IdleGarageTycoon

Fejlesztési jegyzőkönyv (Backend + Frontend) – Dockeres futtatással

Készítette: Dudás Dominik
Neptunkód: PR2UEQ
Dátum: 2026.01.13.

Tartalom

1. Projekt áttekintés és célkitűzés

Az IdleGarageTycoon egy böngészőből futtatható, kliens–szerver architektúrájú webalkalmazás, amely egy egyszerű idle/tycoon játék mechanikájára épül.

A játékos egy autószerelő műhelyt irányít: munkákat indít, megvárja a lefutási időt, majd a jutalmat claim művelettel begyűjti.

A megszerzett pénzből fejlesztéseket (upgrade) vásárol, amelyek a következő munkák időtartamát csökkentik (Speed) vagy a jutalmat növelik (Reward).

A projekt célja egy átlátható, full-stack referencia megvalósítás: autentikáció (JWT), tartós adatkezelés (PostgreSQL + EF Core), REST API, React + TypeScript UI, valamint teljes Dockeres futtatás.

Fő játékmenet elemek:

- Regisztráció / bejelentkezés JWT tokennel.
- Workshop állapot lekérése (pénz, szint, exp, aktív munka).
- Munka indítása (Start job) és időalapú befejezés.
- Claim – jutalom jóváírása csak befejezett munka esetén.
- Upgrade vásárlás – pénzlevonás, szint növelés, következő ár számítása.

2. Követelmények

2.1 Funkcionális követelmények

- Regisztráció e-mail + jelszó alapon.
- Bejelentkezés e-mail + jelszó alapon.
- Sikeres auth után JWT token kiadása a kliensnek.
- Workshop (műhely) automatikus létrehozása regisztrációkor (1 user = 1 workshop).
- Munkák listázása és indítása (szintkövetelmény ellenőrzés).
- Egy időben csak egy aktív munka engedélyezett workshoponként.
- Claim csak akkor engedélyezett, ha a munka befejeződött.
- Upgrade vásárlás elérhető pénz esetén; hatása befolyásolja a munka időtartamát/jutalmát.
- Seedelt alap jobok és upgrade-ek, hogy a játék azonnal futtatható legyen.

2.2 Nem-funkcionális követelmények

- Biztonság – JWT Bearer autentikáció, védett endpointok (Authorize).
- Adatkonzisztencia – tranzakciók / Unit of Work a state-módosító műveleteknél.
- Skálázhatóság és olvashatóság – rétegezés (Controller → Service → Repository).
- Konfigurálhatóság – .env alapú beállítások (DB + JWT + frontend base URL).
- Könnyű futtathatóság – Docker Compose (frontend + backend + PostgreSQL).

3. Projektindítás és architektúra döntések

Célkitűzés

- Projekt scope rögzítése: idle játék, egy felhasználóhoz egy workshop.
- Technológiai stack kiválasztása és repository struktúra kialakítása.
- Dockeres futtatás célként kitűzése (frontend + backend + db).

Meghozott döntések

- Backend: ASP.NET Core Web API + EF Core + PostgreSQL.
- Auth: ASP.NET Core Identity + JWT Bearer.
- Frontend: React + TypeScript (Vite).
- Adatmodell GUID alapú kulcsokkal.
- Rétegezés: Controller → Service → Repository + Unit of Work.

Elvégzett feladatok

- Projekt skeleton létrehozása (backend és frontend).
- Alap entitások megtervezése (Workshop, JobDefinition, UpgradeDefinition).
- Kezdeti endpointlista meghatározása (state, start-job, claim, buy-upgrade, auth).

Eredmények / állapot

- Megszületett a végleges alap koncepció: időalapú munka + claim + upgrade rendszer.
- Kialakult az adatmodell és az API irányvonal (DTO-k, tiszta JSON contract).

Felmerült problémák és megoldások

- Kezdetben kérdés volt, hogy szükséges-e seed: döntés született, hogy a játék katalógusa (jobok/upgradek) HasData seeddel érkezik, így nincs admin panel igény.

Következő lépések

- EF Core modellek és kapcsolatok részletes implementációja.
- Auth (Identity + JWT) bevezetése.
- Workshop service logika megírása.

4. Backend: adatmodell, EF Core, migrációk, seed

Célkitűzés

- EF Core DbContext és kapcsolatok beállítása.
- Migrations futtatása és seedelt alapadatok rögzítése.
- Kapcsolati anomáliák és kulcsok tisztázása.

Meghozott döntések

- DbContext öröklődjön IdentityDbContext-ből (AppUser, IdentityRole<Guid>, Guid).
- Workshop.UserId egyedi index, így 1 user = 1 workshop biztosított.
- WorkshopUpgrade kulcsa: (WorkshopId, UpgradeDefinitionId).

Elvégzett feladatok

- IdleGarageDbContext implementálása és kapcsolatkonfiguráció.
- Seed adatok felvétele HasData-val (JobDefinitions + UpgradeDefinitions).
- Migráció létrehozása és adatbázis frissítése.

Eredmények / állapot

- A PostgreSQL adatbázisban létrejöttek a táblák (Identity + domain entitások).
- A seedelt jobok és upgrade-ek lekérdezhetők és a játék azonnal használható.

Felmerült problémák és megoldások

- Hiba: 'relation already exists' – korábbi táblák léteztek a DB-ben; megoldás: tiszta migration / database reset és újra update.
- Hiba: 'Guid should contain 32 digits...' – seedben hibás GUID stringek voltak; megoldás: minden HasData GUID 8-4-4-4-12 formátumra javítása.

Következő lépések

- Auth endpointok implementálása (register/login).
- Workshop állapot DTO és state endpoint.
- Service réteg bevezetése tranzakcióval.

5. Backend: Auth (Identity + JWT) és biztonság

Célkitűzés

- Regisztráció és bejelentkezés implementálása Identity-vel.
- JWT token generálás és validáció beállítása.
- Védett workshop endpointok engedélyezése csak bejelentkezett usernek.

Meghozott döntések

- JWT token tartalmazza a felhasználó azonosítóját (NameIdentifier claim).
- CORS engedélyezés lokális frontend originre.
- Swagger-ben Bearer auth definíció felvétele.

Elvégzett feladatok

- Program.cs konfiguráció: AddIdentityCore, AddEntityFrameworkStores, AddJwtBearer, AddCors, Swagger security.
- AuthController: /register és /login kialakítása.
- Register során Workshop létrehozása alapértékekkel (Money/Level/Exp/LastSeenAtUtc).

Eredmények / állapot

- A frontend képes JWT tokent kapni regisztráció/login után.
- Authorize attribútummal védettek a workshop műveletek.

Felmerült problémák és megoldások

- 400 Bad Request regisztrációnál (password policy): megoldás: frontend hibaüzenet kinyerése (errors lista), illetve jelszó szabályok kommunikálása.
- Régi token DB reset után: megoldás: token törlése/Logout és új user létrehozása; opcionálisan 'ensure workshop exists' logika.

Következő lépések

- Workshop state endpoint és DTO-k implementálása.
- Start job / claim / buy upgrade service metódusok.

6. Backend: Workshop játékmenet logika (Start/Claim/Upgrade)

Célkitűzés

- StartJob/Claim/BuyUpgrade üzleti szabályok implementálása.
- Upgrade hatások (Speed/Reward) számítása a munkákra.
- Tranzakciós és konzisztens adatmentés biztosítása.

Meghozott döntések

- A job befejezés ideje szerver oldalon kerül rögzítésre (CompletesAtUtc), a UI ezt használja visszaszámláláshoz.
- A jutalom 'RewardAtStart' mezőben rögzítésre kerül, így későbbi balansz változás nem írja felül a már futó jobot.
- Upgrade összhatás type-onként összegzett szintekkel számolható (Sum(Level)).

Elvégzett feladatok

- StartJobAsync: aktív job ellenőrzés, required level ellenőrzés, duration/reward kalkuláció, WorkshopJob létrehozás.
- ClaimAsync: csak completed job claimelhető, pénz hozzáadás és ClaimedAtUtc beállítás.
- BuyUpgradeAsync: nextCost ellenőrzés, pénzlevonás, level növelés / rekord létrehozás.
- Helper: GetUpgradeLevel(type) = upgrades.Where(...).Sum(Level).

Eredmények / állapot

- A backend API-val végigjátszható a game-loop: state → start → wait → claim → upgrade.
- A state válasz tartalmazza a szükséges UI adatokat (jobs/upgrades + active job).

Felmerült problémák és megoldások

- DbContext concurrency hiba ('A second operation was started...'): ok: repository SaveChangesAsync nem volt awaitelve / több SaveChanges hívás; megoldás: repository ne hívjon SaveChanges-t, csak UoW ment egyszer és minden async awaitelve.
- EF shadow property figyelmeztetés (UserId1): ok: hibásan beállított relationship; megoldás: explicit navigation (Workshop.User) + HasOne(w=>w.User) konfiguráció.

Következő lépések

- Frontend auth képernyők (Register/Login) és API kliens.
- Game UI: state megjelenítés, gombok, visszaszámláló.
- CSS: locked/unlocked, hover, disable állapotok.

7. Frontend: Auth, API kliens, hibakezelés

Célkitűzés

- React + TypeScript alap UI létrehozása: Register, Login, Game.
- Közös api.ts réteg: BASE URL env-ből, token kezelés, egységes error parse.
- Felhasználóbarát hibaüzenetek regisztrációnál/bejelentkezésnél.

Meghozott döntések

- Token tárolás localStorage-ban, logoutkor törlés.
- Hibakezelés: backend errors tömb összefűzése és status szerinti magyarázat (400/409/500).

Elvégzett feladatok

- .env beállítása frontend rootban: VITE_API_BASE_URL.
- api.ts módosítása: ne legyen 'undefined' base; fallback + részletes hibaszöveg.
- Register komponens: status-alapú magyarázó hiba panel bevezetése.

Eredmények / állapot

- A kliens képes regisztrálni/bejelentkezni és tokent tárolni.
- A hibák (pl. password policy) érthetően megjelennek a UI-ban.

Felmerült problémák és megoldások

- Hiba: 'POST http://localhost:5173/undefined/api/Auth/register' – ok: env változó nem töltődött; megoldás: VITE_API_BASE_URL és Vite restart.
- Hiba: 400 Bad Request register – megoldás: errors lista megjelenítése és jelszó formátum javítása.

Következő lépések

- Game oldal: state poll, start/claim/upgrade műveletek.
- Smooth visszaszámláló implementálása.
- UI stílusok (locked/unlocked gombok).

8. Frontend: Game UI, polling + smooth timer, stílusok

Célkitűzés

- Workshop state megjelenítése és interakciók (Start/Claim/Buy).
- Smooth visszaszámláló a CompletedAtUtc alapján.
- Gombok állapotkezelése (busy, locked, unlocked) és hover stílusok.

Meghozott döntések

- Polling 3 másodpercenként a backend felé (server truth).
- UI tick 1 másodpercenként a visszaszámlálóhoz (nem spameli az API-t).
- Upgrade gomb className váltása pénz alapján (locked/unlocked).

Elvégzett feladatok

- Game.tsx: state + err + busy kezelés, start/claim/buy metódusok.
- remainingSeconds számítása CompletedAtUtc alapján; tick state bevezetése.
- Gombok disabled logika: job lock level alapján; upgrade lock money alapján.
- Hover implementálás CSS class-szal (pl. logout-button:hover).

Eredmények / állapot

- Az aktív munka visszaszámlálása folyamatosan frissül a UI-ban.
- A lockolt upgrade-ek vizuálisan elkülönülnek és nem kattinthatók.

Felmerült problémák és megoldások

- UI nem frissítette az active jobot useMemo miatt – megoldás: tick state a rerenderhez, vagy backend remainingSeconds használata.
- Token DB reset után workshop hiány – megoldás: logout/új login.

Következő lépések

- Docker futtatás dokumentálása (compose, env, build).
- Tesztelési jegyzőkönyv (Swagger + UI).
- Végső összegzés és továbbfejlesztési javaslatok.

9. Docker: összefűzött repo, compose, futtatás

Célkitűzés

- Frontend és backend egy repositoryba integrálása.
- Teljes stack futtatása Docker Compose-szal (frontend + backend + PostgreSQL).
- Konfigurációk egységesítése .env változókkal.

Meghozott döntések

- Docker Compose a standard futtatási mód, lokális fejlesztés mellett.
- Alapadatok: EF Core migrációk alkalmazása indításkor (külön parancs vagy pipeline).

Elvégzett feladatok

- Repo struktúra: egy projekt alatt futtatható a teljes rendszer.
- Dockerfile-ok és compose beállítások elkészítése (build + port mapping).
- .env változók konszolidálása (PG_*, ConnectionStrings__*, Jwt__*, VITE_*).

Eredmények / állapot

- A teljes alkalmazás Dockerben indítható, külső függőség nélkül.
- Az adatbázis perzisztens volume-on tárolható (ha konfigurálva van).

Felmerült problémák és megoldások

- Külön dev/prod portok és base URL-ek eltérése: megoldás: külön env (dev: localhost portok, docker: service név / proxy).

Következő lépések

- Futtatási lépések rögzítése (docker compose up --build).
- Tesztjegyzőkönyv készítése (API + UI).

10. Tesztelés, tipikus hibák és megoldások

Célkitűzés

- A teljes game-loop validálása Swagger/curl és UI oldalról.
- Tipikus hibák gyűjtése és megoldási javaslatok rögzítése.
- Átadás előtti ellenőrzőlista elkészítése.

Meghozott döntések

- Teszteléshez 3 réteg: (1) Swagger, (2) curl, (3) UI flow.
- Hibaüzeneteket a frontend 'barátságos' formában jelenít meg.

Elvégzett feladatok

- Swagger authorize használata és endpointok manuális tesztje.
- Curl parancsokkal start-job és claim validálása.
- Frontend oldalon a countdown és lock/unlock logikák ellenőrzése.

Eredmények / állapot

- Validált game-loop: register/login → state → start-job → wait → claim → upgrade → state.
- Az állapotváltozások konzisztensen mentődnek DB-be.

Felmerült problémák és megoldások

- DbContext concurrency hiba korábbi implementációban – megoldás rögzítve (await + egy SaveChanges).
- Seed GUID hiba – megoldás: valid GUID stringek és migration újragenerálás.
- Régi token / új DB – megoldás: logout + új login; opcionális automatikus workshop létrehozás state-nél.

Következő lépések

- Dokumentum véglegesítése, mellékletek kitöltése (endpoint táblázat, env összefoglaló).

11. Összegzés és továbbfejlesztési lehetőségek

- Az IdleGarageTycoon teljesíti a kitűzött célokat: működő full-stack játéklogika, JWT alapú belépés, perzisztens PostgreSQL adatbázis és modern React UI.
- A rétegezett backend (service + repository + UoW) segíti a konzisztens üzleti szabályok implementációját és bővíthetőséget.
- A frontend polling + tick kombinációval kíméli a backendet, miközben a UI folyamatos visszajelzést ad a felhasználónak.

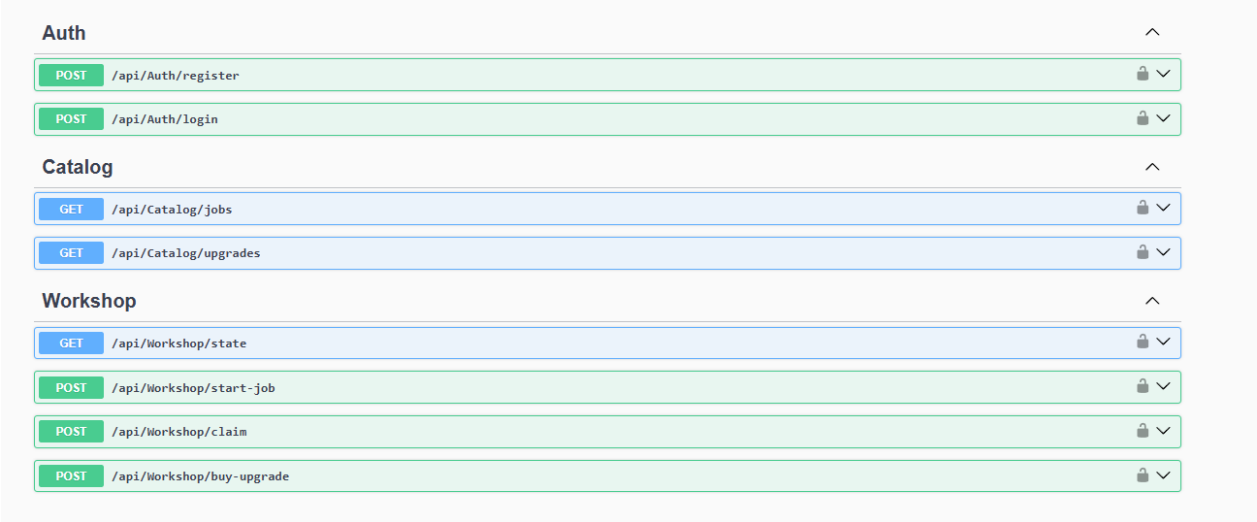
Továbbfejlesztési ötletek:

- Offline progression: LastSeenAtUtc alapján több munka automatikus elszámolása.
- Több upgrade típus (pl. Auto-claim, extra slot, kritikus jutalom esély).
- Balansz és szintezés: exp küszöbök, level-up jutalmak, több munkatípus.
- UI polish: progress bar, toast értesítések, reszponzívabb layout.

Melléklet A – Endpoint összefoglaló táblázat

Módszer	Útvonal	Leírás	Auth
POST	/api/Auth/register	Felhasználó regisztráció + workshop létrehozás + JWT token.	Anon
POST	/api/Auth/login	Bejelentkezés + JWT token.	Anon
GET	/api/Workshop/state	Műhely állapot + aktív munka + jobok + upgrade-ek.	Bearer
POST	/api/Workshop/start-job	Munka indítása jobDefinitionId alapján.	Bearer
POST	/api/Workshop/claim	Befejezett munka jutalmának begyűjtése.	Bearer
POST	/api/Workshop/buy-upgrade	Upgrade vásárlása upgradeDefinitionId alapján.	Bearer

Megjegyzés: a pontos request/response DTO-k a Swagger felületen is ellenőrizhetők.



Melléklet B – Konfigurációs változók (.env) összefoglaló

Backend (.env / környezeti változók):

- ConnectionStrings__DefaultConnection – PostgreSQL connection string.
- Jwt__Key – JWT aláíró kulcs (HMAC).
- Jwt__Issuer – token issuer (alapértelmezett: IdleGarageBackend).
- Jwt__Audience – token audience (alapértelmezett: IdleGarageFrontend).
- PG_HOST, PG_PORT, PG_USER, PG_PASSWORD, PG_DATABASE – adatbázis paraméterek (Docker esetén hasznos).

```
PG_USER=root
PG_PASSWORD=root
PG_DATABASE=idlegarage
PG_PORT=3254
PG_HOST=localhost


ConnectionStrings__DefaultConnection=Host=localhost;Port=3254;Database=idlegarage;Username=root;Password=root

Jwt__Issuer=IdleGarageBackend
Jwt__Audience=IdleGarageFrontend
Jwt__Key=VfBDQqXtCSxd5WAjK6ehJkEvHnTR79GU8wM4sNYarB3Rr8NcqvtdSxE2nzPbw5pV9mUjXQF6ZGWKMCLYuHaNxeuSGPRaXhJm8ZzCArD
```

Frontend (.env):

- VITE_API_BASE_URL – a backend API alap URL-je (pl. http://localhost:5026 dev környezetben).

```
VITE_API_BASE_URL=http://localhost:5026
```

Futtatás (példa):

- Fejlesztői módban: backend dotnet run, frontend npm run dev.
- Dockerben: docker compose up --build (a pontos fájlnevek és portok a repository beállításaitól függnek).