Tartalomjegyzék

[2](#_Toc192584605)

[3 Bevezetés 2](#_Toc192584606)

[3.1 A projekt fő célja 2](#_Toc192584607)

[3.2 A projekt megalakulása 2](#_Toc192584608)

[3.3 Célfelhasználók 2](#_Toc192584609)

[4 Áttekintés 3](#_Toc192584610)

[4.1 Mobiltelefon-specifikációk és iparági szabványok 3](#_Toc192584611)

[4.2 Szűrőrendszerek és algoritmusok 3](#_Toc192584612)

[4.3 Felhasználói élmény (UX) és design szempontok 3](#_Toc192584613)

[4.4 Technológiai háttér 3](#_Toc192584614)



# Bevezetés

## A projekt fő célja

A ChoosePhone projekt célja, hogy jelentős segítséget nyújtson a felhasználóknak a mobiltelefonok széles választékában való felfedezésében. A modern technológia világában, ahol naponta újabb és újabb okostelefonok jelennek meg, elengedhetetlen, hogy a vásárlók megalapozott döntéseket tudjanak hozni. A ChoosePhone egy felhasználóbarát platform, amely a részletes specifikációkon alapuló összehasonlítások révén könnyen navigálható megoldást kínál.

## A projekt megalakulása

A projekt indításának hátterében az állt, hogy a mobiltelefonok vásárlása gyakran bonyolult és időigényes feladat. A felhasználók számos tényezőt, például a processzor teljesítményét, a memória kapacitását, a kamerák minőségét és az akkumulátor élettartamát képesek figyelembe venni, de a sok információ között nehéz a legjobb döntést hozni. A ChoosePhone célja, hogy ezen a problémán segítve, egy átfogó és intuitív platformot biztosítson, ahol a felhasználók könnyedén összehasonlíthatják a különböző modellek jellemzőit.

## Célfelhasználók

A ChoosePhone projekt széles körű felhasználói bázist céloz meg, amely különböző igényekkel rendelkezik:

1. **Átlagos vásárlók**: Akik új telefont keresnek, de nem feltétlenül értenek mélyrehatóan a technológiai specifikációkhoz. Számukra egyszerű, intuitív szűrési lehetőségeket biztosítunk.
2. **Tech-rajongók és szakértők**: Akik részletes műszaki adatokat szeretnének látni, és összehasonlítani a telefonok teljesítményét. Ők nagyra értékelik a részletes specifikációkat és benchmark-eredményeket.
3. **Mobiltelefon-kereskedők**: Akik gyors és pontos összehasonlításra támaszkodnak az ügyfelek tájékoztatásához.
4. **Gamerek és multimédia-felhasználók**: Akik számára a kijelző minősége, a processzor teljesítménye és az akkumulátor kapacitása kritikus tényező.

# Áttekintés

## Mobiltelefon-specifikációk és iparági szabványok

Az okostelefonok specifikációi kulcsfontosságú szerepet játszanak a felhasználói élményben és a vásárlási döntésekben. A legfontosabb paraméterek közé tartozik a processzor teljesítménye, a RAM mérete, a kamera minősége, a kijelző technológiája és az akkumulátor élettartama. Az iparágban elfogadott szabványok, mint például az 5G hálózati támogatás, az OLED kijelzők fejlődése vagy a többkamerás rendszerek elterjedése, meghatározzák a készülékek funkcionalitását és élettartamát.

## Szűrőrendszerek és algoritmusok

A modern adatkezelési és szűrési technológiák lehetővé teszik, hogy a felhasználók testreszabott kereséseket végezzenek. A relációs adatbázisok (pl. MySQL, SQLite) és a NoSQL megoldások (pl. MongoDB) egyaránt alkalmasak nagy mennyiségű telefonos adat kezelésére. Az olyan algoritmusok, mint a többkritériumos döntéstámogatás (Multi-Criteria Decision Analysis – MCDA) vagy a mesterséges intelligencia alapú ajánlórendszerek, tovább növelhetik a felhasználói élményt.

## Felhasználói élmény (UX) és design szempontok

A felhasználói élmény kulcsfontosságú egy sikeres mobiltelefon-összehasonlító platform esetében. Az átlátható felhasználói felület, az intuitív szűrési lehetőségek és a vizuálisan jól strukturált összehasonlítások hozzájárulnak a vásárlói döntéshozatal egyszerűsítéséhez. A modern UX-design trendek, mint a minimalizmus és az interaktív grafikonok, tovább növelhetik a ChoosePhone projekt vonzerejét.

## Technológiai háttér

A ChoosePhone egy webalapú alkalmazásként működik, amely modern frontend és backend technológiákat alkalmaz. A frontend fejlesztés során népszerű JavaScript-keretrendszerek, például React vagy az Angular használhatóak a dinamikus felhasználói élmény biztosítására, a projekt esetében az Angular került kiválasztásra. A backend esetében Node.js vagy PHP kínálhat megfelelő teljesítményt és skálázhatóságot, vagy egy ezen nyelven alapú keret Az API-alapú adatkezelés biztosítja az adatok naprakészségét és gyors elérhetőségét.

# Tervezés és követelmények

A ChoosePhone rendszer tervezése során figyelembe kell venni a funkcionalitást, a teljesítményt és a felhasználói élményt biztosító követelményeket. Az alábbiakban bemutatjuk a legfontosabb követelményeket és az architekturális tervezést.

## Funkcionális követelmények

* **Felhasználói regisztráció és bejelentkezés**: A felhasználók számára lehetőséget kell biztosítani a profil létrehozására és kezelésére.
* **Telefonok összehasonlítása**: A rendszernek lehetővé kell tennie, hogy a felhasználók két vagy több készüléket részletesen összehasonlítsanak.
* **Szűrési és keresési lehetőségek**: Több szempont alapján történő keresés és szűrés szükséges (ár, kijelzőméret, processzor, kamera, akkumulátor stb.).
* **Adatbázis frissítése**: Az újonnan megjelent készülékek és friss specifikációk folyamatosan frissüljenek az adatbázisban.
* **Felhasználói értékelések és vélemények**: A látogatók számára biztosítani kell a lehetőséget, hogy értékeléseket írjanak és megoszthassák tapasztalataikat.

## Nem funkcionális követelmények

* **Teljesítmény**: Az oldalnak gyorsan kell betöltenie, és a szűréseknek valós időben kell működniük.
* **Biztonság**: A felhasználói adatok védelme és a biztonságos bejelentkezés elengedhetetlen.
* **Skálázhatóság**: A rendszernek bővíthetőnek kell lennie, hogy a növekvő felhasználói bázist kiszolgálja.
* **Mobilbarát kialakítás**: Az alkalmazásnak reszponzív designnal kell rendelkeznie, hogy mobiltelefonon és tableten is jól működjön.

## Rendszertervezés

A ChoosePhone architektúrája három fő komponensből áll:

1. **Frontend**: A projektünkhöz az Angular keretrendszert választottuk, amely komponens-alapú megközelítésével lehetővé teszi a dinamikus, jól strukturált felhasználói felület kialakítását. Az Angular előnye a kétirányú adatbinding, a moduláris felépítés és a gazdag ökoszisztéma, ami megkönnyíti a skálázhatóságot és a karbantartást.
2. **Backend**: A backend fejlesztéséhez a NestJS keretrendszert alkalmazzuk, mely a Node.js alapjaira épül, és modern, típusbiztos architektúrát kínál. A NestJS lehetővé teszi a moduláris, könnyen bővíthető RESTful API-k létrehozását, valamint integrált dependency injection és egyéb fejlesztési minták támogatását, amelyek elősegítik a kód tisztaságát és újrafelhasználhatóságát.
3. **Adatbázis**: Az adatok tárolására az SQLite relációs adatbázis-kezelőt választottuk. Az SQLite egy könnyen telepíthető, beágyazott megoldás, amely ideális kisebb és közepes méretű alkalmazások esetén, egyszerűséget és alacsony erőforrásigényt biztosítva.

# Fejlesztés folyamata, indoklása

## Fejlesztői környezet

A projektünk fejlesztésében két fő eszközt emelnénk ki:

* **Visual Studio Code (VS Code)**A VS Code egy modern, könnyen használható és testreszabható integrált fejlesztői környezet (IDE), amely ideális választás Angular és NestJS alapú fejlesztéshez. Az eszköz gazdag bővítmény-kínálata támogatja a különböző programozási nyelvek és technológiák integrációját, így segítve a gyorsabb és hatékonyabb kódolást. Emellett a beépített hibakereső és Git támogatás is megkönnyíti a fejlesztési folyamatot.
* **Git verziókezelő rendszer**A Git alkalmazása lehetővé teszi a kódváltozások nyomon követését, a verziók kezelését és a csapatmunkát. Az elosztott verziókezelés révén minden fejlesztő saját lokális példányban dolgozhat, miközben a központi repóban történik a végleges integráció. Ez nemcsak a hibák visszakövetését könnyíti meg, hanem az együttműködést és a fejlesztési folyamat átláthatóságát is jelentősen javítja.

Ezek az eszközök együtt biztosítják, hogy a fejlesztési folyamat gördülékeny, átlátható és jól dokumentált legyen, támogatva a hatékony kódolást és a csapatmunkát.

## Backend fejlesztés

### Technológiai alapok és architektúra

A NestJS egy progresszív Node.js keretrendszer, amely robusztus, skálázható és könnyen karbantartható szerveroldali alkalmazások fejlesztésére alkalmas.

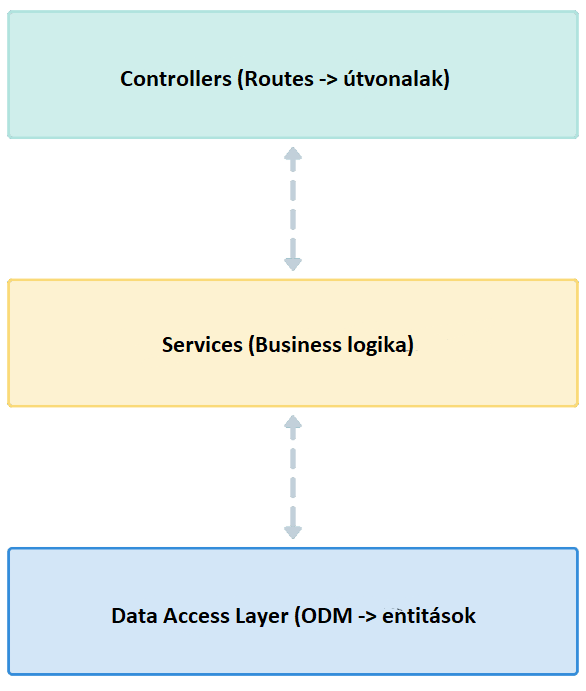
Kombinálja az Objektumorientált Programozás (OOP), Funkcionál Programozás (FP) és Funkcionál Reaktív Programozás (FRP) elemeit egyedi fejlesztési élményt nyújtva.

A Typescript használata és a magas fokú moduláris architektúra révén tiszta, hatékony és biztonságos alkalmazásokat tesz lehetővé.

A NestJS segítségével tiszta, hatékony és biztonságos szerveroldali alkalmazásokat hozhatunk létre, amelyek különösen előnyösek webes API-k, mikroszolgáltatások és valós idejű rendszerek fejlesztésében. Az architektúra modularitása váratlan előnyt jelenthet a fejlesztők számára, például gyorsabb hibajavításban és csapatmunkában.

*Összehasonlítás más Node.js keretrendszerekkel:*

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| Jellemző | NestJS | Express | hapi | Fastify |
| Natív programozási nyelv | TypeScript | JavaScript | JavaScript | JavaScript |
| Függőség injektálás (Dependency Injection) | Igen – beépített | Igen – külön kell megvalósítani | Igen – külön kell megvalósítani | Igen – külön kell megvalósítani |
| Útvonalkezelés | Igen | Igen | Igen | Igen |
| Middleware támogatás | Igen | Igen | Igen | Igen |
| Hibakezelés | Igen | Igen | Igen | Igen |
| Tesztelés | Igen- rengeteg beépített eszközzel | Igen | Igen | Igen |
| GraphQL támogatás | Igen – rengeteg beépített eszközzel | Igen | Igen | Igen |
| Mikroszervíz támogatás | Igen – rengetek beépített eszközzel | Igen | Igen | Igen |
| Közösség | Nagy és aktív | Aktív | Aktív | Aktív |
| Dokumentáció | Kiváló | Jó | Jó | Jó |
| Teljesítmény | Kiváló | Nem rossz | Jó | Jó |
| Skálázhatóság | Kiváló | Jó | Jó | Jó |
| Karbantarthatóság | Jó | Nem rossz | Jó | Jó |

*A NestJS alap architektúrája*

**TypeScript**

Egy olyan környezetben, ahol a JavaScript uralja a szerveroldali fejlesztést, a TypeScript különleges előnyt biztosít alapos típusellenőrzésével és objektumorientált lehetőségeivel. Ez a rész arra világít rá, hogy miért nem csupán hasznos, hanem szinte nélkülözhetetlen választás a TypeScript, ha a NestJS-szel dolgozunk.

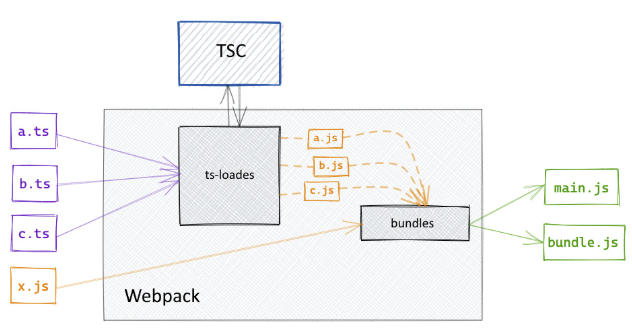
**Miért éppen a TypeScript?**

A JavaScript, bár széles körben elterjedt és sokoldalú, nem nyújt teljes megoldást a típusbiztonság és az öndokumentáló kód terén. A TypeScript, amely a JavaScript egy kiterjesztett változata, a Microsoft fejlesztésében született meg ezeknek a hiányosságoknak a kiküszöbölésére. Ha egy szerveroldali keretrendszer, például a NestJS kontextusában használjuk, a TypeScript előnyei még inkább kiemelkednek.

**Mit kínál a TypeScript?**

A TypeScript statikus típuskezelést hoz a JavaScript világába, lehetővé téve az adattípusok meghatározását már a fordítási időben. Ez számos fontos előnnyel jár:

* **Hibacsökkentés**: A típusokkal kapcsolatos hibák fordítási időben történő kiszűrése rengeteg hibakeresési időt takarít meg, és megelőzheti a rendszer esetleges meghibásodásait.
* **Kódminőség**: A statikus típusozás gyakran olvashatóbbá és öndokumentálóvá teszi a kódot, így a csapat tagjai könnyebben értik meg a kódalap tartalmát.
* **IDE-támogatás**: A modern fejlesztőkörnyezetek (IDE-k) jobb IntelliSense és automatikus kiegészítési funkciókat kínálnak a TypeScript használatakor, ami gyorsabbá és hatékonyabbá teszi a fejlesztést.



*Ábra 1.3: Diagram a TypeScript sima JavaScript-be való fordításáról és a típus-ellenőrzési folyamatáról*

// javascript code

function add(a, b) {

  return a + b;

}

console.log(add(5, "10"));  // Output: 510

*Ez a JavaScript kód egy buggot tartalmaz, mivel egy string-es értéket akar összeadni egy numerikus értékkel.*

// typescript code

function add(a: number, b: number): number {

  return a + b;

}

console.log(add(5, "10"));  // Error: Argument of type

                            // 'string' is not assignable

                            // to parameter of type

                            // 'number'.

!! Figyelje meg, hogyan észleli a TypeScript a típuseltérést a fordítási időben, csökkentve annak esélyét, hogy a hibák bekerüljenek az éles kódba.

*Ez pedig a TypeScript kód, ami megoldja a problémánkat.*

**TypeScript és a NestJS – a tökéletes harmónia**

A TypeScript funkciói tökéletesen illeszkednek a NestJS alapelveihez. Íme, hogyan:

* **Dekorátorok**: A NestJS és a TypeScript egyaránt használ dekorátorokat a metadatok tükrözésére, ami tisztább és rendezettebb kódot eredményez.
* **Erős típusosság**: Ez biztosítja, hogy a NestJS vezérlők, szolgáltatások és egyéb osztályok robusztusak és könnyen karbantarthatók legyenek.
* **Modularitás**: A TypeScript névtere és modulrendszere segít a kód szervezésében, így skálázhatóvá téve azt, ami összhangban van a NestJS moduláris architektúrájával.

**Kódrészlet – NestJs a TypeScript-tel**

Íme egy részlet egy egyszerű NestJs service-ről:

// typescript kód

import { Injectable } from '@nestjs/common';

@Injectable()

export class AppService {

  getData(): string {

    return 'Hello, NestJS!';

  }

}

Ebben a kódrészletben az **@Injectable()** dekorátor és a **: string** visszatérési típus mind példái annak, hogyan segíti elő a TypeScript a NestJS fejlesztést.

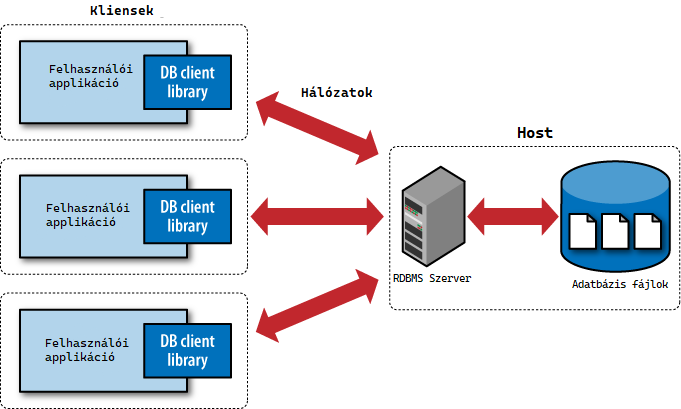
A TypeScript integrálásával a NestJS olyan fejlesztési környezetet kínál, amely robusztus, hatékony, és kevésbé hajlamos a hibákra. A TypeScript statikus típusellenőrzése, fejlettebb IDE támogatása és a modern programozási paradigmákkal való kompatibilitása mind elengedhetetlen részei a NestJS ökoszisztémának.

**SQLite adatbázis**

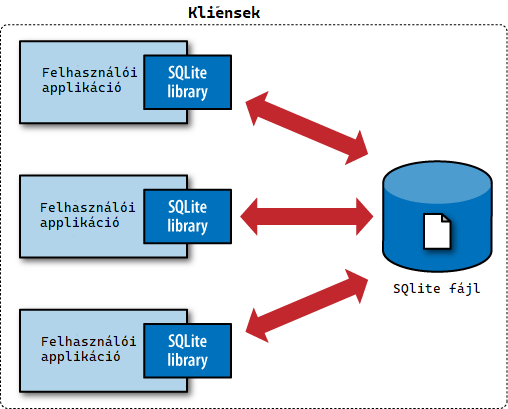
Az adatbáziskezeléshez SQLite-ot alkalmaztunk, mivel a fejlesztési és tesztelési fázis során egy könnyen telepíthető, konfigurálható és fájl-alapú relációs adatbázisra volt szükség. Az SQLite előnye, hogy nem igényel külön adatbázis-szervert, így egyszerűen integrálható és gyorsan beállítható egy prototípus vagy kisebb projektek számára. A későbbi skálázás során szükség esetén egy robusztusabb adatbázis-kezelőre, például PostgreSQL-re lehet áttérni anélkül, hogy jelentős változtatásokat kellene végezni az alkalmazás kódbázisán.

Az SQLite egy beágyazott adatbázis. Nem egy önálló folyamatként fut, hanem az általa kiszolgált alkalmazás folyamatterében, annak szerves részeként működik. A forráskódja közvetlenül beépül a fogadó alkalmazásba, így a külső szemlélő számára láthatatlan marad, hogy az adott program egy relációs adatbázis-kezelő rendszert (RDBMS) tartalmaz. A program egyszerűen végzi a feladatát és kezeli az adatait anélkül, hogy külön figyelmet fordítana arra, hogyan is teszi ezt. A háttérben azonban egy teljes, önálló adatbázis-motor dolgozik.

Az egyik legnagyobb előnye annak, hogy az adatbázis-szerver közvetlenül az alkalmazáson belül fut, hogy nincs szükség hálózati konfigurációra vagy adminisztrációra. Gondoljunk csak bele, milyen felszabadító ez: nincs szükség tűzfalak kezelésére, címfeloldásra vagy összetett jogosultságok beállítására. Az ügyfél és a szerver ugyanabban a folyamatban fut, ami csökkenti a hálózati kommunikációval járó többletterhelést, egyszerűsíti az adatbázis-kezelést, és megkönnyíti az alkalmazás telepítését. Minden szükséges komponens közvetlenül a programba van beépítve.



*Ábra 1.4: A hagyományos relációs adatbázis-kezelő rendszerek kliens/szerver architektúrája, amely klienskönyvtárat alkalmaz.*



*Ábra 1.5: Az SQLite szerver nélküli architektúrája.*

**SQL az SQLite-ban**

Az SQL az egyedüli (és szinte mindenhol alkalmazott) eszköz, amellyel kommunikálni lehet egy relációs adatbázissal. Ez egy olyan nyelv, amely kizárólag az információk feldolgozására szolgál. A nyelvet az információk struktúrázására, olvasására, írására, rendezésére, szűrésére, védelmére, számítására, generálására, csoportosítására, aggregálására és általában véve az információk kezelésére tervezték.

Az SQL egy intuitív, felhasználóbarát nyelv. Használata szórakoztató lehet, és meglehetősen erőteljes. Az SQL egyik érdekes tulajdonsága, hogy függetlenül attól, hogy szakértő vagy kezdő vagy, mindig találhatsz új módokat a dolgok elvégzésére (akár jobb, akár rosszabb eredménnyel). Gyakran többféle megoldás létezik egy adott problémára, és előfordulhat, hogy élvezettel keresed a hatékonyabb módokat a kívánt eredmény elérésére, akár tömörebb kifejezésekkel, akár elegánsabb megoldásokkal, mintha egy rejtvényt oldanál meg. Ráadásul folyamatosan felfedezheted a nyelv olyan elfeledett sarkait, amelyeket eddig nem vettél észre, és amelyek segítségével tovább fejlesztheted a tudásodat, bárhány éves tapasztalattal rendelkezel is.

**Relációs modell**

A relációs modell három alapvető részből áll: forma, funkció és konzisztencia. A forma az információ struktúrájára utal. Egyetlen adatstruktúra létezik, amely minden információt reprezentál. Ezt a struktúrát relációnak nevezik (SQL-ben tábla), amely tuplákból áll (SQL-ben sorok), amelyek pedig attribútumokból (SQL-ben oszlopok) épülnek fel.

A relációs modell formája az információ logikai reprezentációja. Ez a logikai reprezentáció egy tiszta, absztrakt nézetet ad az információról, amelyet semmi nem befolyásol kívülről. Olyan, mint egy matematikai fogalom: tiszta és következetes, egy jól meghatározott, determinisztikus szabályrendszer irányítja, amely nem változik. A logikai reprezentáció teljesen független a fizikai reprezentációtól, amely arra utal, hogyan tárolja az adatbázis szoftver ezt az információt a fizikai szinten (pl. lemez). Így a két reprezentáció különálló: semmi, ami a fizikai szinten történik, nem változtathat meg vagy befolyásolhat semmit a logikai szinten. A logikai szintet nem korlátozzák a hardverek, szoftverek, gyártók vagy technológiák.

A modell második alapvető része a funkcionális rész, más néven manipulációs komponens. Ez határozza meg, hogyan lehet műveleteket végezni az információval a logikai szinten. Ezt formálisan Codd 1972-es „Relational Completeness of Data Base Sublanguages” című cikkében vezette be. A relációs modellt a relációs algebra és a relációs kalkulus meghatározásával bővítette. Ezek két formális, vagy "tiszta" lekérdező nyelv, amelyek erőteljes matematikai alapokon nyugszanak. A relációk, mint ahogy az adatmodell leírja, matematikai halmazok (néhány további tulajdonsággal). A relációs algebra és kalkulus a modellre építve halmazelméleti és formális logikai műveleteket ad hozzá, így képezve a relációs modell funkcionális komponensét. Tehát a relációs modell formája és funkciója közvetlenül a matematikai fogalmakkal származik. Minden egyes származék azonban egy kis pluszt ad hozzá, hogy jobban alkalmazkodjon a számítógépekhez és az információfeldolgozáshoz.

### Backend fejlesztésének folyamata

Az alkalmazásfejlesztés és az informatikai rendszerek integrációja szempontjából az API-k (Application Programming Interfaces) kulcsszerepet játszanak. Ezek az interfészek biztosítják, hogy különböző szoftverkomponensek és rendszerek hatékonyan kommunikáljanak egymással, lehetővé téve az adatok és funkciók megosztását.

Az API-k használata elengedhetetlen a modern fejlesztésben, mivel segítségükkel a fejlesztők gyorsabban és hatékonyabban építhetnek skálázható, moduláris és könnyen karbantartható alkalmazásokat. Különösen fontosak a webes és mobilalkalmazások esetében, ahol az adatok gyakran távoli szerverekről érkeznek, és az API-kon keresztüli kommunikáció biztosítja a zavartalan működést.

A projektünk kezdetekor létrehoztunk az alapvető API-kat, amik a fejlesztéskor nagy segítség lesz. Itt van néhány kódrészlet amiket felhasználtunk a webalkalmazásunkban:

findOne(id: number) {

    return this.telephoneRepository.findOne({ where: { id }, relations: ['specs', 'os', 'display', 'camera', 'battery'] });

  }

Ez a metódus egyetlen Telephone entitást kér le az adatbázisból az id alapján. Emellett a kapcsolódó entitásokat is betölti (specs, os, display, camera és battery) ugyanabban a lekérdezésben, a TypeORM eager loading funkcióját használva.

async search(searchTerm: string): Promise<Telephone[]> {

    if (!searchTerm || searchTerm.trim() === '') {

      return [];

    }

    const queryBuilder = this.telephoneRepository.createQueryBuilder('telephone');

    queryBuilder.leftJoinAndSelect('telephone.specs', 'specs');

    queryBuilder.leftJoinAndSelect('telephone.os', 'os');

    queryBuilder.leftJoinAndSelect('telephone.display', 'display');

    queryBuilder.leftJoinAndSelect('telephone.camera', 'camera');

    queryBuilder.leftJoinAndSelect('telephone.battery', 'battery');

    const loweredSearchTerm = searchTerm.toLowerCase();

    return queryBuilder

      .andWhere('LOWER(telephone.modell) LIKE :searchTerm', { searchTerm: `%${loweredSearchTerm}%` })

      .orWhere('LOWER(telephone.marka) LIKE :searchTerm', { searchTerm: `%${loweredSearchTerm}%` }).getMany();

  }

Ez egy keresőfunkció egy TypeScript osztályban, amely a NestJS alkalmazás része. A megadott keresési kifejezés alapján telefonokat keres az adatbázisban. Az alábbiakban röviden összefoglalva a működését:

* Először ellenőrzi, hogy a keresési kifejezés üres-e vagy csak szóközöket tartalmaz. Ha igen, akkor egy üres tömböt ad vissza.
* Ezután egy lekérdezésépítőt hoz létre az adatbázis-lekérdezés összeállításához.
* A keresés során több kapcsolódó táblát (specs, os, display, camera és battery) is csatlakoztat a telephone táblához.
* A keresési kifejezést kisbetűssé alakítja, és a modell és marka oszlopokban keresi, figyelmen kívül hagyva a kis- és nagybetűk közti különbségeket. A keresés részleges egyezéseket is támogat, mivel a % karaktereket használja helyettesítőként.
* Végül végrehajtja a lekérdezést, és az eredményeket egy Telephone objektumokat tartalmazó tömbként adja vissza.

async findByFilter(termekSzuroDto: TermekSzuroDto): Promise<Telephone[]> {

    const queryBuilder = this.telephoneRepository.createQueryBuilder('telephone');

    queryBuilder.leftJoinAndSelect('telephone.specs', 'specs');

    queryBuilder.leftJoinAndSelect('telephone.os', 'os');

    queryBuilder.leftJoinAndSelect('telephone.display', 'display');

    queryBuilder.leftJoinAndSelect('telephone.camera', 'camera');

    queryBuilder.leftJoinAndSelect('telephone.battery', 'battery');

    if (termekSzuroDto.markak && termekSzuroDto.markak.length > 0) {

      queryBuilder.andWhere('telephone.marka IN (:...markak)', { markak: termekSzuroDto.markak });

    }

    if (termekSzuroDto.minAr !== undefined) {

      queryBuilder.andWhere('telephone.ar >= :minAr', { minAr: termekSzuroDto.minAr })

    }

    if (termekSzuroDto.maxAr !== undefined) {

      queryBuilder.andWhere('telephone.ar <= :maxAr', { maxAr: termekSzuroDto.maxAr })

    }

Ez egy TypeScript osztályban található metódus, amely különböző szempontok alapján szűri a telefonkészülékek listáját. A findByFilter metódus egy TermekSzuroDto típusú objektumot kap paraméterként, amely tartalmazza a szűrési lehetőségeket, például a márkát, az árkategóriát, a RAM-méretet, a tárhelyet és egyéb specifikációkat.

A metódus egy lekérdezésépítőt használ az adatbázis-lekérdezés összeállítására, amely a megadott szűrési feltételek alapján szűri a telefonokat. Egy sor if feltételt alkalmaz, hogy csak azokat a szűrési feltételeket vegye figyelembe, amelyek ténylegesen jelen vannak a TermekSzuroDto objektumban.

A lekérdezés során több kapcsolódó táblát (specs, os, display, camera és battery) is összekapcsol a telephone táblával, hogy részletesebb információt szolgáltasson az egyes készülékekről.

A metódus egy ígéretet (Promise) ad vissza, amely egy olyan Telephone objektumokat tartalmazó tömbre oldódik fel, amely megfelel a megadott szűrési feltételeknek.

Összességében ez a metódus lehetővé teszi a telefonkészülékek széles körű szűrését, így rugalmas és hatékony eszközt biztosít a kereséshez és a megfelelő eszközök lekérdezéséhez.

## Frontend fejlesztés

### Technológiai alapok és architektúra

Az Angular, az egyik legnépszerűbb frontend fejlesztési keretrendszer, egy fordulóponthoz érkezett. Az elmúlt évek során jelentős fejlesztéseken ment keresztül a teljesítmény, a felhasználói élmény és az új funkciók terén – például az Ivy renderelőmotor bevezetésével, amely csökkentette a csomagméretet és javította a futtatási sebességet. Ezek az előrelépések kedvező helyzetbe hozták a keretrendszert.

Mostantól a közösség nemcsak a keretrendszer látható elemeinek fejlesztésére összpontosíthat, hanem olyan tényezőkre is, amelyek közvetlen hatással vannak a felhasználói élményre. Különösen nagy hangsúlyt kaphatnak a fejlesztői élményt érintő fejlesztések, például a jobb skálázhatóság és komponálhatóság. Ezek az újítások különösen értékesek azok számára, akik napi szinten dolgoznak az Angularral. A keretrendszer korábbi verziói már jelentős fejlődést hoztak mind a felhasználói, mind a fejlesztői élmény terén, és a jövőbeni kiadások további fejlesztéseket ígérnek.

**Az Angular alapvető funkciói**

Ebben az írásban az Angular keretrendszer néhány, de nem minden fontos aspektusát tárgyaljuk. Elsősorban azokra a funkciókra összpontosítok, amelyek jelentős átalakuláson mennek keresztül a legújabb kiadások révén.

**Objektumorientált programozás**

Az objektumorientált programozás (OOP) hosszú ideig az összetett vállalati projektek egyik meghatározó jellemzője volt, különösen a Java és C# nyelvek népszerűségének köszönhetően. Ez volt az alapértelmezett megközelítés a bonyolult alkalmazások kezelésére. Az Angular maga is szoros és összetett kapcsolatban áll az OOP-val.

Az Angular fő építőelemei – például a komponensek, pipe-ok, direktívák és guardok – hagyományosan objektumorientált módon készültek. Mindegyik egy osztályként van reprezentálva, az adatok tulajdonságként tárolódnak, a viselkedésüket pedig metódusok határozzák meg. Azonban a következő fejezetekben, különösen a 3. és 10. fejezetben, látni fogjuk, hogy ezek közül az építőelemek közül néhány nem feltétlenül igényli az osztályalapú megközelítést. Sőt, egyes esetekben az osztályok használata akár félrevezető is lehet, különösen azok számára, akik más frontend keretrendszerekből, például a funkcióorientált Reactból érkeznek. Megnézzük, hogyan változik meg ez a paradigma.

**Függőséginjektálás**

A függőséginjektálás (DI) az Angular egyik legfontosabb és legvonzóbb jellemzője, amelyet szinte minden Angular fejlesztő használt már valamilyen projektben. Korábban a DI teljes mértékben össze volt kötve az osztályokkal és az objektumorientált programozással: a függőséginjektáláshoz elengedhetetlen volt egy osztály az @Injectable dekorátorral. Ebből adódóan egy szolgáltatás, konfiguráció vagy más DI-fában lévő elem nem volt elérhető egy függvényből, hacsak azt kifejezetten paraméterként nem adtuk át. Ez a megkötés azonban csökkentette a komponálhatóságot. Az új inject függvény bevezetésével ez a korlátozás megszűnt, megnyitva az utat egy rugalmasabb és újrafelhasználhatóbb megközelítés felé.

**Modulalapú architektúra**

A 14-es verzió előtt minden Angular alkalmazás az NgModules köré épült. Az NgModule egy speciális Angular koncepció, amely egy osztályba szervezte az alkalmazás többi építőelemét, biztosítva azok együttműködését. Az NgModule-ok kulcsszerepet játszottak az architektúra kialakításában és a funkcionalitások megosztásában. Ugyanakkor sok fejlesztő problémákba ütközött az NgModules használata során. Az újabb Angular verziók lehetővé teszik, hogy az alkalmazások NgModules nélkül is épülhessenek, egy úgynevezett „standalone” megközelítést alkalmazva.

**RxJS**

Az RxJS, a JavaScript reaktív kiterjesztési könyvtára, kulcsszerepet játszik az Angular alkalmazások állapotmegosztásában. Például az autentikációs és jogosultságkezelési események (hozzáférés engedélyezése vagy visszavonása) gyakran ezen keresztül propagálódnak. Az alkalmazások többsége vagy egy egyszerű állapotkezelési megoldást épít egy szolgáltatáson és egy Subjecten keresztül, vagy egy már meglévő állapotkezelő könyvtárat, például az NgRx-et használja, amely szintén RxJS-re épül. Az új Angular verziók jelentősen növelték az Angular és az RxJS közötti interoperabilitást.

**Változásérzékelés**

A változásérzékelés, amely biztosítja, hogy a komponensek adatváltozásai megfelelően megjelenjenek a felhasználói felületen, egy összetett és nem mindig optimális algoritmus. Jelenleg egy külső könyvtárra, a zone.js-re támaszkodik, amely sok fejlesztő számára jelentős többletterhelést okoz. Sokan keresik azokat a megoldásokat, amelyekkel ki lehet kerülni ezt a réteget. A következő fejezetekben részletesen bemutatjuk a változásérzékelés működését, valamint azt, hogyan lehet hatékonyabbá tenni a folyamatot.

**Modern Angular app készítése**

Az Angular-projektek létrehozására többféle módszer létezik, beleértve különböző egyedi és harmadik féltől származó builderek, bundlerek és egyéb eszközök használatát. Bár ezek az eszközök kétségtelenül hasznosak, ebben a könyvben nem foglalkozunk velük. Fontos megjegyezni, hogy a teljes könyv során kizárólag az Angular hivatalos CLI eszközét fogjuk használni.

Az Angular CLI számos különböző parancsot és egyedi sémát tartalmaz. A könyvben fokozatosan megismerkedünk különféle – köztük néhány újabb – parancsfájlokkal, amelyek lehetővé teszik számunkra, hogy adott beállításokkal építsünk projekteket, környezeti fájlokat generáljunk, meglévő kódokat migráljunk, és így tovább.

Most azonban a legismertebb parancsra, az ng new-re összpontosítunk, amely új projektek létrehozására szolgál. Az ng new több testreszabási lehetőséget kínál, amelyeket az Angular dokumentáció erre dedikált szekciója részletesen ismertet.