek lehet saját kódbázisa.

Például:

- `user-service` → külön repository
- `order-service` → külön repository
- `payment-service` → külön repository

Ez teljesen helyes, mert ezek külön alkalmazások vagy komponensek.

---

2. faktor: Dependencies

A 12-Factor App módszertan második alapelve szerint egy alkalmazásnak **minden külső függőségét explicit módon deklarálnia kell**. Ez azt jelenti, hogy az alkalmazás nem támaszkodhat arra, hogy bizonyos könyvtárak vagy eszközök már telepítve vannak a rendszerben.

A függőségek deklarálása általában egy **függőségkezelő rendszer segítségével** történik. Így az alkalmazás pontosan meghatározza, hogy milyen külső csomagokra, könyvtárakra vagy frameworkökre van szüksége.

A cél az, hogy **az alkalmazás bármilyen környezetben ugyanúgy felépíthető és futtatható legyen**.

Mit jelent ez a gyakorlatban?

A modern alkalmazások gyakran több külső könyvtárat használnak, például:

- web framework
- adatbázis kliens
- JSON feldolgozó könyvtár
- autentikációs modulok

A 12-Factor elv szerint ezeknek a függőségeknek **mind szerepelniük kell a projekt konfigurációjában**.

Ha valaki letölti a projektet, akkor egyetlen paranccsal telepíthetővé kell válniuk a szükséges csomagoknak.

Jó megközelítés

Python példa:

`requirements.txt`

```
fastapi==0.110
uvicorn==0.29
pydantic==2.6
```

A projekt struktúrája például:

```
invoice-service/
  app/
  requirements.txt
```

Telepítés:

```
pip install -r requirements.txt
```

Ez biztosítja, hogy minden fejlesztő ugyanazokat a csomagokat használja.

Node.js példa

Node.js esetén a függőségek a `package.json` fájlban szerepelnek.

```
{
  "name": "student-api",
  "dependencies": {
    "express": "^4.18.2",
    "jsonwebtoken": "^9.0.0"
  }
}
```

Telepítés:

```
npm install
```

Ez automatikusan letölti az összes szükséges csomagot.

Java példa

Java projektekben gyakori a Maven vagy Gradle használata.

Maven példa:

```
pom.xml
```

```
<dependencies>
  <dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-web</artifactId>
  </dependency>
</dependencies>
```

A függőségek automatikusan letöltődnek a Maven repositoryból.

Rossz megközelítés

Hibás gyakorlat, ha a projekt implicit módon feltételezi, hogy bizonyos könyvtárak már telepítve vannak.

Példa:

```
# kódban használjuk

import fastapi
import pandas
```

de nincs:

```
requirements.txt
```

Ebben az esetben egy másik fejlesztő gépén az alkalmazás nem fog elindulni.

Klasszikus probléma

Sok fejlesztő találkozott már a következő hibával:

```
ModuleNotFoundError: No module named 'fastapi'
```

Ez tipikusan azért történik, mert a projekt függőségei nincsenek deklarálva.

Függőség izoláció

A 12-Factor elv gyakran együtt jár **virtuális környezetek használatával**.

Python példa:

```
python -m venv venv
source venv/bin/activate
pip install -r requirements.txt
```

Ez biztosítja, hogy a projekt saját csomagkészlettel rendelkezzen.

Konténeres példa

Docker használatakor a függőségek a Dockerfile-ban jelennek meg.

```
FROM python:3.11

WORKDIR /app
```

```
COPY requirements.txt .
RUN pip install -r requirements.txt

COPY . .
```

Ebben az esetben a konténer minden szükséges függőséget tartalmaz.

Miért fontos ez?

Az explicit függőségkezelés biztosítja, hogy:

- minden fejlesztő ugyanazt a környezetet használja
- az alkalmazás bármikor reprodukálható
- a build folyamat automatizálható
- a CI/CD pipeline stabilan működik

Ez különösen fontos nagy csapatoknál.

Gyakori félreértés

A függőségek deklarálása nemcsak programkönyvtárakat jelent.

Ide tartozhatnak például:

- build eszközök
- CLI segédprogramok
- fordítók
- runtime környezetek

Helyes működés

flowchart TD A[Git repository] --> B[Dependency file] B --> C[Automatikus telepítés] C --> D[Alkalmazás futtatása]

---

3. faktor: Config

A 12-Factor App módszertan harmadik alapelve szerint az alkalmazás **konfigurációját el kell választani a forráskódtól**. A konfigurációt nem a programkódban kell tárolni, hanem **környezeti változókban (environment variables)**.

A konfiguráció olyan értékeket jelent, amelyek **környezetenként változhatnak**, például:

- adatbázis kapcsolat
- API kulcsok
- jelszavak

- külső szolgáltatások URL-jei
- debug beállítások

A 12-Factor elv szerint ezek **nem lehetnek hardcode-olva a kódban**.

Mit jelent ez a gyakorlatban?

Ha egy alkalmazás több környezetben fut (például fejlesztői, teszt, éles), akkor a konfiguráció különböző lehet.

Például:

Környezet	Adatbázis
fejlesztői	localhost
teszt	test-db
production	prod-db

Ha az adatbázis címe a kódban van, akkor minden deploy előtt módosítani kellene a programot. A 12-Factor megközelítés szerint a konfiguráció **külső paraméterként kerül a programba**.

Rossz megközelítés (hardcode konfiguráció)

Python példa:

```
DATABASE_URL = "postgres://user:password@prod-db:5432/app"
```

Ebben az esetben:

- a jelszó a kódban van
- a konfiguráció commitolódik a repositoryba
- környezetváltáskor módosítani kell a kódot

Ez biztonsági és üzemeltetési problémákat okoz.

Jó megközelítés: környezeti változó

Python példa:

```
import os

DATABASE_URL = os.getenv("DATABASE_URL")
```

A környezetben:

```
export DATABASE_URL=postgres://user:password@prod-db:5432/app
```

Így a kód minden környezetben azonos marad.

Node.js példa

```
const dbUrl = process.env.DATABASE_URL;
```

Indítás:

```
DATABASE_URL=postgres://localhost/mydb node server.js
```

Docker példa

Docker konténer esetén a konfiguráció szintén environment változóként jelenik meg.

```
docker run -e DATABASE_URL=postgres://db/app myapp
```

Kubernetes példa

Kubernetesben a konfiguráció gyakran **ConfigMap** vagy **Secret** formájában jelenik meg.

```
env:
- name: DATABASE_URL
  valueFrom:
    secretKeyRef:
      name: database-secret
      key: url
```

Mi számít konfigurációnak?

Tipikus konfigurációk:

- adatbázis kapcsolatok
- külső API kulcsok
- SMTP szerver
- cache szerver
- feature flag-ek
- log szintek

Ami **nem konfiguráció**, az a program működési logikája.

Klasszikus hiba

Sok projektben található például ilyen fájl:

```
config.py
```

```
DATABASE_URL = "postgres://localhost/app"
SECRET_KEY = "123456"
```

Ha ezt commitolják, akkor:

- jelszavak kerülhetnek nyilvános repositoryba
- minden környezethez külön config fájl kell
- nő a hibalehetőség

Környezeti változók előnyei

Az environment változók használata több előnyt ad:

- nem kerülnek be a forráskódba
- deploy során könnyen módosíthatók
- biztonságosabb kezelés
- jól működik konténeres rendszerekben

Mermaid ábra: konfiguráció kezelése

flowchart TD A[Alkalmazás kód] --> B[Környezeti változó] B --> C[Konfiguráció betöltése] C --> D[Alkalmazás futása]

---

4. faktor: Backing Services

A 12-Factor App módszertan negyedik alapelve szerint az alkalmazás által használt infrastruktúra komponenseket **külső szolgáltatásként (backing service)** kell kezelni. Ezek olyan erőforrások, amelyek **nem az alkalmazás részei**, hanem külön rendszerek, amelyekhez az alkalmazás hálózaton keresztül kapcsolódik.

Tipikus backing service példák:

- adatbázis (PostgreSQL, MySQL)
- cache rendszer (Redis, Memcached)
- üzenetsor (Kafka, RabbitMQ)
- keresőmotor (Elasticsearch)
- objektumtároló (S3)

A fontos elv az, hogy ezek a szolgáltatások **cserélhető erőforrásként** jelenjenek meg az alkalmazás számára.

Mit jelent ez a gyakorlatban?

Az alkalmazás logikája nem tartalmazza magát az infrastruktúrát. Az adatbázis, cache vagy más komponensek külön szolgáltatásként futnak, és az alkalmazás csak kapcsolódik hozzájuk.

Például egy webalkalmazás használhat egy adatbázist és egy cache rendszert:

flowchart TD A[Alkalmazás] --> B[Adatbázis] A --> C[Cache]

Ebben a modellben az adatbázis és a cache külön szolgáltatások.

Cserélhetőség

A backing service elv egyik fontos következménye, hogy a szolgáltatás **könnyen lecserélhető**.

Például egy alkalmazás használhat:

- lokális PostgreSQL adatbázist fejlesztéskor,
- felhő alapú adatbázist éles környezetben.

Az alkalmazás működése ettől nem változik, mert az adatbázis külső szolgáltatásként jelenik meg.

Példa architektúra

flowchart TD A[Web alkalmazás] A --> B[(PostgreSQL)] A --> C[(Redis)] A --> D[(Message Queue)]

Az alkalmazás minden külső erőforrást külön szolgáltatásként kezel.

5. faktor: Build, Release, Run

Az ötödik faktor azt írja elő, hogy az alkalmazás életciklusát **három jól elkülönülő fázisra kell bontani**:

- **Build**
- **Release**
- **Run**

A három fázis szétválasztása azért fontos, mert így a telepítés **reprodukálható, automatizálható és biztonságos** lesz.

A három fázis jelentése

Build

A build fázis során a forráskódból egy **futtatható csomag (artifact)** készül.

Ebben a lépésben történik például:

- a forráskód lefordítása
- a függőségek telepítése
- a csomag vagy konténer image elkészítése

Példák:

- Java → JAR fájl
- Node.js → buildelt alkalmazás
- Docker → container image

Release

A release fázis a **build eredményének és a konfigurációnak az összekapcsolása**.

Ekkor jön létre egy konkrét alkalmazásverzió, amely telepíthető.

A release tehát:

build + konfiguráció

Run

A run fázis során az alkalmazás **futtatásra kerül a kiválasztott környezetben**.

Ez lehet például:

- szerver
- konténer
- cloud platform

A run fázis már **nem módosítja a buildet**, csak elindítja az alkalmazást.

A három fázis kapcsolata

flowchart LR A[Source Code] --> B[Build] B --> C[Release] C --> D[Run]

Miért fontos a szétválasztás?

Ha a három fázis nem válik el egymástól, akkor nehéz lesz:

- reprodukálni a telepítést
- visszaállni egy korábbi verzióra
- automatizálni a deploy folyamatot

A Build–Release–Run modell biztosítja, hogy **ugyanaz a build több környezetben is futtatható legyen**.

Példa Docker környezetben

Build:

```
docker build -t myapp:1.0 .
```

Release:

```
docker tag myapp:1.0 registry/myapp:1.0
```

Run:

```
docker run myapp:1.0
```

Példa CI/CD pipeline-ban

Egy modern pipeline gyakran pontosan ezt a három lépést követi.

flowchart LR A[Git commit] --> B[CI build] B --> C[Release artifact] C --> D[Deployment]

---

6. faktor: Processes

A hatodik faktor szerint az alkalmazás **stateless folyamatokból (processes)** álljon. Ez azt jelenti, hogy az alkalmazás futó példányai nem tárolhatnak tartós állapotot a saját memóriájukban vagy lokális fájlrendszerükben.

A tartós adatokat mindig **külső szolgáltatásban** kell tárolni, például adatbázisban vagy cache-rendszerben.

Mit jelent a stateless működés?

A stateless azt jelenti, hogy bármelyik futó példány képes kiszolgálni egy kérést anélkül, hogy előző kérések állapotát ismerné.

Ha az alkalmazás több példányban fut, akkor a rendszer bármelyik példányhoz irányíthatja a kéréseket.

Példa stateless architektúrára

flowchart TD A[Load Balancer] A --> B[App instance 1] A --> C[App instance 2] A --> D[App instance 3] B --> E[(Database)] C --> E D --> E

Ebben a modellben több alkalmazás példány fut egyszerre. Mindegyik ugyanahhoz az adatbázishoz kapcsolódik, és egyik sem tárol tartós adatot saját magában.

Rossz megközelítés: állapot a memóriában

Hibás megoldás, ha az alkalmazás a felhasználói állapotot a saját memóriájában tárolja.

Példa:

```
sessions = {}

def login(user):
    sessions[user.id] = "active"
```

Ha az alkalmazás több példányban fut, akkor az egyik példány memóriájában lévő állapot a másik példány számára nem lesz elérhető.

Jó megközelítés: állapot külső szolgáltatásban

A felhasználói állapotot külső rendszerben tároljuk.

Például Redis-ben:

```
flowchart TD
    A[App instance 1] --> B[(Redis)]
    C[App instance 2] --> B
```

Így minden alkalmazás példány ugyanazt az állapotot látja.

Fájlok kezelése

A stateless elv a fájlokra is vonatkozik.

Hibás megoldás:

```
/tmp/uploads/file1.jpg
```

Ha az alkalmazás új példányba kerül, a fájl eltűnhet.

Jó megoldás:

- objektumtároló (S3)
- hálózati fájlrendszer
- adatbázis

Miért fontos ez?

A stateless folyamatok lehetővé teszik:

- horizontális skálázást
- konténeres deploy-t
- automatikus újraindítást
- load balancing működését

Modern cloud környezetben

A cloud rendszerek gyakran automatikusan indítanak és állítanak le alkalmazás példányokat.

Ha az alkalmazás stateless, akkor ez nem okoz problémát.

---

7. faktor: Port Binding

A hetedik faktor azt mondja ki, hogy az alkalmazás **önálló szolgáltatásként publikálja magát egy porton keresztül**. Az alkalmazás saját HTTP vagy hálózati szervert indít, és ezen keresztül válik elérhetővé.

Ez azt jelenti, hogy az alkalmazás **nem függ külső web szervertől**, hanem maga biztosítja a szolgáltatás elérését.

Mit jelent ez a gyakorlatban?

Sok régebbi rendszerben az alkalmazás egy külső webszerverhez kapcsolódik.

Példa:

- Apache
- Nginx
- IIS

Ebben a modellben a webszerver tölti be az alkalmazást, például egy plugin vagy modul segítségével.

A 12-Factor megközelítés ezzel szemben azt javasolja, hogy az alkalmazás **saját szervert indítson**, és egy porton keresztül legyen elérhető.

Példa modern webalkalmazásra

Egy Node.js vagy Python webalkalmazás gyakran így indul:

```
app.listen(8080)
```

vagy

```
uvicorn main:app --port 8000
```

Az alkalmazás ekkor közvetlenül a megadott porton érhető el.

Architektúra példa

flowchart TD A[Client] --> B[Application server :8000]

Az alkalmazás saját szervert futtat, és közvetlenül fogadja a kéréseket.

Konténeres környezet

Konténeres rendszerekben ez különösen fontos.

Egy Docker konténer tipikusan egy porton szolgáltat.

Példa:

```
docker run -p 8000:8000 myapp
```

Az alkalmazás a konténeren belül a 8000-es porton fut.

Microservice architektúra

Microservice rendszerekben minden szolgáltatás saját porton publikálja magát.

flowchart TD A[API Gateway] --> B[User Service :8001] A --> C[Order Service :8002] A --> D[Payment Service :8003]

Minden szolgáltatás külön porton érhető el.

Miért fontos ez?

A port binding lehetővé teszi:

- szolgáltatások egyszerű indítását
- konténeres futtatást
- microservice architektúrát
- dinamikus infrastruktúrát

A platform (például Kubernetes vagy egy cloud szolgáltató) képes a szolgáltatásokat automatikusan összekapcsolni.

---

8. faktor: Concurrency

A nyolcadik faktor azt írja le, hogyan lehet az alkalmazást **skálázni**. A 12-Factor App módszertan szerint az alkalmazás **több folyamat indításával skálázható**, nem pedig egyetlen nagyobb folyamat erősítésével. Ez azt jelenti, hogy ha nő a terhelés, akkor **több azonos alkalmazás példányt** indítunk el párhuzamosan.

Mit jelent ez a gyakorlatban?

A skálázás két alapvető módja létezik:

- **vertikális skálázás** – egy erősebb szerver használata
- **horizontális skálázás** – több példány indítása

A 12-Factor szemlélet a **horizontális skálázást** támogatja.

Példa horizontális skálázásra

flowchart TD A[Load Balancer] A --> B[App instance 1] A --> C[App instance 2] A --> D[App instance 3]

A terheléselosztó (load balancer) a kéréseket több futó példány között osztja el.

Folyamat típusok

Az alkalmazások gyakran több különböző típusú folyamatot tartalmaznak.

Például:

- web szerver
- háttér feldolgozó (worker)
- időzített feladat (scheduler)

Ezek külön folyamatként futtathatók.

Példa architektúrára

flowchart TD A[Client] --> B[Web process] B --> C[(Database)] B --> D[Worker process]

A webfolyamat a kéréseket fogadja, a worker pedig háttérfeladatokat végez.

Példa worker skálázásra

Ha sok háttérfeladat érkezik, több worker indítható.

flowchart TD A[Queue] --> B[Worker 1] A --> C[Worker 2] A --> D[Worker 3]

Konténeres környezetben

Modern cloud rendszerekben a skálázás gyakran automatikus.

Példa Kubernetesben:

- 1 pod → kis terhelés
- 5 pod → nagy terhelés

A rendszer automatikusan indíthat új példányokat.

Miért fontos ez?

A több folyamat használata lehetővé teszi:

- nagyobb terhelés kezelését
- jobb hibakezelést
- rugalmas skálázást
- cloud infrastruktúra használatát

---

9. faktor: Disposability

A kilencedik faktor szerint az alkalmazás folyamatai **gyorsan induljanak el és gyorsan álljanak le**. Az ilyen folyamatokat könnyen lehet létrehozni vagy megszüntetni anélkül, hogy a rendszer működése megszakadna.

Ez a tulajdonság különösen fontos modern cloud- és konténeres környezetekben, ahol az alkalmazás példányai gyakran automatikusan indulnak és állnak le.

Mit jelent ez a gyakorlatban?

Ha egy alkalmazás új példányát kell indítani (például terhelésnövekedése miatt), akkor az **néhány másodperc alatt elinduljon**.

Ha pedig egy példányt le kell állítani (például frissítés vagy skálázás miatt), akkor az alkalmazás **biztonságosan befejezze a futó műveleteket**, majd leálljon.

Gyors indulás

A gyors indulás lehetővé teszi, hogy a rendszer új példányokat indítson el, amikor nő a terhelés.

flowchart TD A[Terhelés növekedés] --> B[Új alkalmazás példány indítása] B --> C[Új példány fogadja a kéréseket]

Gyors leállás

A folyamat leállításakor az alkalmazásnak lehetőséget kell adni a futó műveletek befejezésére.

Példa folyamat:

1. a rendszer jelzi a leállítást, 2. az alkalmazás befejezi az aktuális kéréseket, 3. a folyamat kilép.

Példa worker folyamatnál

Egy háttérfeldolgozó (worker) a leállítás előtt befejezheti az aktuális feladatot.

flowchart TD A[Stop jelzés] --> B[Folyamat befejezi az aktuális feladatot] B --> C[Folyamat leáll]

Miért fontos ez?

A gyors indulás és leállás lehetővé teszi:

- automatikus skálázást
- gyors deploy folyamatot
- hibás példányok cseréjét
- konténeres futtatást

Modern cloud-rendszerek gyakran indítanak új példányokat rövid idő alatt.

Példa konténeres környezetben

Egy Kubernetes-rendszer például új konténert indíthat, ha nő a terhelés. Ha egy példány hibás, a rendszer leállítja és újat indít.

10. faktor: Dev/Prod Parity

A tizedik faktor azt mondja ki, hogy a **fejlesztői (development), teszt (staging) és éles (production) környezetek között a különbség legyen minél kisebb**.

Minél jobban hasonlítanak egymásra a környezetek, annál kisebb az esélye annak, hogy az alkalmazás a fejlesztés során működik, de az éles rendszerben hibát okoz.

A klasszikus probléma

Sok rendszerben a fejlesztői és az éles környezet nagyon különbözik.

Például:

Fejlesztés	Production
SQLite adatbázis	PostgreSQL
lokális fájlrendszer	cloud storage
egyszerű szerver	több szerverből álló rendszer

Ilyenkor gyakran előfordul, hogy a program fejlesztés közben működik, de production környezetben hibát okoz.

Helyes megközelítés

A cél az, hogy a környezetek közötti különbség **minimális legyen**.

Ideális esetben:

- ugyanaz az adatbázis típus
- ugyanaz a futtatási környezet
- ugyanaz az infrastruktúra

Példa architektúrára

flowchart LR A[Development] --> B[Staging] B --> C[Production]

Mindhárom környezet ugyanazt az alkalmazást futtatja, csak más konfigurációval.

Konténeres megoldás

Docker használatával könnyen biztosítható a környezetek egyezése.

```
flowchart TD
    A[Docker Image] --> B[Dev környezet]
    A --> C[Test környezet]
    A --> D[Production környezet]
```

Ugyanaz az image kerül minden környezetbe.
