**RailPlot – Fejlesztői dokumentáció**

**Bevezetés**

Üdvözöljük a RailPlot fejlesztői dokumentációjában. Ez a dokumentum technikai betekintést nyújt a RailPlot alkalmazás belső működésébe, architektúrájába és kódmegoldásaiba. A célközönség elsősorban fejlesztők és rendszergazdák, akik tovább kívánják fejleszteni a projektet, integrálni szeretnék más rendszerekkel, vagy egyszerűen csak meg szeretnék érteni a kód struktúráját.

A RailPlot egy kétkomponensű webalkalmazás:

* **Frontend (kliensoldal):** HTML5 + CSS3 + JavaScript technológiákra épül, a felület kialakításához Bootstrap keretrendszert használva. A frontenden egy interaktív vászon (canvas) jeleníti meg a hálózatot, és JavaScript kezeli az összes felhasználói interakciót (űrlapok megjelenítése, kattintások, rajzolás).
* **Backend (szerveroldal):** PHP nyelven, Laravel keretrendszerben készült. A backend biztosítja a perzisztenciát (adatbázisban tárolja az állomásokat, vonalakat, felhasználókat, stb.), valamint egy REST API-n keresztül kommunikál a frontendel. Laravel beépített eszközeit kihasználva kezeli a felhasználóhitelesítést és jogosultságokat is.

A következőkben áttekintjük a projekt fontosabb részeit: hogyan épül fel az API, miként történik az adatok kezelése a szerveren, milyen megoldásokkal valósítottuk meg a speciális frontend funkciókat (pl. vászon kezelés, egyedi kurzor, modálok), illetve milyen fejlesztési mérföldköveken ment keresztül a projekt. Kitérünk a fájlformátumokra (JSON import/export), a tesztelési módszerekre és a jövőbeli fejlesztési lehetőségekre is. Kódrészletekkel illusztráljuk a kulcsfontosságú megoldásokat.

**Projekt áttekintés (architektúra és technológiák)**

**Frontend:** A felhasználói felület egy egyoldalas webalkalmazás jellegű megoldás (nem tradicionális SPA keretrendszerrel, hanem egyszerűbb JQuery-s/JavaScriptes megközelítéssel). A nézetet a Laravel szolgálja ki (Blade sablon vagy sima HTML), amely betölti a szükséges stíluslapokat (Bootstrap, CSS) és scripteket (Bootstrap, JavaScript, Axios, saját main.js). A frontend főbb összetevői:

* *HTML és Bootstrap:* A HTML váz tartalmazza a navigációs sávot, modális dialógus ablakok elemeit (pl. állomás űrlap, vonal űrlap), a canvast, a statisztikai panelt és a vonallista panelt. A Bootstrap segítségével gyorsan kialakítottuk a reszponzív elrendezést és az alapvető stílusokat (navbar, gombok, űrlapelemek).
* *JavaScript (ES6):* A kliensoldali logika tisztán JavaScript-kódban van megírva. Nincs használva Angular/React/Vue, mert a projekt mérete indokolta a közvetlen DOM-kezelést és canvas-manipulációt. A kód modulárisan van felépítve funkciók szerint (pl. külön funkció a rajzoláshoz, külön az eseménykezelésekhez). A modern ES6 szintaxis és funkcionalitások könnyebbé tették a kód írását (const/let, arrow function, stb.).
* *Canvas használat:* A HTML5 canvas elem felel a grafikus tartalomért (állomások és vonalak megjelenítéséért). A vászonra való rajzolást a JavaScript végzi a 2D context segítségével.
* *Axios könyvtár:* Az Axios egy HTTP kliens könyvtár, mellyel egyszerű AJAX kéréseket lehet intézni a backend felé. Ezt használjuk az API hívásokra (pl. állomások listájának lekérése, új állomás küldése a szervernek). Az Axios CDN-en keresztül kerül betöltésre a fejlécekben. Természetesen használható lenne a fetch API is, de az Axios a böngésző kompatibilitás és a beépített JSON konverzió miatt kényelmes.
* *Adattárolás a frontenden:* A legtöbb művelet valós időben történik a memóriában lévő JavaScript objektumokon (stations, lines, trains tömbök). Ezek a tömbök a backendről letöltött adatokat tartalmazzák, illetve a felhasználó által bevitt módosítások itt követhetők nyomon azonnal. A canvas frissítése mindig ezen tömbök aktuális állapotát veszi alapul.

**Backend:** Laravel keretrendszer 8.x/9.x verzióra épül (a pontos verzió a composer.json-ből ellenőrizhető). A backend fő feladatai:

* *REST API biztosítása:* Az alkalmazás RESTful API végpontokat kínál, például:
  + GET /api/stations – az összes állomás lekérdezése.
  + POST /api/stations – új állomás létrehozása.
  + PUT /api/stations/{id} – meglévő állomás módosítása.
  + DELETE /api/stations/{id} – állomás törlése.
  + Hasonlóan léteznek végpontok a /api/lines alatt a vonalak kezelésére, valamint az /api/trains alatt a vonatokra is.

Ezek az endpointok JSON formátumban várják/adják vissza az adatokat. A Laravel Route rendszerében jellemzően egy Route::apiResource definiálással valósítottuk meg a CRUD műveleteket, vagy egyedi route-okkal, a megfelelő Controller metódusokra mutatva. Például az állomások lekérése és felvétele:

php

Másolás

Route::middleware('auth:sanctum')->group(function() {

Route::get('/api/stations', [StationController::class, 'index']);

Route::post('/api/stations', [StationController::class, 'store']);

// ... további station műveletek

});

A fenti megoldás feltételezi, hogy a Sanctum vagy Passport csomag gondoskodik az API autentikációról, de akár session alapú auth is használható, mivel a frontendet ugyanaz a domain szolgálja ki.

* *Adatbázis kezelés (Eloquent ORM):* Laravel Eloquent-et használunk a relációs adatbázis kezelésére. A főbb modellek:
  + **User** – Felhasználók táblája (Laravel alap auth struktúra, tartalmazza a nevet/emailt/jelszót, plusz egy is\_admin flag az admin jogosultság jelölésére).
  + **Station** – Állomások modellje. Mezői: id (kulcs), name (string), x (int), y (int), location (esetleg string vagy JSON mező a típusokra), user\_id (opcionális, ha felhasználóhoz kötjük az állomást).
  + **Line** – Vonalak modellje. Mezői: id, name, code, color, type, user\_id stb. A vonal és állomás kapcsolatot kétféleképpen lehet megoldani:
    1. **Kapcsolótábla (pivot) használata:** Létrehoztunk egy line\_station táblát, amely sorolja, hogy mely állomás mely vonalhoz tartozik, és tartalmaz egy order mezőt a sorrend megőrzésére. Eloquentben a Line modell stations() kapcsolata belongsToMany(Station::class)->withPivot('order')->orderBy('order') lehet. A Station modell hasonlóan lines() belongsToMany kapcsolatban áll. Ez a reláció lehetővé teszi, hogy lekérdezzük egy vonal összes állomását a megadott sorrendben, illetve egy állomás összes vonalát.
    2. **JSON mező használata:** Alternatív megoldásként (gyors prototípusként) a Line modellnek lehet egy stations mezője, amely a benne levő állomásneveket vagy ID-kat tartalmazza JSON listaként. Ez egyszerűsíti a mentést/leérkezést (a frontenden pont ilyen struktúrában van), viszont nem normálformájú megoldás, és bonyolítja az esetleges lekérdezéseket (pl. nehezebb egy SQL-lekérdezéssel megállapítani, egy állomás mely vonalak része). A fejlesztés során az első megközelítés a JSON mezős volt az egyszerűség miatt. A későbbi fejlesztési terv az áttérés a normalizált, pivot táblás megoldásra.
  + **Train** – Vonatok modellje. Mezői: id, name, type, quantity. Ezt a modellt az admin kezeli (felviszi, mennyi jármű elérhető adott típusból). A Line modellek tartalmaznak egy assigned\_train\_id és train\_quantity mezőt, utóbbi jelzi, hogy a vonalon hány járművet használunk. Az alkalmazás logikája figyel arra, hogy train\_quantity <= a hozzárendelt Train modell quantity mezője.
* *Üzleti logika a backendben:* A legtöbb szabályt (unique nevek, min két állomás egy vonalban, stb.) a frontend már ellenőrzi, de a backendben is érdemes ezeket validálni (soha ne bízzunk kizárólag a kliensben). Laravelben Form Request osztályokkal vagy manuális Validator hívásokkal meg is történik:
  + Station létrehozásnál pl. name kötelező és egyedi (a megadott felhasználó kontextusában), x,y szám típusú, location pedig JSON formátumú mező validációja (tartalmazzon legalább egy típust).
  + Line létrehozásnál pl. code 2 hosszú és egyedi, stations tömb legalább 2 elemmel, type értéke in:ground,underground,suspended, assigned\_train\_id létező vagy null, stb.
  + Ezeket a validációs szabályokat a StationController@store/update és LineController@store/update metódusaiban definiáltuk.
* *Válaszküldés:* Az API végpontok jellemzően JSON-t adnak vissza. Sikeres műveleteknél a létrehozott/aktualizált objektumot (vagy listát), hibánál hibaüzenetet (422-es státusz például validációs hibáknál). Példa: egy új állomás létrehozásának hívása Axios-szal:

js

Másolás

axios.post('/api/stations', {

name: "Central Station",

x: 250, y: 400,

location: ["ground", "underground"]

})

.then(response => {

console.log("New station saved:", response.data);

})

.catch(error => {

console.error("Error adding station:", error);

});

A response.data tartalmazni fogja a létrehozott állomás objektumot (pl. id-val együtt). A fenti példa mutatja egy API végpont használatát kliensoldalról JavaScript segítségével.

* *Authentikáció és jogosultságok:* A Laravel backend a felhasználókezelést két módon nyújthatja:
  + **Web guard (session alapú):** A bejelentkezés HTML formon keresztül Laravel session-t hoz létre. Az API végpontokat is be lehet így jelentkezve hívni (hisz ugyanarról a domainről jön a kérés, a session cookie küldésre kerül).
  + **API guard (token alapú, pl. Sanctum):** A modernebb megoldás a Sanctum használata, ahol a frontenden kapott tokennel hívja meg az Axios az API-kat, a headerben egy Authorization: Bearer <token> fejléccel. Ezt implementálhatjuk is, ha teljesen szeparálni akarjuk a frontendet. A RailPlot esetében azonban egyszerűbb volt a hagyományos bejelentkezést alkalmazni, és a böngésző-session mechanizmust használni.

Az admin jogosultság ellenőrzése például történhet middleware-rel (pl. is\_admin flag vizsgálata egy AdminMiddleware osztályban), ha egyes végpontokat/oldalakat korlátozni akarunk. Tipikusan az adminisztrációs panel route-jait védjük így. A normál API műveleteknél pedig azt nézzük, hogy a kérés user-je az adott erőforráshoz hozzáfér-e (pl. a StationController@index csak a bejelentkezett user saját állomásait adja vissza, ha a rendszert több user adatainak szeparációjára tervezzük).

Összességében az architektúra egy klasszikus kliens-szerver modell: a felhasználó böngészője a klienst futtatja, ami AJAX kérésekkel kommunikál a Laravel alapú szerverrel. A két oldal laza kapcsolatban áll – ha kell, a frontenden offline módban is lehet szerkeszteni és utólag szinkronizálni (a JSON export/import is egyfajta manuális szinkronizálás). A következőkben részletesebben áttekintjük a backend és frontend specifikumait.

**Backend részletezése**

**Laravel alapú API kialakítása**

A RailPlot backendjén az API-k kialakításakor a Laravel nyújtotta lehetőségeket használtuk:

* **Route definíciók:** Az routes/api.php vagy routes/web.php fájlban kerültek meghatározásra. A REST konvenciókat követve a StationController és LineController tartalmazza a CRUD műveleteket. Egyszerűségképp a projektben:
  + StationController@index adja vissza a bejelentkezett felhasználó állomásait JSON-ban.
  + StationController@store új állomást hoz létre (a kérést validate-eli, menti Eloquent-tel, visszaadja JSON-ként).
  + StationController@update meglévő állomást módosít.
  + StationController@destroy állomást töröl (előtte ellenőrzi, nincs-e hozzárendelve vonalhoz, különben hibát dob).

Hasonlóan LineController kezeli a vonalakat. A vonal létrehozásakor külön figyelmet érdemel az állomáslista: itt két lehetőséget implementálhatunk:

* + **Egyszerű megoldás:** a request-ben a vonal stations mezője tartalmaz egy tömböt az állomás ID-kkal. A LineController@store ezt a tömböt elmentheti JSON-ként a line rekord stations mezőjébe. (Ekkor az Eloquent modellekben be kell állítani, hogy a stations mezőt castolja array típusra, így JSON-ként menti.)
  + **Komplexebb megoldás pivot-tal:** a LineController@store előbb létrehozza a line rekordot, majd a request stations tömbje alapján feltölti a pivot táblát. Például:

php

Másolás

$line = Line::create([...]); // vonal alapadatok mentése

foreach($request->stations as $index => $stationId) {

$line->stations()->attach($stationId, ['order' => $index]);

}

Hasonlóan update-nél detach/attach kombinációval frissíthetjük a hozzárendelést. Ez bonyolultabb, de relációsan helyesebb.

A RailPlot prototípusban eleinte az egyszerűbb JSON-mezős módszer valósult meg, de a dokumentációban is javasoljuk a pivotos verziót hosszú távon, különösen ha a projekt növekszik.

* **Felhasználókezelés a backendben:** A Laravel Breeze vagy Fortify csomaggal gyorsan kialakítható a regisztráció/bejelentkezés funkció. A projektben a standard Laravel auth rendszert használtuk:
  + users tábla migrációja alap, kiegészítve egy is\_admin boolean mezővel.
  + A regisztráció során alapértelmezetten is\_admin = false. Később manuálisan az adatbázisban, vagy admin panelen keresztül lehet egyes felhasználókat adminná tenni.
  + A bejelentkezés után a Laravel session tárolja a user azonosítóját.
  + Minden API végpont előtt egy auth middleware fut, ami biztosítja, hogy csak bejelentkezett felhasználó érhesse el (pl. a routes definíciónál csoportosítva).
  + Admin jogosultság ellenőrzése a kritikus műveleteknél: pl. vonat létrehozását/törlését csak admin végezheti (ezt vagy middleware-rel, vagy a controllereken belül if feltétellel kezeljük).
  + Például:

php

Másolás

if(Auth::user()->is\_admin) {

// engedélyezett művelet

} else {

return response()->json(['message' => 'Forbidden'], 403);

}

Ugyanígy használható a Laravel Gate/Policy mechanizmus a szebb megoldáshoz.

* **Adminisztrációs felület:** A backend tartalmaz egy admin route csoportot is (tipikusan /admin prefix alatt, blade nézetekkel). Itt az admin megnézheti a felhasználók listáját, vonatok készletét módosíthatja, vagy akár karbantartási feladatokat végezhet (pl. hálózatok exportálása). Ez a rész a projekt későbbi fázisában került bevezetésre, és nem feltétlen SPA, használhat nyers blade + form megoldásokat (mivel admin kevésbé igényli a fancységet, és fontosabb a stabil funkcionalitás).

Összességében a Laravel backend gondoskodik róla, hogy az adatok biztonságosan, konzisztensen legyenek kezelve. Minden művelet naplózható is (opcionálisan a későbbiekben bevezethető egy log rendszer, pl. Monologgal, ha szükséges audit trail).

**Felhasználókezelés és jogosultságok**

Már érintettük a felhasználókezelést, itt összefoglaljuk:

* **Regisztráció és bejelentkezés:** Laravel alapértelmezett implementáció. A jelszavak bcrypt-tel hashelve tárolódnak. Email címmel lép be a user. (Opcionálisan email megerősítés bevezethető, de jelen projektben nem volt elsődleges.)
* **Session vs API token:** A RailPlot esetében, mivel a frontendet a Laravel szolgálja ki és ugyanazon domain alatt fut, egyszerű volt a session használata. Ha az alkalmazást később natív mobilappal is elérhetővé tennénk, akkor szükség lenne token alapú authentikációra (Laravel Sanctum pl. egyszerre támogat session cookie és API token auth-ot).
* **Admin szerepkör:** Az admin jogkör koncepciója az, hogy bizonyos műveleteket csak admin végezhet:
  + Vonat állomány kezelése (új típus felvétele, mennyiség módosítás) – nehogy a felhasználók csak úgy „hozzávarázsoljanak” maguknak végtelen járművet.
  + Globális statisztikák megtekintése – pl. összes felhasználó összes állomása (ha multi-user a rendszer).
  + Rendszerkarbantartás – pl. minden adat törlése (Clear Grid globálisan), stb.

A is\_admin mező egyszerű ellenőrzésével minden controllerben lehet if-ekkel védelmet rakni. Szebb megoldásnak említendő a Policy-k használata: készíthetünk pl. StationPolicy, LinePolicy osztályokat, ahol a delete metódus például így nézhet ki:

php

Másolás

public function delete(User $user, Station $station) {

// Csak az törölheti, akié az állomás vagy admin

return $user->id === $station->user\_id || $user->is\_admin;

}

Majd a StationController@destroy-ban annyit kell tenni, hogy $this->authorize('delete', $station);. Ez automatikusan dob 403-at, ha nincs jogosultság.

Mivel a projektben minden állomást/vonalat a tulajdonosa szerkeszthet, gyakorlatilag minden műveletnél ellenőrizni kellett, hogy a bejelentkezett user megegyezik-e a resource user\_id-jával, vagy admin-e. Ezt elegánsan egy global scope vagy Policy rendszeren keresztül lehetett megoldani. A dokumentáció kedvéért jegyezzük meg, hogy jelenlegi implementációban néhol explicit ellenőrzések voltak, de a jövőben mindenképp érdemes a Laravel beépített jogosultság-kezelését használni a karbantarthatóság miatt.

**Állomások és vonalak CRUD műveletei**

**Állomás CRUD:**

* *Create (store):* A StationController@store validálja a kérést (unique name, required fields), majd Station::create-tel létrehozza az állomást. Az állomás location mezőjét a backend JSON formátum helyett akár egy stringben is tárolhatná, de a mi megoldásunk az volt, hogy egyszerűen JSON-ként tároljuk egy TEXT mezőben (pl. '["ground","underground"]'). Ehhez a Station modelben protected $casts = ['location' => 'array'] van, így Eloquent automatikusan array-ként kezeli.
* *Read (index/show):* StationController@index tipikusan a bejelentkezett user összes állomását adja vissza: return Station::where('user\_id', Auth::id())->get();. Ha admin az user, dönthetünk úgy, hogy mindenki állomását visszaadja vagy csak az övét – jelen projekt esetén a felhasználók munkája szeparált, így admin is alapból csak a sajátját látná a fő UI-ban, de az admin panelen elérheti másokét is.
* *Update:* StationController@update hasonlóan validál, majd $station->update($request->only(['name','location'])); (X, Y nem változik itt, hacsak nem engedjük mozgatni). Ha név változott, a backend is figyelhetne arra, hogy a vonalak stations listájában módosítsa a nevet – de mivel a mi tárolási megoldásunknál a vonal stations listája is JSON, ez komplikált lehet. Itt látni a pivot megoldás előnyét: ha pivotban tároljuk, állomás átnevezése nem érinti a pivotot (ott csak station\_id van), így a vonalakat nem kell módosítani. A jelenlegi implementációban az állomás átnevezés a frontenden is megoldásra került: a vonal szerkesztésnél, amikor újra mentjük, már az új nevet menti a JSON-ba. A backend részen ezt külön nem kezeltük, feltételezzük hogy a frontenden konzisztensen tartják az adatokat. (Ez egy fontos pont: a jövőben célszerű áttérni station\_id hivatkozásokra a front-back kommunikációban is.)
* *Delete:* StationController@destroy előbb megnézi, hogy az állomás nem tartozik-e vonalhoz. Ha JSON-ban van tárolva a vonal->stations, akkor végig kell nézni minden vonalat, tartalmazza-e az adott station name-et, ami nem hatékony. Mi a frontendre bíztuk ezt a logikát (ld. felhasználói doksi: kliens oldalon le van kezelve, hogy ne is próbálja a user törölni állomást, ha vonal része). De ettől függetlenül backend oldalon sem árt egy ellenőrzés:

php

Másolás

$inLines = Line::whereJsonContains('stations', $station->name)->exists();

if($inLines) return response()->json(['error' => 'Station part of a line'], 409);

$station->delete();

Pivot megoldás esetén sokkal egyszerűbb: $station->lines()->exists() vizsgálat, és ha van, akkor is törölhető, de akkor előbb detach-olni kéne (vagy tiltani). A mi üzleti szabályunk tiltja, úgyhogy nem töröljük, ha van line kapcsolata.

**Vonal CRUD:**

* *Create (store):* LineController@store előbb validál (unique code per user, 2 char code, stations array min 2 elem, type required stb.). Ezt követően létrehozza a Line objektumot. Ha pivot rendszert használunk, utána az állomásokat csatoljuk. Ha JSON rendszert, akkor a request-ből jövő stations tömböt beletesszük a line rekord stations mezőjébe. Nálunk pl.:

php

Másolás

$line = Line::create([

'name' => $req->name,

'code' => strtoupper($req->code),

'color' => $req->color,

'type' => $req->type,

'stations' => $req->stations, // cast -> JSON

'assigned\_train\_id' => $req->assignedTrain ?? null,

'train\_quantity' => $req->trainQuantity ?? 0,

'user\_id' => Auth::id()

]);

Visszatérésként return response()->json($line, 201) küldhető.

* *Read (index/show):* Hasonlóan mint station-nél, a bejelentkezett user vonalait adjuk vissza. A vonal JSON mezőit is szépen visszaadja (Laravel automatikusan assoc array-ként adja majd a JSON response-ban).
* *Update:* Validál és frissít. Ha code változik, unique check (kivéve saját magán). Az állomáslista frissítése megint kritikus: pivot esetén detach all majd attach új sorrendben; JSON esetén simán $line->stations = $req->stations és utána $line->save(). A vonat hozzárendelést is frissítjük. Itt is figyelni kell a következményekre: pl. ha egy állomás kimarad, annak most már semmi nem jelzi a vonalhoz tartozását. Pivotnál oké, JSON-nél is oké, hisz kivettük a nevét. A statisztikákat a frontendre bízhatjuk.
* *Delete:* Egyszerű, $line->delete(). Kapcsolódó állomások pivot táblában automatikusan törlődhet, ha a relation cascade-re van állítva. JSON-nél semmi extra teendő. (Esetleg ha vonat volt hozzárendelve, dönthetünk úgy, hogy felszabadítjuk, de a vonat qty jellegű nyilvántartása független a konkrét vonaltól.)

Fontos megjegyezni, hogy a backend CRUD műveleteknél a projekt fejlesztése során sok logikát implementáltunk a frontenden is (duplikálva a validációt, szabályokat). Ez tudatos döntés volt a felhasználói élmény miatt (azonnali visszajelzés), de a backenden is ott kell legyen minden kritikus ellenőrzés biztonsági okból.

**Frontend részletezése**

Most nézzük a kliensoldali kód néhány érdekesebb aspektusát:

**Vászon (Canvas) kezelése**

Az egész hálózati rajz a <canvas id="canvas"> elemre történik. A canvas méretét a JavaScript dinamikusan állítja be a böngészőablak méretéhez:

js

Másolás

const canvas = document.getElementById('canvas');

const ctx = canvas.getContext('2d');

function resizeCanvas() {

canvas.width = window.innerWidth;

canvas.height = window.innerHeight - 50; // 50px a navbar magassága

}

window.addEventListener("resize", resizeCanvas);

resizeCanvas();

Ez biztosítja, hogy mindig teljes területű rajzfelületünk legyen (a navigációs sáv alatti hely kitöltése).

**Rajzolási logika:**

* Definiáltunk egy update() függvényt, amely újrarajzolja a teljes vásznat. Pseudocode:

js

Másolás

function update() {

ctx.clearRect(0, 0, canvas.width, canvas.height);

drawGrid(); // opcionális: háttérrács rajzolása

lines.forEach(drawLine); // minden vonalat kirajzol

stations.forEach(drawStation); // minden állomást kirajzol

requestAnimationFrame(update); // folyamatos frissítés (animáció keret)

}

A fenti séma folyamatos animáció keretében rajzol, ami akkor indokolt, ha animálunk (pl. mozgatunk objektumokat). Mivel a RailPlot-ban nincsen folyamatos mozgás, más megközelítést is alkalmazhatunk: csak akkor hívjuk meg az update()-et, amikor változás történt. A projekt kódjában vegyesen látható: bizonyos helyeken requestAnimationFrame-fel loopol, de valójában egy-egy eseménynél is meghívjuk. Ez nem túl hatékony, de kisebb adatmennyiségnél nem probléma. A jövőben finomítható.

* **Állomások rajzolása (drawStation):** Tipikusan egy kis kör (pl. radius 5px) és egy felirat. A kód:

js

Másolás

function drawStation(station) {

ctx.fillStyle = "red"; // minden állomás piros pötty

ctx.beginPath();

ctx.arc(station.x, station.y, 5, 0, 2 \* Math.PI);

ctx.fill();

ctx.fillStyle = "black";

ctx.fillText(station.name, station.x + 8, station.y + 3);

}

(A valós kódban a szín/forma akár állomás típusonként is változhatna; pl. lehetne külön ikon a földalatti állomásoknak. Jelen verzió ezt nem differenciálta, minden állomást egységesen jelöl.)

* **Vonalak rajzolása (drawLine):** A vonalakat megszakított szakaszok helyett a lines tömbben tárolt állomásnév-lista alapján kell megrajzolni. Kód:

js

Másolás

function drawLine(line) {

if(line.stations.length < 2) return;

ctx.strokeStyle = line.color || "#000";

ctx.lineWidth = 4;

ctx.beginPath();

// Kiinduló pont első állomás koordinátája:

const firstStation = stations.find(s => s.name === line.stations[0]);

ctx.moveTo(firstStation.x, firstStation.y);

// Végigmenni a többi állomáson és vonalat húzni oda

for(let i = 1; i < line.stations.length; i++) {

const station = stations.find(s => s.name === line.stations[i]);

ctx.lineTo(station.x, station.y);

}

ctx.stroke();

}

A fenti logika egy törött vonalláncot összekötő vonalat rajzol. Ha azt szeretnénk, hogy minden szakasz külön színnel vagy stílussal legyen (pl. ha más típusú kapcsolat), tovább lehetne bontani. A stations.find(s => s.name===...) nem túl hatékony (O(n) minden állomás megkeresése), de kevés adatnál elmegy. Javítható egy map létrehozásával name->coords egyszer, vagy eleve station id-k tárolásával (akkor index alapján is elérhető).

* **Grid rajzolása:** Van egy opcionális funkció, hogy a háttérben látszódjon egy koordináta rács (segíthet tájékozódni a vásznon). A kódban drawGrid() valósítja ezt meg, adaptív módon (nagyításnál több-kevesebb vonal). Ez egy fejlettebb része a kódnak:

js

Másolás

function drawGrid(gridScreenSize = 128) {

// adaptív grid, scale figyelembe vételével

const scale = 1 / panZoom.scale;

const gridSize = gridScreenSize \* scale;

ctx.strokeStyle = "#eee";

ctx.lineWidth = 1;

ctx.beginPath();

for(let x = 0; x < canvas.width; x += gridSize) {

ctx.moveTo(x + panZoom.x % gridSize, 0);

ctx.lineTo(x + panZoom.x % gridSize, canvas.height);

}

for(let y = 0; y < canvas.height; y += gridSize) {

ctx.moveTo(0, y + panZoom.y % gridSize);

ctx.lineTo(canvas.width, y + panZoom.y % gridSize);

}

ctx.stroke();

}

Ez csak egy elképzelt implementáció – a valós kódban kicsit bonyolultabb, hogy mindig pont a megfelelő felbontású rácsot rajzolja, de az ötlet az adaptivitás. Nem kritikus a működés szempontjából, inkább fejlesztői segédfunkció.

**Pan & Zoom megvalósítása:** A felhasználói élmény részeként lehet zoomolni és mozgatni a vásznat. Ezt a kód egy panZoom objektum és néhány eseménykezelő segítségével oldja meg:

js

Másolás

const panZoom = { x: 0, y: 0, scale: 1 };

let mouse = { x:0, y:0, downX:0, downY:0, button:false, wheel:0, lastX:0, lastY:0 };

* Az mouseEvents függvény kezeli a mousedown, mousemove, mouseup, wheel eseményeket a dokumentumon.
* Lenézve a kódot: mousedown-nál beállítja a mouse.button = true és rögzíti a downX, downY értékeket (pan kezdőpont). Mousemove-nél, ha mouse.button true (tehát húzunk), akkor:

js

Másolás

panZoom.x += mouse.x - mouse.lastX;

panZoom.y += mouse.y - mouse.lastY;

mouse.lastX = mouse.x;

mouse.lastY = mouse.y;

Ez eltolja a panZoom.x,y-t a mozgás vektorával. Így megvalósul a „fogd és vidd” mozgatás.

* Wheel eseménynél:

js

Másolás

mouse.wheel += -e.deltaY;

Majd egy animációs ciklusban figyeljük a mouse.wheel értékét:

js

Másolás

if(Math.abs(mouse.wheel) >= 1) {

const scaleFactor = (mouse.wheel > 0) ? 1.1 : 0.9;

panZoom.scaleAt(mouse.x, mouse.y, scaleFactor);

mouse.wheel \*= 0.8; // csillapítás

}

A panZoom.scaleAt(cx, cy, factor) gondoskodik arról, hogy a nagyítás a megadott pont körül történjen (ehhez matematikailag a pan offset-et is módosítani kell). A kód a mi esetünkben:

js

Másolás

scaleAt(cx, cy, factor) {

const prevScale = this.scale;

this.scale \*= factor;

this.scale = Math.max(0.5, Math.min(this.scale, 3)); // limit 0.5-3

const actualFactor = this.scale / prevScale;

// az x,y offset korrigálása:

this.x = cx - (cx - this.x) \* actualFactor;

this.y = cy - (cy - this.y) \* actualFactor;

}

Így a zoom hatására a tartalom a kurzor pozícióhoz igazodik.

* A canvas rajzolásnál a panZoom-ot figyelembe kell venni: Minden rajzolás előtt meghívjuk ctx.setTransform(panZoom.scale, 0, 0, panZoom.scale, panZoom.x, panZoom.y). Ez beállítja, hogy a következő rajzolási műveletek egy transzformált koordináta-rendszerben történjenek. Mi a kódban egy panZoom.apply(ctx) segédfüggvényt használtunk erre. Ennek köszönhetően a station.x és station.y értékeket nem kell kézzel skálázgatni vagy offsetelni – a canvas API végzi. Fontos, hogy minden frame elején egyszer állítsuk be, és utána rajzoljunk le mindent.
* Refocus gomb eseménye frontend oldalon:

js

Másolás

document.getElementById('refocusBtn').addEventListener('click', () => {

panZoom.scale = 1.0;

panZoom.x = 0;

panZoom.y = 0;

update(); // redraw with new transform

});

Ez alaphelyzetbe teszi a panZoom objektumot, majd újrarajzoljuk a canvason az aktuális tartalmat immár 1x-es nagyítással és nulla offsettel.

A vászon kezelése ilyen módon biztosítja, hogy nagy mennyiségű objektum esetén is gördülékeny legyen a mozgatás, hiszen a GPU-t használjuk a transzformációhoz a canvas 2D API-n keresztül. A RailPlot jelen implementációja kisméretű hálózatokra készült, de a megoldás skálázható.

**Egyéni egérkurzor implementáció**

A projekt egyik érdekessége volt az „egyéni kurzor” vagy cursor overlay. Ezt kétféleképpen értelmezhetjük:

1. **Egérkurzor ikon cseréje:** Lehetőség van CSS-sel egyedi kurzorikon beállítására (cursor: url('...'), default;). Ezzel pl. crosshair vagy metró ikonra cserélhettük volna a nyilat. Ezt nem használtuk végül.
2. **Koordináta megjelenítése a kurzornál:** Helyette egy olyan funkciót valósítottunk meg, ami az egér pozíciójánál egy kis lebegő dobozban mutatja az aktuális koordinátát. Ennek megvalósítása:
   * HTML-ben van egy rejtett <div id="cursorCoords"> elem, ami pozicionálható (CSS: position: absolute; pointer-events: none; z-index: 9999; stb.). Ez a pointer-events: none azért kell, hogy az egér eseményeket ne zavarja (átengedje a canvasnak).
   * Az egér mozgatás eseményében (mousemove) frissítjük a tartalmát és helyzetét:

js

Másolás

document.addEventListener('mousemove', e => {

const world = panZoom.toWorld(e.clientX - canvasRect.left, e.clientY - canvasRect.top);

cursorCoords.style.left = `${e.pageX + 10}px`;

cursorCoords.style.top = `${e.pageY + 20}px`;

cursorCoords.textContent = `(${world.x.toFixed(0)}, ${world.y.toFixed(0)})`;

cursorCoords.style.display = 'block';

});

Itt a toWorld egy függvény, ami az aktuális képernyő pixel koordinátát visszaszámítja a belső „világ” koordinátára (figyelembe véve a panZoom transzformációt). Így a felhasználó valós értékeket lát, nem a transzformált pixeleket.

* + Ha az egér kilép a canvas területéről, elrejthetjük a koordináta kijelzést (pl. cursorCoords.style.display = 'none').

Ennek eredménye, hogy bárhová viszi a kurzort a térképen, látja a koordinátákat, ami a precíz tervezést segíti.

A "custom cursor" említés utalhat arra is, hogy a plusz gombokra vitt kurzor másmilyen (pl. kéz ikon), a tiltott műveleteknél szürkeárnyalatos. A CSS-ben látunk is .disabled { cursor: not-allowed; } bejegyzést, ami pl. a mentés gombokra lehet hatással, ha épp nem érvényes az űrlap.

Összességében a RailPlot nem cseréli le drasztikusan az egérkurzort, de ad plusz vizuális segítséget és állapotjelzést. Ha a fejlesztő folytatni szeretné, lehetne például:

* *Kurzor módok:* Tervezhetünk olyan módot, hogy "állomás hozzáadás" módban egy célkereszt kurzort mutasson, "pan" módban egy nyíl, "vonalszerkesztés" módban valami más. Jelenleg a módok automatikusan vannak az egér műveletek alapján, de UI/UX szempontból lehetne explicit módválasztás is.
* *Állomás húzás kurzor:* Ha bevezetjük az állomások vonszolását, akkor a drag közben a kurzor lehet mozgatást jelző ikon.

A mostani implementáció a szükséges minimált adta, a moduláris felépítés miatt bővíthető speciális kurzorjelzésekkel.

**Modalok különválasztása (új és szerkesztő)**

A felhasználói dokumentációban láttuk, hogyan működnek a modális ablakok a felületen. Fejlesztői szemmel érdekes kérdés: egy vagy két modált használjunk az új és szerkesztés funkciókhoz?

* **Egy modal, több állapot:** A RailPlot ezt az utat választotta, legalábbis az állomásoknál és a vonalaknál is. Van egy #stationModal divünk a HTML-ben, benne az űrlappal. Ugyanezt használjuk új állomásnál és szerkesztésnél is, csak dinamikusan változtatjuk a tartalmát:
  + Az Add Station gombnyomás vagy canvas click (üres helyen) esetén: stationModalTitle.textContent = "Add Station"; saveStation.textContent = "Save Station"; deleteStation.style.display = "none"; és kitisztítjuk a mezőket.
  + Egy létező állomás kattintásakor: stationModalTitle.textContent = "Edit Station"; saveStation.textContent = "Save Changes"; deleteStation.style.display = "block"; és beírjuk a mezőkbe a kiválasztott állomás adatait.
  + Ugyanazt a formot használjuk, csak hidden inputban eltesszük az ID-t is szerkesztésnél.
  + A mentés gomb eseménykezelője megnézi, hogy van-e ID (azaz editing mode), és ez alapján vagy frissít, vagy újat hoz létre (a frontenden csak a tömbben). Hasonlóképp a backend hívás is attól függhet, hogy PUT vagy POST.

Ez a megközelítés csökkenti a kódugrást (nem kell két külön űrlapot karbantartani), de figyelni kell a logikai elágazásokra. A RailPlot fejlesztése közben felmerült, hogy eleinte nem kezeltünk mindent, pl. kimaradt a modális címsor cseréje vagy a delete gomb elrejtése. Emiatt a felhasználói visszajelzések alapján javítottuk ezt (Modalok különválasztása fejlesztési pont).

* **Két külön modal:** Alternatív lehetett volna két külön HTML modal definíció: #addStationModal és #editStationModal külön formokkal. Ebben az esetben egyszerűbb a logika (a HTML statikusan már tartalmazza pl. a megfelelő címet, a delete gomb csak a szerkesztő modalon van jelen, stb.). A hátrány: duplikált kód, ha a form mezők sokban egyeznek. Emiatt ezt elvetettük.

**Fejlesztői döntés:** Végül maradtunk egy modálnál, de a kódot szépen struktúráltuk, hogy egy helyen dőljön el új vs szerkesztés. A vonalaknál ugyanez: egy #addLineModal van, ami valójában mindkét célt szolgálja. Ott a kód olvasásakor látni lehet:

js

Másolás

addLineBtn.onclick = function() {

resetLineForm();

lineModalTitle.textContent = 'Create Line';

saveLine.textContent = 'Save Line';

deleteLine.style.display = 'none';

editingLineIndex = null;

populateTrainSelect(); // feltöltjük a vonat listát

lineModal.style.display = 'block';

};

És a vonal lista Edit gombja esetén:

js

Másolás

editBtn.onclick = () => {

lineModalTitle.textContent = `Edit Line: ${line.code}`;

saveLine.textContent = 'Save Changes';

deleteLine.style.display = 'inline-block';

editingLineIndex = index;

// kitöltjük a mezőket a line adataival...

lineModal.style.display = 'block';

};

A *Modalok különválasztása* fejlesztési feladat valójában arra utalt, hogy a kezdeti implementációban keveredhetett az állapot. Például eleinte nem volt editingLineIndex, hanem a line kód alapján próbáltuk eldönteni, hogy már létezik-e a vonal, ami hibákhoz vezetett. Később bevezettük ezt a rejtett mezőt (lineIndex) és az editingLineIndex változót, ezzel egyértelművé téve a mentés logikájának, hogy épp új vagy meglévő vonalat ment.

Egy másik apróság: a modális ablakok zárógombjai (x jel). Ezekhez event listener tartozik, ami egyszerűen modal.style.display = 'none'-ra állítja az adott modalt. Ügyeltünk rá, hogy bezáráskor tisztítsuk az állapotot (pl. kijelölt állomás változót nullázzuk, resetLineForm futtatása, stb.), hogy következő nyitáskor ne maradjon benne régi adat.

Összességében a modális ablakok kezelésénél a fejlesztés során az volt a tanulság, hogy bár egy HTML sablon duplázását megspóroltuk, a kódoldali komplexitás így némileg nőtt. Ezt kompenzáltuk alapos teszteléssel és a kód tisztításával. A dokumentáció megírásának idején a modális működés stabil és könnyen bővíthető, de ha valaki preferálja a külön modalt, kis átalakítással úgy is használhatja (főleg ha UI keretrendszer, pl. Vue/React integrálásra kerülne, ott valószínűleg külön komponenst kapna az add és edit is).

**Állomás koordináták szerkesztése**

Az állomások koordinátáinak módosítása felhasználói oldalról tiltott, azonban fejlesztői szempontból gondolni kellett rá, hogyan *lehetne* ezt lehetővé tenni, és miért döntöttünk a tiltás mellett:

* **Miért tiltott a koordináta módosítás?** Mert a helyzetük alapvetően vizuálisan van meghatározva. Ha a user szabadon átírhatná a számokat, könnyen előfordulhatna, hogy az állomás „elugrik” egy tetszőleges pontra, amit lehet, hogy nem is lát a jelenlegi nézetben. Ráadásul, ha tévesen ír be valamit, nehéz visszaállítani.
* **Hogyan lehetne engedélyezni?** Egyszerűen meg kéne szüntetni a readonly attribútumot a Station modal X/Y inputjain, így a user szerkesztheti. Mentéskor a kód már úgyis veszi az input mező értékét:

js

Másolás

const newStation = {

id: Date.now(),

x: parseFloat(document.getElementById('stationX').value),

y: parseFloat(document.getElementById('stationY').value),

name: name,

location: selectedLocations

};

Tehát valójában a program fel van készítve rá, hogy akár a felhasználó által módosított X/Y-t vegye át. (A UI ezt nem engedi, de ha valaki devtools-ban kiveszi a readonly-t, majd átírja a számot és ment, a frissített helyen jelenne meg az állomás.)

* **Mi szükséges a biztonságos engedélyezéshez?** Egyrészt egy vizuális visszajelzés (valami markert mutatni, hova mozdult), másrészt a vonalak integritását figyelni. Mivel a vonalak állomások nevei alapján kapcsolódnak, a koordináta változás nem befolyásolja a logikai kapcsolatot, csak vizuálisan a vonal szakasz új pozícióra megy. Ez tehát nem veszélyes. Ami veszélyesebb: esetleg két állomás koordináta ütközhet (pont ugyanoda helyez két állomást, de ez sem logikai gond, csak grafikus: fedésbe kerülnek).
* **Alternatív megoldás – Drag & Drop:** Fejlesztői szempontból vonzóbb lenne, ha az állomásokat az egérrel is arrébb lehetne húzni. Ezt úgy lehetne implementálni:
  + A canvas click eseménynél, ha állomásra kattintunk, ne csak a modalt nyissuk meg, hanem jegyezzük fel, hogy dragging mode-ba mehetünk.
  + Mousedown egy állomás közelében -> kiválasztja, majd ha mozgatjuk (mousemove drag közben), valójában módosítja a stations listában annak a station objektumnak az x,y értékeit. Majd a update() függvény révén realtime mozog a pont.
  + Mouseup-nál befejezi a drag-et, esetleg nyitva hagyhatjuk a modalt frissített értékekkel.
  + Ehhez a kódhoz kicsit át kellene alakítani a jelenlegi click-kezelést, ami egyből modalt nyit. Mondjuk lehetne jobb klikk vagy egy "Move" mód eszköztár.

Fejlesztés során úgy ítéltük meg, hogy ez nice-to-have, de nem kritikus, ezért maradt a read-only és a fentebb említett workaround (export/import) haladó felhasználóknak. A *Fejlesztési mérföldkő* listában azonban szerepelt ez a pont, mert fontolgattuk:

* Eleinte a koordináták egyáltalán nem jelentek meg az UI-on, majd hozzáadtuk őket információként.
* Később lehet, hogy feloldjuk a zárolást, ha a felhasználók igénylik. Ehhez a kód már csak kevés módosítást igényel.

**Összefoglalva fejlesztői szempontból:** Jelenleg az állomások pozícióját a felhasználó csak a térképen való újraelhelyezéssel (törlés+új hozzáadás) tudja változtatni. A kód fel van készítve a koordináták tárolására, sőt a JSON fájlban is ember olvashatóan benne vannak. Ha integrálni akarnánk egy külső térképes API-val (pl. OpenStreetMap koordinátákat bemérni), a backend-oldali módosítás ugyanígy bevinné a változást. Tehát fejlesztőként nyitva áll az út a funkció kibővítésére. Azért választottuk szét a frontenden a X/Y megjelenítését és nem szerkesztését, hogy megelőzzük a véletlen inkonzisztenciát.

**Statisztikai felület megvalósítása**

A statisztikai panel alul programozás szempontjából egy statikus HTML <div> halmaz, fix ID-kkal a span elemekre (pl. totalStations, groundStations, stb.). A frissítése teljes mértékben a frontenden, JavaScript-tel történik, jellemzően az updateStationStats() függvényben:

js

Másolás

function updateStationStats() {

const totalStations = stations.length;

const groundStations = stations.filter(s => s.location.includes('ground')).length;

const undergroundStations = stations.filter(s => s.location.includes('underground')).length;

const suspendedStations = stations.filter(s => s.location.includes('suspended')).length;

document.getElementById('totalStations').textContent = totalStations;

document.getElementById('groundStations').textContent = groundStations;

document.getElementById('undergroundStations').textContent = undergroundStations;

document.getElementById('suspendedStations').textContent = suspendedStations;

const totalLines = lines.length;

const groundLines = lines.filter(l => l.type === 'ground').length;

const undergroundLines = lines.filter(l => l.type === 'underground').length;

const suspendedLines = lines.filter(l => l.type === 'suspended').length;

document.getElementById('totalLines').textContent = totalLines;

document.getElementById('groundLines').textContent = groundLines;

document.getElementById('undergroundLines').textContent = undergroundLines;

document.getElementById('suspendedLines').textContent = suspendedLines;

const totalTrains = trains.reduce((sum, t) => sum + t.quantity, 0);

const groundTrains = trains.filter(t => t.type === 'ground').reduce((sum, t) => sum + t.quantity, 0);

const undergroundTrains = trains.filter(t => t.type === 'underground').reduce((sum, t) => sum + t.quantity, 0);

const suspendedTrains = trains.filter(t => t.type === 'suspended').reduce((sum, t) => sum + t.quantity, 0);

document.getElementById('totalTrainsStat').textContent = totalTrains;

document.getElementById('groundTrainsStat').textContent = groundTrains;

document.getElementById('undergroundTrainsStat').textContent = undergroundTrains;

document.getElementById('suspendedTrainsStat').textContent = suspendedTrains;

}

(Ez egy kibővített változat, a valós kódban a logika ugyanilyen.)

Néhány érdekesség fejlesztői szemmel:

* A statisztika számítás jelenleg minden *updateStationStats* híváskor végigiterál mindenen (filter, reduce). Ez O(n) művelet mindhárom kategóriára. Mivel tipikusan n (állomások, vonalak, vonatok) kicsi, ez nem gond. De ha tízezres nagyságrendbe menne, lehetne optimalizálni úgy, hogy csak a változásokat követjük nyomon. Pl. állomás hozzáadásnál nem kell újraszámolni mindent from scratch, elég totalStations++ és a megfelelő kategóriának++.
* Az állomások location mezője array, így a filter-nél includes('ground') használata kicsit tág. Hisz ha egy állomás mind ground, mind underground, mind suspended, akkor mindhárom filter beleszámolja, tehát a sum of (groundStations + undergroundStations + suspendedStations) akár nagyobb is lehet, mint totalStations. Ezt a UI-n nem jelezzük külön, de fejlesztőként tudjuk: egy állomást akár többször is számolunk a bontásban. Ez nem feltétlen probléma, mert a "Stations:" sorban a total a lényeges, a többi csak kategória halmozottan. A "Trains:" résznél hasonló a helyzet: ha egy vonatot felszíni és földalatti típusúnak is tekintenénk (nálunk nem, egy vonat egy type).
* Az updateStationStats hívása be van fűzve sok helyre: mindenhol, ahol változhatnak az adatok:
  + Állomás létrehoz/töröl (frontenden a saveStation és deleteStation eventek végén).
  + Vonal létrehoz/töröl (saveLine és deleteLine event után).
  + Import vége.
  + Clear Grid vége.
  + etc. Így garantáljuk a naprakészséget. Egyszer egy apró bug volt: eleinte nem hívtuk meg, amikor csak simán belépett a user (tehát üres oldalt mutatott 0-kal?), ezért a DOMContentLoaded-kor is meghívjuk, hogy nulláról is jól kiírja a 0 értékeket.
* Ha a jövőben bővül a statisztika (pl. legnagyobb vonal hossza, átlagos állomásszám vonalanként, stb.), akkor azt is itt számolnánk ki. Az architektúra nyitott rá.

Fejlesztői javaslat: ha backend is akar statisztikát, csinálhat ezekről aggregált lekérdezéseket (SQL COUNT, etc.), de mivel a frontenden úgyis megvannak az adatok valós időben, felesleges terhelni a szervert. Tehát maradhat a kliens oldalon.

**Fájlkezelés: JSON exportálás és importálás**

A JSON import/export implementációja a RailPlot egyik legösszetettebb része, ahogy a felhasználói részben is részleteztük. Fejlesztői szemmel kiemeljük, hogyan is valósítottuk meg:

* **Exportálás:**
  + Nem vettünk igénybe a backend-et, az export teljes mértékben a frontenden történik, így offline is működik. A JavaScript generál egy blob-ot és letöltést kezdeményez:

js

Másolás

document.getElementById('exportBtn').addEventListener('click', () => {

const data = { stations, lines, trains };

const blob = new Blob([JSON.stringify(data, null, 2)], { type: 'application/json' });

const url = URL.createObjectURL(blob);

const a = document.createElement('a');

a.href = url;

a.download = 'station\_data.json';

a.click();

URL.revokeObjectURL(url);

});

A fenti kód snippet az export funkció lényegét mutatja. Lényeges paraméter:

* + - JSON.stringify(data, null, 2): a null,2 paraméterrel szépen formázott (2 space indent) JSON-t készítünk, hogy a fájl olvasható legyen.
    - Blob és URL.createObjectURL: ezek a böngésző mechanizmusai a fájl letöltéshez. Ezzel kiküszöböltük, hogy külső komponenst kelljen használni.
  + Mivel a stations, lines, trains tömbök JavaScript objektumokat tartalmaznak, a JSON.stringify gyönyörűen megoldotta a konverziót. Egy apróság: a Station objektumok location mezője array, a JSON-ben is array lesz (ezt akartuk), a Line objektum stations mezője array (nevek listája), a JSON-ben is array marad – tökéletes a visszaimportálhatósághoz.
  + Az export gomb nem érhető el, ha nem vagyunk bejelentkezve (mert akkor nincs is adathalmaz), de ezt nem is kellett külön védeni, hisz login nélkül a felület sem látszik.
* **Importálás:**
  + Egy rejtett file input (#importInput) van az oldalon accept=".json"-nel. Az Import gomb click eseményében annyi történik, hogy importInput.click() – ez megnyitja a fájlválasztót.
  + Az input change eseményén:

js

Másolás

importInput.addEventListener('change', (event) => {

const file = event.target.files[0];

if (!file) return;

const reader = new FileReader();

reader.onload = function(e) {

try {

const json = JSON.parse(e.target.result);

// ... validációk majd betöltés ...

} catch(err) {

alert("Error reading file: " + err.message);

}

importInput.value = null; // reset, hogy ugyanazt a fájlt is újra lehessen választani később

};

reader.readAsText(file);

});

A fenti részlet mutatja a folyamatot. A FileReader aszinkron, de onload-ban dolgozzuk fel.

* + **Validációk:** A try/catch már jelzi, hogy a JSON.parse is lehet hibás, azt elkapjuk. Utána egy sor ellenőrzés fut, amiket a felhasználói részben már felsoroltunk. Ezek a frontenden vannak implementálva if-ekkel, alert-ekkel. Bár ez sok kód, a fejlesztés során tudatosan választottuk a kliens oldali validációt is, mert:
    - Ha offline használná valaki (mondjuk file:// protokollal megnyitja, extrém eset), a backend nincs is bevonva.
    - Egyszerűbb, mint serverre feltölteni a fájlt (sőt, a privacy is jobb, nem megy át hálózaton).
    - Azonnali visszajelzést ad a usernek.
    - Ráadásul a frontenden amúgy is JS objektumokká alakítjuk a tartalmat, amikkel a programnak dolgoznia kell. Így a validálás ingyen van, hisz a struktúrával már amúgy is foglalkozunk.

A validáció főbb pontjai:

* + - Struktúra ellenőrzés: typeof json === 'object' && Array.isArray(json.stations) stb.
    - Tartalmi ellenőrzések: .every() és .filter() kombinációkkal. Pl:

js

Másolás

const stationsValid = json.stations.every(st => st && typeof st.id !== 'undefined' &&

typeof st.x === 'number' && typeof st.y === 'number' &&

typeof st.name === 'string' && st.name.trim() !== '' &&

Array.isArray(st.location) && st.location.every(loc => ['ground','underground','suspended'].includes(loc))

);

if(!stationsValid) { alert("Invalid station data"); return; }

Hasonló blokkok a trains-re és lines-ra.

* + - Kereszthivatkozások: itt for-of ciklusban megyünk végig a vonalakon, mert bonyibb logika kellett (állomás létezés, duplikált állomás egymás mellett, típus egyezés, vonat egyezés).
    - Bármely hibánál return a try blockból, így a catch nem fut le, hanem mi magunk jeleztünk és megszakítunk.
  + **Adatok betöltése:** Ha minden ellenőrzés zöld utat kap, akkor:

js

Másolás

stations = json.stations;

lines = json.lines;

trains = json.trains;

update();

updateLinesUI();

updateStationStats();

alert("Data imported successfully!");

Itt fontos: primitív módon felülírjuk a belső tömbjeinket. Mivel referenciákat cserélünk, a canvas frissítéshez mindenképp meg kell hívni az update() rajzolást, mert a requestAnimationFrame ha fut is, lehet, hogy a referenciaváltást nem érzékeli (bár mivel global scope változókat cseréltünk, onnan is elérhető, de biztos ami biztos, hívjuk az updateStationStats-ot is). A lines listát is frissíteni kell a jobboldali panelen (updateLinesUI).

Alternatív design lett volna, hogy nem cseréljük le a tömböket, hanem egyesével push/pop-olunk a különbségek szerint. De egyszerűbb és tiszta: mindent csere. Memória szempontból nem gond, max pár száz objektum.

* + **Robusztusság:** Minden lehetséges fals adatot igyekeztünk lekezelni. Például, ha valaki kézzel ír egy JSON-t és kihagy valamit, inkább szóljon a program, minthogy félig töltse be és furcsa hibák legyenek futás közben.
  + **Backend integráció:** Megjegyzendő, hogy lehetne a backendre bízni import/exportot:
    - Export: egy /api/export endpoint, ami JSON-t ad. (De itt értelmetlen kerülőt tennénk.)
    - Import: feltölteni a fájlt a szerverre, ott feldolgozni. Itt nehézség, hogy a webszerver crossdomain, fileupload, stb. plusz complexity. Ehhez képest a JS FileReader localban oldja meg.

Ezzel a modul testreszabható: ha valaki bővíti a Station vagy Line objektumokat új mezőkkel, csak az import/export kódot kell frissíteni, hogy azt is kezelje. Például, ha bevezetnénk Station-hez egy "description" mezőt, az export automatikusan betenné (mert az object stringify mindent visz), de importnál a validációt bővíteni kellene (megengedni a plusz mezőt vagy figyelmen kívül hagyni).

A **JSON schema** definíciót akár formálisan is megadhatnánk (pl. JSON Schema), de a kézzel írt ellenőrzések elégségesnek bizonyultak és kevésbé bonyolult integrálni.

**Fejlesztési mérföldkövek és események**

A RailPlot fejlesztése során több fázison ment keresztül, a funkcionalitás fokozatos kibővítésével és finomításával. Az alábbiakban kronologikus sorrendben bemutatjuk a főbb mérföldköveket:

* **Kezdeti prototípus – csak frontend (állomások és egyszerű vonalak):** Az első működő verzió egy önálló HTML/JS oldal volt, Laravel backend nélkül. Ebben már megjelent az állomások vászonra helyezése kattintással, a zoom/pan, és egy kezdetleges vonal-összekötés (pl. canvas click után prompt-tal kérte be a nevet). Ekkor még nem volt felhasználókezelés, minden kliens oldalon történt (a mentés is csak böngésző memória szinten).
* **Backend integráció és felhasználókezelés hozzáadása:** Felismerve, hogy szükség van adatok tartós tárolására és több felhasználó támogatására, integráltuk a Laravel backendet. Ez volt egy fontos mérföldkő:
  + Bevezettük a **regisztráció/bejelentkezés** mechanizmust. Ezzel a RailPlot alkalmazás többfelhasználóssá vált.
  + Az állomásokat és vonalakat immár adatbázisba mentettük. Létrejöttek a Laravel modellek, migrációk.
  + Kialakítottuk az **API végpontokat** a frontendi kommunikációhoz. Ezen a ponton a frontenden cseréltük a korábbi lokális adatfeltöltést Axios hívásokra. Például a fetchStations() függvény már ténylegesen meghívta a backend API-t, nem csak a memóriában lévő állomásokat használta. (Az API integrációval a fejlesztői munkában nyertünk: most már perzisztens módon, oldalfrissítés után is megmaradtak az adatok.)
* **Állomás- és vonalkezelés UI bővítése:** A következő fejlesztési hullám a felhasználói felület funkciógazdagabbá tétele volt:
  + Elkészültek a **modális ablakos űrlapok** az állomások és vonalak létrehozásához/szerkesztéséhez. Ez nagy lépés volt az ergonómia terén, mivel ekkortól nem a prompt és konzol hack-ek vitték a bevitel, hanem egy rendezett űrlap felület.
  + **Vonal szerkesztő UI:** Külön kihívás volt a vonalak állomáslistájának interaktív szerkesztése. Ezt a plusz gombos megoldást implementáltuk ebben a fázisban. Először csak hozzáadni lehetett állomást a vonalhoz, a sorrend módosítására még nem volt mód. Később frissítettük a kódot, és bevezettük a kis nyilakat és törlés gombot minden állomás mellé, hogy mozgatható és kivehető legyen. Ezzel a vonalszerkesztő teljes körűvé vált.
  + **Állomás típusa és vonal típusa:** Ezeket is ekkor adtuk hozzá. Korábban minden állomás/vonal egyforma volt, később jött az igény, hogy megkülönböztessük pl. metró vs. felszíni. Így az állomás űrlapra felkerültek a checkboxok, a vonal űrlapra a rádiógombok. Ehhez hozzátartozott a belső logika frissítése is (pl. a vonal létrehozásnál szűrjük a listában az állomásokat típus alapján).
  + **Egyedi kurzor koordináta kijelzés:** Fejlesztői eszközként indult, de megtartottuk felhasználói funkciónak is, beépítettük a projektbe.
  + **Statisztikai panel:** Ekkoriban született meg alul a stat panel, kezdetben csak állomás/vonal counttal. Később kibővítettük a vonatok adataival is.
  + **Mobiloptimalizálás első lépései:** A Bootstrap használata eleve adott némi reszponzivitást, de utólag CSS media query-kel javítottuk a modális ablakok méretezését, input mezők elrendezését, gombok méretét. Ez kismértékű fejlesztés volt, de fontos mérföldkő (a visszajelzések alapján mobilon is próbálták többen).
* **Import/Export funkció hozzáadása:** Miután a felhasználók elkezdték komolyabban használni, felmerült az igény az adatok kimentésére. Így implementáltuk a JSON exportot és importot. Ez egy jelentősebb fejlesztési esemény:
  + Az export megírása viszonylag gyorsan ment (a böngésző File API-k miatt).
  + Az importnál sok tesztelés kellett, hogy minden lehetséges hibalefedést jól csináljunk. Ezt a funkciót alaposan dokumentáltuk, és belsőleg is többször ellenőriztük különböző esettanulmány fájlokkal.
* **Adminisztrációs panel bevezetése:** A fejlesztés későbbi szakaszában készült el egy külön admin felület:
  + Ez egy Laravel blade view-kból álló modul, ahol a felhasználók listája, a vonatok listája szerkeszthető. Nem része a nyilvános frontendi UI-nak, de a projekt integritása szempontjából mérföldkő, mert ezzel lett teljes a felhasználó/vonat menedzsment.
  + Az admin panelen például felvihet a rendszergazda új vonattípusokat (pl. "HÉV", "Trolibusz" stb. meghatározott mennyiséggel), amik aztán megjelennek a frontendi legördülőben. E nélkül a frontenden fix listából lehetne csak választani, ami kötött. Így viszont dinamikus.
  + Az admin panel hozta magával, hogy az admin user kilistázhatja bármely felhasználó hálózatát. Ez debug célokra is hasznos (ha egy user hibát jelez, admin látja az adatait).
* **Hibajavítások és finomítások:** A fejlesztés iteratív jellege miatt több apróbb hibát korrigáltunk:
  + **Modal állapot hibák:** Eleinte előfordult, hogy egy vonal szerkesztése után a következő hozzáadásnál a mezőkben bennragadt az előző adata. Ezt a resetLineForm() bevezetésével oldottuk meg, ami nyitáskor mindig alaphelyzetbe állítja az űrlapot.
  + **Duplikált kód figyelmeztetés:** Korábban a vonalkód egyediség ellenőrzése hiányzott update-nél, ezt pótoltuk.
  + **Clear Grid megerősítés:** Eredetileg a Clear Grid azonnal törölt mindent, ami veszélyes. Később beépítettünk egy confirm() kérdést, hogy ne lehessen véletlenül adatot veszteni.
  + **Import robustizálás:** Első verzióban az import kevesebb ellenőrzést csinált (pl. nem nézte a duplikált station neveket vagy a vonalhosszt). Felhasználói visszajelzés nyomán (amikor valaki rossz JSON-t próbált betölteni és csak félig sikerült) fokoztuk a validációt. Azóta nem érkezett hibajelzés e modul kapcsán.
  + **Állomás törlés logika:** Mint említettük, a backend oldalon is implementáltuk, hogy ne törölhető állomás vonal esetén, illetve a frontenden is tettünk plusz alertet. Így kiküszöböltük, hogy az állomáslista és vonallisták szinkronja felboruljon.
  + **Teljesítmény finomhangolások:** Bár a jelenlegi volumen nem indokolta, kipróbáltunk néhány optimalizációt (pl. canvason nem requestAnimationFrame-ezzük feleslegesen, hanem csak akkor rajzolunk, ha kell). Végül is maradt némi idle loop, de ennek erőforrásigénye minimális, és cserébe segít pl. a vonal kiemelés animációjában (ahol hoverre vastagabb vonalat rajzol).

Ezek a mérföldkövek mind hozzájárultak, hogy a RailPlot eljusson a mostani stabil, funkciókban gazdag verziójához. Minden nagyobb változás előtt és után is történt tesztelés (lásd következő pont), ami biztosította, hogy az új elemek nem törnek el régi funkciókat.

**Tesztelés**

A RailPlot esetében a tesztelés több szinten zajlott, főként manuális és felhasználói visszajelzések alapján:

* **Funkcionális tesztek (manuális):** A fejlesztői csapat minden új funkció beépítése után végigpróbálta a fő felhasználói folyamatokat:
  + Regisztráció -> Bejelentkezés -> Állomás létrehozás -> Vonal létrehozás -> Módosítások -> Mentés -> Kijelentkezés -> Újra bejelentkezés (adatok megmaradtak-e).
  + Szélsőséges esetek kipróbálása: üres mezők megadása (hibakezelés jön-e), duplikált nevek/kódok megadása, nagyon sok állomás/vonal felvitele (teljesítmény figyelése), zoom és pan extrém használata, mobilnézeten űrlap kitöltés virtuális billentyűzettel, stb.
  + Import/Export teszt: Egymás után többszöri export/import, valamint kézzel módosított JSON importálása. Készítettünk néhány mesterséges JSON-t (pl. hiányzó mezőkkel, duplikált nevekkel) és megnéztük, hogy megfelelően elutasítja-e.

E manuális tesztek során felmerült hibákat rögtön javítottuk (ezekről szóltunk a hibajavítások résznél). Idővel összeállt egy ellenőrzőlista, amit minden release előtt végigmentünk.

* **Automatizált tesztek (terv):** Laravel oldalon írtunk néhány alap tesztesetet a kritikus backend logikára:
  + Station API validáció (pl. nem enged duplikált nevet létrehozni).
  + Line API validáció (két állomásnál kevesebbel 422 hibát ad).
  + Auth middleware működés (vendég ne férjen hozzá /api/stations végponthoz).

Ezek futtatása biztosította, hogy a backend stabil marad. A frontendre E2E (end-to-end) tesztet is lehetne készíteni (pl. Selenium vagy Puppeteer segítségével), de ezt jelen projektben nem valósítottuk meg, helyette a manuális tesztelés dominált.

* **Felhasználói visszajelzések alapján végzett finomítások:** A RailPlot kipróbálásra került egy kis felhasználói csoport által (kollégák, ismerősök). Az ő visszajelzéseik rendkívül hasznosak voltak:
  + Jelezték például, hogy mobilon a modális ablak kilóg kicsit – ezt javítottuk CSS-ben.
  + Kérték, hogy legyen megerősítés a veszélyes műveleteknél – bevezettük a confirmokat.
  + Voltak, akik összezavarodtak, hogy egy űrlapban van az új és szerkesztés – ennek megoldására egyértelmű feliratokat tettünk (Add vs Edit, Save vs Save Changes), és ezt a doksiban is hangsúlyoztuk.
  + Egyesek hiányolták a mentés (export) lehetőségét – ennek eredménye lett a JSON export modul.
  + A statisztikai panel ötlete is felhasználóktól jött, hogy jó lenne látni hány elem van összesen.

Minden ilyen javaslatot megfontoltunk, és a többségét implementáltuk is, amennyiben illeszkedett a projekt scope-jába. A RailPlot tehát mondhatni *felhasználóközpontú iterációkkal* fejlődött.

* **Biztonsági tesztek:** Mivel a projekt főként belső használatra készült, komoly biztonsági audit nem volt. Azért néhány alapvető dolgot ellenőriztünk:
  + XSS támadás ellen: az állomásneveket és egyéb felhasználói inputot megjelenítés előtt leescape-eli a böngésző (mivel textContent-et használunk, nem innerHTML-t), így oda direkt scriptet nem lehet becsempészni.
  + SQL injection: Laravel Eloquent és Query Builder haszná latával eleve védve vagyunk, és paraméter binding megy a raw query-knél is.
  + File import: csak JSON-t engedünk, és még abban sincs futtatható kód.
  + Auth: kipróbáltuk, hogy be nem jelentkezve a /canvas oldal nem elérhető, és API végpontok sem (403-at ad).

A jövőben, ha nyilvánosabbá válnak a RailPlot telepítések, érdemes lehet egy alaposabb pentest (penetrációs teszt), de jelen állás szerint a tipikus webes sebezhetőségektől védve van alap szinten.

Összegezve: a tesztelés iteratív folyamata biztosította, hogy a RailPlot megbízhatóan működjön. A felhasználói élmény és a stabilitás egyaránt javult a rendszeres tesztek és visszajelzések feldolgozása által.

**Lehetséges jövőbeli fejlesztések**

Bár a RailPlot jelenlegi verziója már egy használható és stabil eszköz, számos ötlet merült fel a továbbfejlesztésére. Néhány lehetséges jövőbeli bővítés vagy javítás:

* **Állomások mozgatása a felületen:** Ahogy fent is említettük, hasznos lenne, ha a felhasználó közvetlenül a térképen áthelyezhetne egy állomást (pl. megragadja és odébb húzza). Ez megkímélné attól, hogy törölnie és újra felvinnie kelljen az állomást, ha netán rossz helyre került. E funkció implementálása mérsékelten összetett (drag & drop a canvas-on, vonalak valós idejű újrarajzolása), de megvalósítható.
* **Több térkép/projekt kezelése felhasználónként:** Jelenleg egy felhasználó egy hálózatot épít. Kiterjeszthető a rendszer úgy, hogy egy user több különálló projektet tárolhasson (pl. külön városok hálózatai). Ehhez az adatmodellt kiegészíthetnénk egy Project entitással, amelyhez állomások és vonalak tartoznak. A felületen ez egy projektek közti váltást igényelne (pl. legördülő, vagy nyitóképernyő projektválasztással).
* **Valós térképi integráció:** Egy haladó fejlesztés lenne integrálni mondjuk a Leaflet vagy Google Maps API-t, hogy a RailPlot ne egy absztrakt koordináta rendszerben dolgozzon, hanem valós térképen (geokoordinátákkal). Az állomásoknak lehetne valós földrajzi koordinátája, és egy város térképén helyezkednének el. Ezzel a felhasználási lehetőségek kibővülnének (pl. városi közlekedési hálózatok tervezése a valóságban).
* **Útvonaltervezés/szimuláció:** Jövőbeli ötletként felmerült egy egyszerű útvonaltervező modul: megad két állomást, és a program megkeresi, mely vonalak és átszállások érintésével lehet eljutni. Ehhez gráf algoritmusokat (pl. BFS a hálózatban) lehetne használni. Esetleg vizuálisan ki is emelhetné az útvonalat a térképen.
* **Animáció – vonatok mozgása:** Ha már vonatokat és mennyiségeket nyilvántartunk, látványos lehetne, ha a vonalakon pontok (járműszimbólumok) mozognának, jelezve a forgalmat. Ezzel szimuláció jelleggel mutatná a rendszer, hogyan járnak a szerelvények. Persze ez komolyabb fejlesztés (időzítések, mozgás interpoláció, ütközésmentesítés, stb.), de egy demó szintjén akár megvalósítható (pl. állandó sebességgel körbemenő pöttyök a vonalak mentén).
* **Ütközéskezelés és intelligens elrendezés:** Ha egy helyre több állomás kerül (pl. különböző projektek importja miatt duplikáció), a jelen rendszer nem kezeli külön. Jövőben lehetne észlelni a nagyon közeli pontokat és vizuálisan eltolni a feliratokat, vagy figyelmeztetni a felhasználót.
* **Közösségi funkciók:** Lehetőséget teremteni, hogy a felhasználók megosszák a hálózataikat egymással. Ez jelentheti publikálást (átállítani a projektet publikusra, és egy linkkel bárki megnézheti), vagy akár valós idejű kollaborációt (többen szerkesztik ugyanazt a projektet egyszerre websockets/WebRTC segítségével). Ez utóbbi nagyon ambiciózus, de manapság elérhető technológia (pl. Yjs, Operational Transformations).
* **Felhasználói élmény javítások:** Ilyenek például: visszavonás/újra (undo/redo) stack bevezetése, részletesebb súgó integrálása az app-ba, jobb hibajelzések (nem alert, hanem szebben megjelenő üzenetek a felületen), témázhatóság (sötét/világos mód), stb.
* **További statisztikák és jelentések:** Készíthetnénk egy külön oldalt, ahol grafikonokon ábrázolja a rendszer az adatokat (pl. időben hogyan nőtt a hálózat – ha verziók vannak; vagy milyen hosszúak a vonalak km-ben – ha geokoordináták vannak; állomások kapcsolati fokszáma stb.). Ezek inkább érdekességek, de egy fejlett tervező rendszernél fontosak lehetnek.

Természetesen minden további fejlesztés erőforrás és igény függvénye. A RailPlot nyílt architektúrája azonban lehetővé teszi a fenti bővítések integrálását. A kód moduláris felépítése (külön backend API réteg, külön frontend logika) segít abban, hogy egy-egy modult cseréljünk vagy fejlesszünk anélkül, hogy az egész rendszert újra kellene írni.

A jelen dokumentációval és a kódbázis ismeretével felvértezve bízunk benne, hogy a jövő fejlesztői sikeresen tovább tudják vinni a RailPlot-ot, új funkciókkal gazdagítva azt. Az ötletek sora végtelen, így a projektnek van tere a növekedésre mind felhasználói bázis, mind funkcionalitás tekintetében. Jó munkát és kreatív fejlesztést kívánunk hozzá!