Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Elektronikai Technológiai Tanszék

Kenyeres Kristóf

AJÁNLATKEZELŐ ALKALMAZÁS FEJLESZTÉSE NODEJS ALAPOKON, REACT FRONTENDDEL

KONZULENS

Dr. Villányi Balázs

BUDAPEST, 2021

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Kenyeres Kristóf**, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy hitelesített felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Kelt: Budapest, 2021. 05. 05.

...……
Kenyeres Kristóf

Összefoglaló

A mai világban, egy vállalkozás sikeres és eredményes működéséhez elengedhetetlen a megfelelő informatikai háttér, mind a vállalat belső folyamatainak menedzseléséhez, mind pedig a vevőkkel való kapcsolattartáshoz szükség van különböző informatikai rendszerekre. A diplomamunkám ezen részének keretein belül egy felületkezelő vállalat számára készített ajánlatkezelő rendszer megtervezésének a folyamatát, valamint a munkám során szerzett tapasztalataimat szeretném bemutatni.

Ez a felületkezelő vállalat eddig nem használt semmilyen rendszert az ajánlatok kezelésére, mindent manuálisan készítettek, tároltak, és ez eleinte rendben is volt. Ám az utóbbi években nagy fejlődésen mentek keresztül és így már olyan nagy vevőkörük lett, hogy egyszerűen túl sok energiát emésztett fel az ajánlatkérések kezelése. A programom ezeknek a problémák megoldására készül.

Manapság már egyre inkább elfogadottabb az otthoni munkavégzés is, ezért egy fontos kitétel volt, hogy a programot el lehessen érni az interneten keresztül is, akár egy mobiltelefonon is. Így először annak néztem utána, hogy milyen platformon tudnám megvalósítani a programot, úgy, hogy minden követelményt tudjak teljesíteni. Végül egy weben elérhető alkalmazás mellett döntöttem.

Több technológia is megfelelő lenne a program elkészítésére, így először ezeket hasonlítottam össze részletesen, mind a frontend, mind pedig a backend oldalon. Valamint még az adatbázis elkészítése is több módon lehetséges lenne, ezeket a lehetőségeket is megvizsgálom és ezen vizsgálat alapján választottam ki a ténylegesen használt technológiákat.

A dolgozatomban ezen részében bemutatom, hogy hogyan zajlik jelenleg az ajánlatadási folyamat a vállalatnál és milyen főbb problémák vannak vele, és ezekre milyen megoldásokat találtam. Ezt követően bemutatom, hogy milyen technológiák közül választhattam és miért azokat választottam, amit. Továbbá ismertetem a megoldandó problémákat és a vállalat részéről felmerült igényeket és az ezekre adott megoldásokat. Valamint ezek után még az adatbázis megtervezésének a folyamatáról lesz szó.

Abstract

In today's world, the proper IT background is essential for the successful and efficient operation of a business, and various IT systems are needed both to manage the company's internal processes and to communicate with customers. Within the framework of this part of my dissertation, I would like to present the process of designing a quote management system for a surface treatment company and my experience gained during my work.

This surface treatment company has not used any system to handle quotes so far, everything was created and stored manually, and that was fine at first. But they have undergone great development in recent years and have already had such a large customer base that it has simply consumed too much energy to handle requests for quotations. My program is designed to solve these problems.

Nowadays, working from home is also becoming more and more accepted, so it was an important clause that the program could be accessed via the Internet, even on a mobile phone. So I first looked at what platform I could implement the program on so that I could meet all the requirements. Eventually, I decided on an application available on the web.

Several technologies would be suitable for building the program, so I first compared them in detail, both on the frontend and backend side. As well as even creating a database would be possible in several ways, I will examine these possibilities and select the technologies actually used based on this study.

In this part of my dissertation, I present how the quoting process is currently going on at the company and what are the main problems with it, and what solutions I have found for them. Then I show what technologies I could choose from and why I chosed the React and the NodeJS. I also describe the problems to be solved and the needs of the company and the solutions given to them. And after that, it will be a process of designing the database.

1 Bevezetés

Manapság a vállalatok mindennapjai szorosan összefonódtak az informatikával, nélküle már szinte működésképtelenné válnának, mivel a legtöbb folyamataikat már ezen a platformon végzik különböző rendszerekben. Egy vállalaton belül a méretétől függően több rendszert is használnak/használhatnak. Az egyik leggyakoribb ilyen rendszer lehet egy termeléstervező rendszer vagy egy pénzügyi nyilvántartó rendszer, vagy a mi esetünkben egy ajánlat menedzselő rendszer is.

1.1 Vállalat tevékenységének bemutatása

A vállalat, amely számára a program készül Magyarországon az egyik legnagyobb felületkezelési szolgáltatásokat nyújtó vállalat. Felületkezelésre a fém alkatrészek esetében van szükség, hogy a legyártott termékek ellenállóak legyenek a környezeti behatásoknak. Az egyik ilyen behatás az oxidáció, azaz a rozsda. Ez akkor alakul ki, amikor az oxigén reakcióba lép a fémekkel. Ennek kivédésére szolgálnak a felületkezelési technológiák. Több ilyen technológia is létezik, ezeknek felhasználása attól függ, hogy milyen fémből készült az alapanyag, illetve még fontos az is, hogy hol lesz felhasználva az adott termék. A technológia kiválasztása után, az egyéb paramétereket különböző szabványok határozzák meg. Ezek a szabványok alapján megkapjuk, hogy milyen eljárással, milyen rétegvastagságú, és bizonyos esetben milyen színű legyen a bevonat.

1.2 Miért van szükség erre a rendszerre

Az utóbbi pár évben a vállalat sokat fejlődött, növekedett sok új vevőt szerzett, és sok új árajánlat kérést is kap nap mint nap. Ezzel a növekedéssel nem tudott lépést tartani a jelenlegi árajánlat készítési folyamat, elérte a korlátait és gazdaságosan már nem lehetne bővíteni az erőforrásokat. Ezen okokból kifolyólag az egész ajánlatképzési folyamat nagyon lelassult és nem is nyomon követhető. A potenciális vevőknek sokszor több napot is kell várniuk, mire megkapják az ajánlatot a kért termékek felületkezelésére, és így akár még az is előfordulhat, hogy így nem sikerül megszerezni az adott munkát. Vagy adott esetben, ha egy vevő kétszer is kért ajánlatot már, olyan is előfordulhat, hogy két különböző árat kapott rá, és ebből lehetnek fennakadások.

1.3 Célok meghatározása

A végső cél egy olyan webes alkalmazás létrehozása, amely alkalmas lesz arra, hogy lefedje a vállalat ajánlatadási folyamatát, megfeleljen az érvényben lévő szabályozásoknak, és segítse az alkalmazottak munkáját, valamint biztosítsa, hogy az ajánlatok visszamenőleg nyomon követhetőek legyenek. Továbbá fontos szempont, hogy az alkalmazást bárhonnan el tudjuk érni, így akár távmunka esetén is tudnak dolgozni az alkalmazottak, ami manapság szintén fontos lehet.

2 A vállalat jelenlegi ajánlatadási folyamatának bemutatása

Ebben a fejezetben a vállalat jelenlegi ajánlatadási folyamatába nyerünk betekintést. Az elején bemutatom a jelenlegi folyamatokat, illetve az ajánlat előállítás menetét, az ehhez használt programokat, majd ezt követően azonosítom a jelenlegi folyamatban a problémákat és az optimalizációs lehetőségéket. Végül pedig felállítom a követelményeket és definiálom megvalósítandó funkciókat.

2.1 Ajánlatok készítésének folyamata

Jelenleg a vállalaton belül az ajánlatadási folyamat a következőképpen zajlik. A jelenlegi partnerek vagy az érdeklődők egy vagy több konkrét alkatrész felületkezelésére kérnek ajánlatot. Ezt általában emailben történik, amihez csatolnak egy műszaki rajzot arról az alkatrészről, amire a szolgáltatást igénybe szeretné venni. Mivel többféle felületkezelési technológiával is rendelkezik a vállalat, így a technológia kiválasztását is általában az ügyfél végzi el, ritkán előfordulhat, hogy ezt a vállalatnak kell megadnia az ügyfél által elmondottak alapján, például az alapanyag és a felhasználási mód és hely szerint. Vannak továbbá olyan esetek, amikor az ügyfél pontosan definiálja a követelményeket szabványok segítségével. Ennek az azonosítóját megadja és ebből, a vállalat minden szükséges információt meg tud keresni. Ilyen például a felületet bevonó réteg vastagsága, a technológia, az elvárt időtartamú korrózió mentesség.

A jelenlegi módszer szerint a vállalat munkatársai ilyen esetben kikeresik a sok dokumentum közül a szabványt és manuálisan megnézik az az által előírt értékeket, ami alapján az ajánlatban szereplő árat meg tudják határozni.

Azonban még mielőtt a konkrét árat kiszámítanák, előtte el kell végezni egy gyárthatósági tesztet, ami azt jelenti, hogy megvizsgáljuk az ajánlatban szereplő alkatrészeket és ezeknek a befoglaló méreteit. Ez alapján eldöntjük, hogy az adott felületkezelési eljárást el tudjuk-e végezni rajta, azaz, hogy nem-e túl nagy a mérete vagy, hogy az alkatrész alapanyaga megfelel-e a technológia előírásainak, mivel nem minden alapanyagra lehet elvégezni a különböző technológiákat. Továbbá az alkatrészek gyártásához szükség van különböző szerszám keretekre is. Bizonyos tételeknél főleg nagyobb mennyiség esetén előfordul, hogy az alkatrész kialakítása miatt szükséges

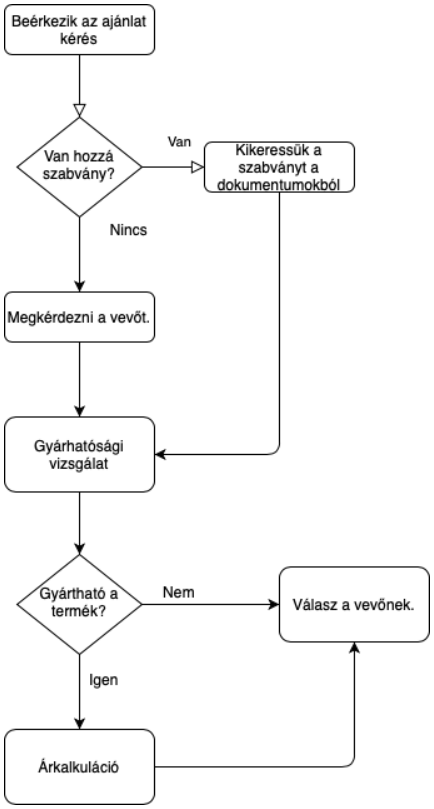
egyedig szerszámot gyártani, ami plusz költség, és ezt a vevőnek kell fizetnie vagy az árba be kell építeni.

Ha a gyárthatósági vizsgálat során megfelelt az alkatrész akkor kezdődik az árkalkuláció, amit a következő adatokon alapul. Egyszer figyelembe kell venni, hogy milyen technológiai eljárást kell alkalmaznunk, aztán az alkatrész felületét és súlyát kell figyelembe venni. Nagyon kicsi alkatrészek esetén melyeket nagy mennyiségben kell gyártani, például csavarok azokra általában súlyra adunk árat. Nagyobb, igényesebb alkatrészeknél pedig mennyiségre adunk árat. Továbbá figyelembe kell vennünk, hogy egy keretre hány darab alkatrészt tudunk feltenni, illetve, hogy át kell-e vizsgálni az eljárás végén az alkatrészeket. Ezen adatok alapján számoljuk ki az árat.

Az eljárások végén lehetőség van plusz szolgáltatás igénybevételére, például lakkozásra vagy olajozásra, ezeknek plusz költsége van, ami egyedileg kerül meghatározásra.

Sok külföldi vállalattal is kapcsolatban van a cég, így sokszor az árakat EUR-ban kell megadni, aminek az átváltása forintra vagy fordítva sokszor időigényes, mivel a napi árfolyamot mindig ki kell keresni és manuálisan kellett átszámolni.

Ha a végső ár és feltételek meghatározásra kerültek, akkor az ajánlatot összeállító munkatárs emailben válaszol az ajánlatkérésre.



1. ábra Jelenlegi ajánlatadási folyamat

2.2 Problémák és javaslatok

A jelenlegi folyamat több szempontból sem optimális, mivel minden manuálisan történik lassú és sokszor körülményes az ajánlatok megírása, ezért előfordulhat, hogy a potenciális vevőknek sokáig kell várnia mire megkapja az ajánlatot. Ennek kiküszöbölésére az alkalmazásom bekéri az adatokat és a megadott adatok alapján automatikusan kiszámolja majd az árat és akár plusz szolgáltatások ajánlására is képes lesz. A szabványokat is be lehet majd tenni az alkalmazásba, amire lehet majd keresni az azonosító száma alapján, így nem lesz szükség több dokumentumon át keresgélni, hanem minden egy helyen lesz tárolva.

Másik probléma az, hogy a kiküldött ajánlatokat nem lehet nyomon követni, mivel az egész folyamat email-eken keresztül zajlik, így az egyetlen lehetőség a nyomon követésre, ha visszakeressük az emaileket, de ez nem életképes megoldás. Előfordulhat, hogy egy alkatrészre több vállalat is kér ajánlatot és sokszor különböző árakat kapnak, ami kellemetlen lehet. Ennek kiküszöbölésére azt javasolom, hogy minden elküldött ajánlatott eltárolunk az alkalmazásban, és kereshetünk közöttük a rajzszám vagy más adatok alapján. Illetve ha egy ajánlatot készítünk és megadjuk a rajzszámot, akkor a program figyelmeztet, minket ha volt már korábban ajánlat kérve erre az alkatrészre.

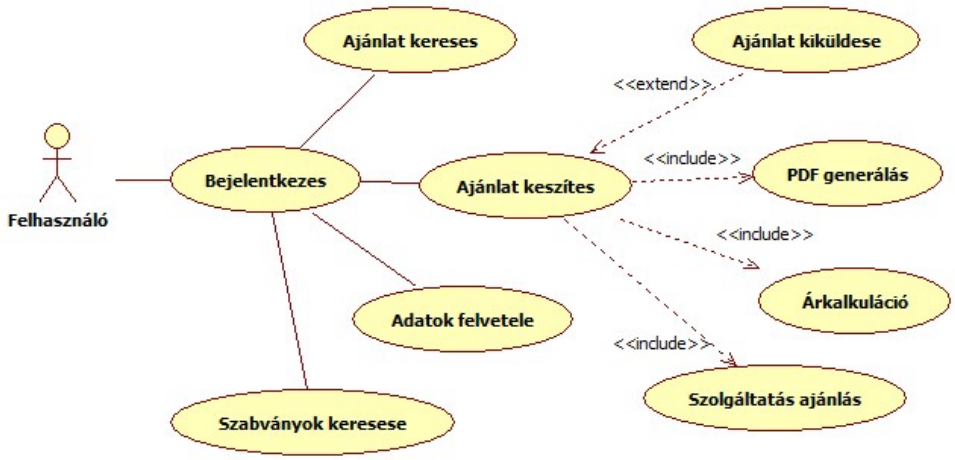
További probléma a jelenlegi módszerrel a különböző pénznemek közötti váltás sem pontos minden esetben, mivel ugye ez is manuális módon zajlik, erre egy olyan megoldást javaslok, hogy napi szinten lekérjük az árfolyamot egy webszolgáltatástól és azzal automatikusan átváltjuk HUF és EUR között az értékeket.

2.3 Követelmények az új rendszerrel szemben

Az alkalmazásomnak képesnek kell lennie a jelenlegi folyamatot kiváltania és nagyban felgyorsítania és biztonságosabbá tennie, valamint plusz szolgáltatásokat is ajánlani, ha lehetséges.

Meg kell valósítania egy authentikációt, amivel a felhasználók be tudnak jelentkezni és ajánlatokat kiadni, így azonosítható lesz, hogy ki küldte ki az adott ajánlatot. Az alkalmazásnak tárolnia kell a technológiai szabványokat és keresési lehetőséget kell biztosítani az azonosító szám alapján, valamint ehhez kell egy megjelenítési felület.

Továbbá az alkalmazásnak automatikusan ki kell számolnia egy árat a megadott adatok alapján, de ez utólag lehessen manuálisan változtatni. A kalkulált ára, az utóbbi időszak átlag árfolyamán át kell váltani EUR pénznemre is, azért az átlagot vesszük alapul, hogy ha van egy hirtelen erősödés vagy gyengülés a devizánál az ne befolyásolja jelentősen az árat. Az elkészült ajánlatból lehessen generálni egy PDF formátumú dokumentumot, ami emailbe ki is lehessen küldeni, vagy le lehessen tölteni a gépünkre.



2. ábra Use Case diagram az alkalmazás funcióiról

3 Adatbázis technológiák

Az alkalmazásomban szereplő adatok hatékony tárolására mindenképpen szükségem lesz valamilyen adatbázisra. Alapvetően kétféle adatbázis technológiát vizsgáltam meg, amelyekkel a későbbiekben megvalósíthatom az alkalmazásom adatbázisát. Az egyik ilyen a NoSQL típusú adatbázisok a másik pedig az általam már sokszor használt és jobban ismert SQL adatbázisok voltak. Ebben a fejezetben összehasonlítom ezt a két adatbázis megvalósítási módszert, ismertetem az előnyeiket és a hátrányaikat is, valamint végül megnézem, hogy melyik módszer lenne az alkalmasabb számomra az alkalmazásom adatbázisának megvalósítására. Ezek után pedig ismertetem, hogy melyik technológiát választottam és milyen szempontok játszottak fontos szerepet ebben a választásban.

3.1 NoSQL adatbázisok

Manapság nagyon sok adat keletkezik és a jövőben egyre több adat fog keletkezni, a növekedés mértéke olyan nagy, hogy a ma tárolt adatok nagy része az utóbbi 2-3 évben keletkezett. Továbbá megjelentek időközben a mobil eszközök is, amik egyre népszerűbbek, és egyre elterjedtebbek, valamint a webes alkalmazások és a felhős rendszerek is megjelentek. Ezeknek a rendszerek többek között gyors válaszidőkre, magas rendelkezésre állásra és horizontális skálázhatóságra van szükségük. Ezek teljesen új kihívások elé állították az adatbázisokat. Ezeknek a kihívásoknak a megoldására jöttek létre a NoSQL adatbázisok. A NoSQL adatbázisok már az 1960-as években is léteztek, de igazán csak most kezdenek népszerűvé válni, az új követelmények megjelenésével.[1]

A NoSQL rendszerek megalkotásának a célja, egy olyan adatbázis rendszer létrehozása, ami jól skálázható, akár több száz szerveren is. A NoSQL rendszerek leggyakoribb tulajdonságai a következők[2]:

- horizontális skálázhatóság
- nyílt forráskódúak
- elosztott működés támogatása
- sémamentesség, szabad forma
- egyszerű API (programozói interfész).

A NoSQL adatbázisok egyik jellemzője a sémamentesség, azaz, hogy szabad formában tudjuk az adatainkat tárolni. Azonban az adatok tárolására így is van négy különböző modell, ami nagyvonalakban iránymutatást ad, hogyan tároljuk a rekordjainkat. Ez a négy modell a következő:

- Dokumentum adatbázis – szabad formájú JSON fájl
- Kulcsértékű üzletek – értékek kulcs szerint érhetőek el
- Széles oszloptárolók – sorok helyett oszlopok
- Grafikus adatbázisok – adatokat grafikonként tároljuk

Ez a négy modell közül én most részletesebben csak a Dokumentum adatbázisokkal foglalkoztam, mivel úgy gondoltam számomra ez lehetne a legmegfelelőbb.

Dokumentum adatbázisokban a mezőket vagy adatértékeket egy dokumentumként hivatkozott entitásban tároljuk el. Ez jellemzően JSON formátumban történik. A dokumentumok mezőiben szereplő adatok többféle formátumban lehetnek, például JSON, XML vagy sima szövegként is tárolhatjuk őket, sőt ezeket variálhatjuk is. Egy dokumentum általában egy entitás összes adatát tartalmazza. Az entitás tartalma az adatbázist használó alkalmazástól függ. Minden dokumentumnak van egy kulcsa, aminek segítségével le tudjuk kérni a dokumentumokat. A dokumentum típusú adatbázisok a következőképpen néznek ki[3]:

Kulcs	Dokumentum

100	<pre>{
	  "BookId": "001231",
	  "author": {
	  "name": "Kristóf"
	  },
	  "type": "Sci-fi"
	}</pre>

101	<pre>{
	  "BookId": "001232",
	  "author": {
	  "name": "Sándor"
	  },
	  "type": "Horror"
	}</pre>

Összegezve a NoSQL adatbázisok alkalmas a nagy tömegű, nem kapcsolódó, gyorsan változó adatok kezelésére, sémafüggetlen adattárolás megvalósításával. Elsősorban olyan alkalmazásoknál érdemes használni, ahol a teljesítmény és a rendelkezésre állás fontosabb, mint a konzisztencia. Ilyen alkalmazási helyzetek lehetnek például a mobil alkalmazások, valós idejű elemzések.

3.2 SQL adatbázisok

Az SQL azaz a relációs adatbázisok már az 1970-es évek óta léteznek, az alapjait az IBM fektette le. Maga az SQL a relációs adatbázisok lekérdező nyelve, azonban különböző típusai léteznek, melyeknek lehet különböző kulcs szavai, így azok nem feltétlen kompatibilisek egymással.

A relációs adatbázisok tábla alapúak, a tábláknak vannak attribútumai és a táblák között lehetnek kapcsolat. A relációs adatbázisokban így történik az adatok tárolása ellentétben mondjuk a NoSQL-el, ahol például egy kulcshoz tartozik egy dokumentum. Lehetőség van még indexek használatára, ezzel gyorsíthatjuk a keresést az adatbázisban. Alkalmazhatunk kényszereket is, ezek korlátozó szabályok az adatbázisra, például, hogy egy adott tábla adott mezője, milyen értékeket vehet fel.

A relációs adatbázisok azonban kötöttebbek, mint a nem relációs adatbázisok. Itt fontosak az ACID tulajdonságok megtartása. Az ACID tulajdonságok a következők:

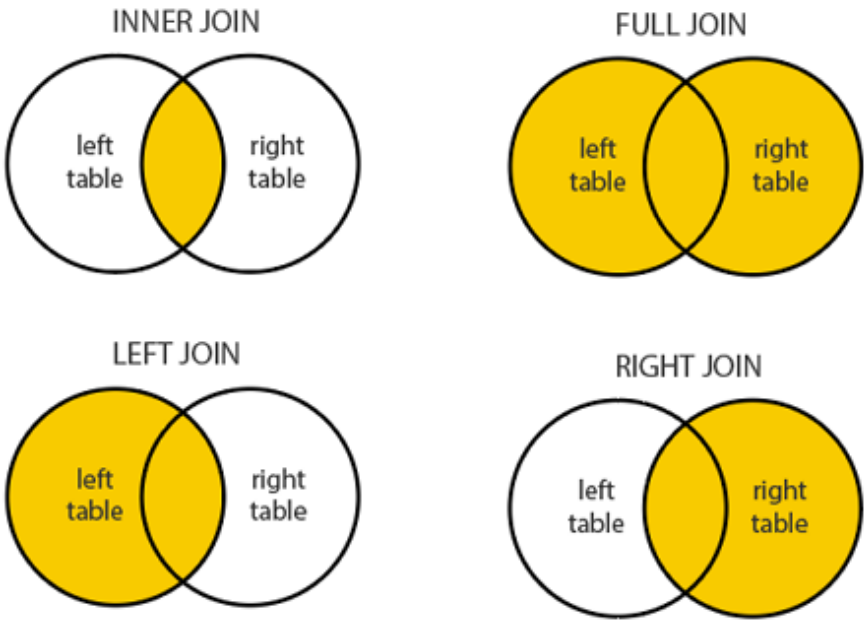
- Atomicitás: Vagy az összes művelet végrehajtódik vagy egyik sem.
- Konzisztencia: Adatok érvényes állapotba legyen tranzakció előtt és után is.
- Izoláció: Egy tranzakció hatása nem látható a többi tranzakcióból.
- Tartósság: Tranzakciók változásait el kell tartós módon tárolni.

Elosztott rendszerek esetében azonban nehéz relációs adatbázisok készíteni, mivel bonyolult lekérdezéseink lennének, valamint az ACID tulajdonságokat is nehéz lenne garantálni. Ami magát az SQL nyelvet illeti egy egyszerű lekérdezés a következőképpen néz ki:

```
SELECT mezőnév FROM táblanév WHERE feltétel
```

Lehet Joinokat is alkalmazni a táblák között, ezzel kombinálhatunk sorokat kettő vagy több táblából is. Többféle join műveletet tudunk megkülönböztetni.

- Inner join: Mindkettő táblában van megegyező érték.
- Left join: Az összes érték a bal oldali táblából és a megegyező értékek a jobb oldali táblából.
- Right join: Az összes érték a jobb oldali táblából és a megegyező értékek a bal oldali táblából.
- Full join: Minden értéket visszaad mindkét táblából, ha van egyezőség.



3. ábra Join típusok [5]

```
SELECT Orders.ID,Customers.Name,Orders.Date
FROM Orders
INNER JOIN Customers ON Orders.CustomerID=Customers.ID;
```

Nem az adatbázis, hanem az adatbáziskezelők részeként lehet még használni Triggerek is, amik egy adatbázisban történő változtatás hatására véghez visznek valamilyen folyamatot.

Az SQL adatbázisok létrehozás előtt szükséges, hogy legyen egy részletes adatbázis modellünk, ami alapján elkészíthetjük az adatbázist, hiszen későbbiekben a változtatások az egész sémára hatással lehetnek.

3.3 Az én választásom

Miután megvizsgáltam a relációs és a nem relációs adatbázisok nyújtotta előnyöket és hátrányokat, így arra a következtetésre jutottam, hogy számomra és az alkalmazásom számára a relációs adatbázis lehet a jobb megoldás. Fontos, hogy az adatok konzisztensek legyen, az ajánlatok vagy a szabványok tárolásánál. Ugye az adatbázis létrehozása előtt szükséges, egy részletes adatmodell is, amit én definiálni tudok még az implementáció előtt. Továbbá arra a következtetésre jutottam, hogy a nem relációs adatbázis nyújtotta előnyöket az én alkalmazásom nem használná ki teljes egészében.

Valamint a döntésemben az is fontos szerepet játszott, hogy az SQL nyelvet már jól ismerem, van benne tapasztalatom, így biztos vagyok benne, hogy hatékonyabb és jobb adatbázist tudok készíteni a MySQL segítségével, mint egy NoSQL adatbázissal, például a MongoDB-vel.

4 Frontend technológiák

Az alkalmazásom felhasználói interfészének megalkotásához a React az Angular és a Vue keretrendszerek közül kellett kiválasztanom azt, amelyikkel a legkönnyebben és a leghatékonyabban tudom elkészíteni a felhasználói felületet, valamint fontos még, hogy az alkalmazáshoz használt többi technológiával is egyszerűen lehessen integrálni. Ebben a fejezetben ismertetem ezen keretrendszerek működését, sajátosságait. Először mind a három keretrendszert bemutatom általánosan röviden, majd a főbb témakörök szerint részletesen is bemutatom őket, majd a legvégén konklúziót vonva kiválasztom azt, ami számomra a legjobban passzol.

4.1 React.js

A React egy nyílt forráskódú JavaScript könyvtár, amit felhasználói felületek készítésére terveztek és hoztak létre. A React-et a Facebook hozta létre. Kiválóan alkalmas úgynevezett single-page vagy mobil alkalmazások fejlesztésére. Mivel a React nem egy keretrendszer csak egy könyvtár ezért itt nem kapunk meg minden olyan funkcionalitást, mint más keretrendszereknél. Ezért React alkalmazások fejlesztésénél gyakran igénybe kell venni más könyvtárakat is, például a routinghoz, vagy szerver oldali funkciókhoz. Nem rendelkezik saját HTTP klienssel. Azonban, amire kiváló az az, hogy tartalmat jelenítsünk meg a DOM segítségével és ezt hatékonyan ellenőrizzük.

4.2 Vue.js

A Vue.js egy a felhasználói felületek létrehozására alkalmas framework, ami a JavaScript programnyelvre épül. A Reacttel ellentétben ez egy keretrendszer nem pedig egy JavaScript könyvtár, ezáltal több funkcióval rendelkezik. Rendelkezik beépített állapot menedzsmenttel, ami a React-ben csak hellyel közzel létezik és van benne routing is, ami a Reactből teljesen hiányzik. Azonban ez sem rendelkezik saját HTTP klienssel és ebben sem található meg az űrlapok menedzselésének lehetősége. Vue.js keretrendszer használata lehetővé teszi számunkra, hogy deklaratívan rendereljünk ki információt a felhasználói felületre, a DOM segítségével.

4.3 Angular.js

Az Angular.js egy keretrendszer, amit dinamikus webalkalmazások készítéséhez használnak. A Google fejlesztette ki, hogy egyszerűsítse a webalkalmazások frontendjének fejlesztését. A dolgozatomban taglalt három technológia közül egyértelműen ez a legnagyobb, abban az értelemben, hogy ez rendelkezik a legtöbb funkcióval, valamint támogatás is van hozzá, amit a Google nyújt. A fejlesztés HTML és TypeScript nyelven történik. Az Angularban megtalálható az összes olyan funkció, ami az előző kettő technológiában, plusz az Angular képes HTTP kérések küldésére, űrlapok kezelésére, valamint van hozzá egy CLI is. Az Angularban található adatösszekötési módszer és a függőség injektálás miatt, sokkal csökkent a megírandó kód mennyisége. Ezeknek elvégzése browser oldalon történik, így a legtöbb szerver oldali technológiával használható.

4.4 Szintaktika

Mind a három keretrendszer a JavaScript programozási nyelvre épül, de azért vannak különbségek közöttük. Ezeket a különbségeket és a keretrendszerek szintaktikai sajátosságait mutatom be ebben a fejezetben.

A React JavaScriptet és a JSX-et használ. A JSX használata lehetővé teszi, hogy JavaScript környezetben HTML elemeket használjunk és hozzáadjuk a DOM-hoz, anélkül, hogy használnunk kellene a createElement() vagy az appendChild() metódusokat. Ez nagyban megkönnyítheti a fejlesztést.[6]

```
const element = <h1>Hello JSX!</h1>;
const element = React.createElement('h1', {}, 'NO JSX!');
```

Az Angular egy TypeScript nevezetű programozási nyelvet, valamint HTML-t használ. A TypeScript alapja a JavaScript, de van pár alapvető különbség. A TypeScriptet nem tudják a böngészők közvetlenül futtatni, ezért először lefordítjuk JavaScriptre, így már tudják értelmezni a kódot a böngészők. Másik különbség, hogy a TypeScript objektum orientált nyelv, míg a JavaScript egy script nyelv. A TypeScriptba van egy olyan funkció, hogy statikus gépelés, ami azt jelenti, hogy ha van egy változónk annak nem változathatjuk meg utána a típusát. Például, ha egy változónak egyszer már adtunk egy szám értéket utána, nem adhatunk neki szöveget. TypeScripten vannak interfészek is, amiket tudunk használni. De összességében azt mondhatjuk, hogy a TypeScript az egy kibővített JavaScript.[12]

A Vue.js sima JavaScriptet és HTML-t használ, azonban külön választja a kettőt. Az alábbi táblázatban mind a három technológiával az alap Hello World program látható.

Technológia	Hello World példa

React.js	`ReactDOM.render(`
	`  <h1>Helló, világ!</h1>,`
	`  document.getElementById('root')`
	`);`

Vue.js	`HTML:`
	`<div id="app">`
	`  {{ message }}`
	`</div>`
	`JS:`
	`var app = new Vue({`
	`  el: '#app',`
	`  data: {`
	`  message: 'Hello Vue!'`
	`  }`
	`})`

Angular.js	`HTML:`
	`<h1>{{title}}</h1>`
	
	`TS:`
	`import { Component } from '@angular/core';`
	
	`@Component({`
	`  selector: 'app-root',`
	`  templateUrl: './app.component.html'`
	`})`
	`export class AppComponent {`
	`  title = 'Hello World!';`
	`}`

4.5 Data Binding

Először is azt kell megértenünk mit is értünk data binding alatt. Az informatikában data binding alatt azt a folyamatot értjük, ami kapcsolatot biztosít az alkalmazás felhasználói felülete és a megjeleníteni kívánt adat között. Ennek segítségével tudjuk elérni, hogy ha a megjelenített adat megváltozik, akkor a felületen automatikusan újra legyenek generálva az elemek. Létezik egyirányú és kétirányú megoldás is, egyirányú amikor csak az adat megváltozása idézi elő a felület frissítését, kétirányú esetében azonban a felhasználói felületen történő változás is módosíthatja az adatot. Például egy

szöveg beviteli mezőbe, ha módosít a felhasználó, akkor az a változóban is megváltozik.[15]

React.js esetében alapból csak egyirányú data binding elérhető. Amennyiben szükségünk van kétirányú data bindingra, például egy formot készítünk és a felhasználói input hatására módosítani akarunk valamit az alkalmazásunkban, akkor ezt tudjuk implementálni a LinkedStateMixin használatával.[6]

```
constructor() {
  super();
  this.state = { value: 'Hello World' };
}
Change = (e) => {
  this.setState({ value: e.target.value });
};
render() {
  return (
    <div>
      <inputtype="text"value={this.state.value
onChange={this.Change}/>
      <p>{this.state.value}</p>
    </div>
  );}
```

Az Angular.js három féle data binding megoldást biztosít számunkra, az adatáramlás iránya szerint. Egyszer lehet az adatforrástól a nézet irányában, lehet fordítva, azaz a nézet irányából az adatforrás felé, illetve lehet a kétirányú megoldás. Ezeknek a használata különböző szintaktikai megoldásokkal lehetséges.[11]

```
<button(click)="onSave()">Save</button>
<app-hero-detail(deleteRequest)="deleteHero()"></app-hero-detail>
<div (myClick)="clicked=$event" clickable>click me</div>
```

A Vue.js is biztosítja számunkra mind az egyirányú, mind a kétirányú data binding lehetőségét. Az egyirányú data binding legegyszerűbb használata a dupla kapcsos zárójel közé beírt változónévvel alkalmazható, azonban vannak erre kifejlesztett direktívák is, mint például a v-bind vagy a v-on. A kétirányú data binding pedig a v-model direktíva használatával elérhető.[10]

```
<!-- full syntax -->
<button v-bind:disabled="someDynamicCondition">Button</button>
<!-- shorthand -->
<button :disabled="someDynamicCondition">Button</button>
```

4.6 Felépítési filozófia

Alapvetően mindhárom keretrendszer komponens alapú megközelítést alkalmaz. Az alkalmazást a funkciók szerint vagy valamilyen más megközelítés szerint komponensekre bontva kell a fejlesztést elvégezni. Ennek a megközelítésnek számos előnye van. Lehetővé teszi a fejlesztőnek, hogy független, újra felhasználható komponenseket készítsenek, így elkerülhetőek a kód duplikációk és átláthatóbbá válik az alkalmazásunk. További előnye a komponens alapú felépítésnek, hogy nem muszáj a fejlesztőnek megírnia az egész komponenst, előfordulhat, hogy az interneten már létezik ahhoz nagyon hasonló működésű komponens, mint amire nekünk szükségünk van, és így ezeket fel tudjuk használni, ezzel is rengeteg időt megtakarítva.

4.6.1 React.js alkalmazások felépítése

React.js-ben ezek a komponenseket JavaScript metódusokként lehet elképzelni, vannak bemeneti paramétereik és visszaadnak egy React objektumot, ami megmondja minek kell megjelennie a felületen. Komponenseket kétféle módon tudunk létrehozni, lehet metódusok vagy osztályok készítésével.

```
class exampleComponent extends React.Component {
    render() {
       return <p>Ez egy példa:, {this.props.name}</p>;
    }
}
```

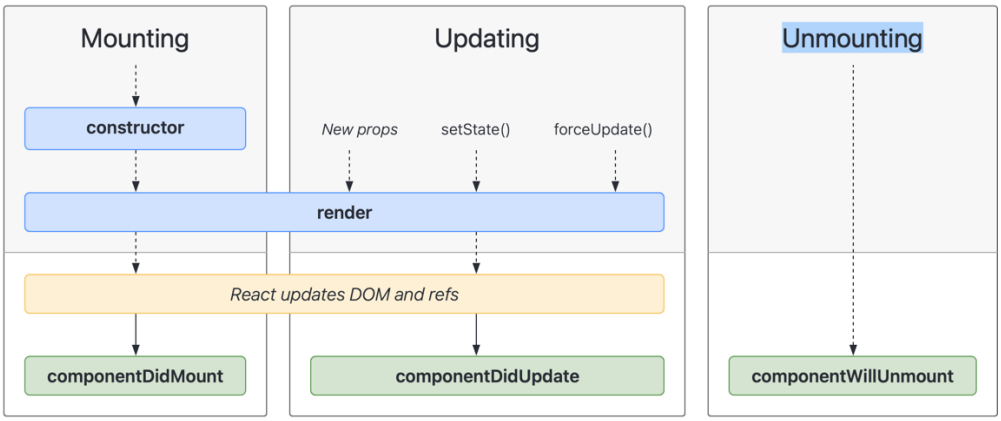
Van egy fontos szabálya a komponenseknek, ami nem más, mint az, hogy a komponens property-jeit (későbbiekben prop) nem módosíthatja. Ezt úgy nevezzük, hogy tiszta módon futnak a komponensek.

```
function modifyOrderAmount(Order, newValue) {
  Order.Amount = newValue;
}

function AddToOrderAmount(Order, toAdd) {
  Order.Amount += toAdd;
}
```

További fontos tudnivaló még a komponensekkel kapcsolatban az nem más, mint a komponensek életciklusa. Egy komponensnek három állapota lehet, egyszer lehet létrehozás állapotban, ebbe akkor kerülnek a komponensek, amikor létrehozzuk őket. Egy másik állapot a frissítés, ebbe akkor kerülhet egy komponens, amikor megváltozik az állapota, vagy a propjai. Illetve lehet még leszerelés állapotban, amikor eltávolítjuk a

DOM-ból. Ezekben az életciklusokban találhatóak metódusok is, amik lefutnak az életciklus során egy meghatározott sorrendben. Lehetőségünk van ezeket a metódusokat felülírni, és így a saját kódjainkat futtatni. Az életciklusokban található metódusok és ezeknek a futási sorrendje az alábbi ábrán látható.



4. ábra React komponensek életciklusai [7]

Ahhoz, hogy egy komponens megjelenjen a kimeneten, vagyis a felhasználói felületen ki kell renderelni. Ezt a ReactDOM.render() függvény meghívásával tehetünk meg. Van lehetőségünk meghatározni, hogy mely komponensek milyen feltételek teljesülése esetén renderelődjön ki. Ezt egyszerűen eldönthetjük egy változó alapján, egy if-else segítségével, vagy már magában egy komponensben is. Olyan is előfordulhat, hogy egy komponenst többször szeretnék megjeleníteni a képernyőt, erre szintén lehetőségünk van. Azonban ezeket azonosítanunk kell valamivel, hogy meg tudjuk találni a komponenseket, erre kulcsokat használunk. A kulcs általában Id-ként van hivatkozva, és ez egy szöveg vagy szám.

4.6.2 Angular.js alkalmazások felépítése

Angular.js esetében az alkalmazásunk egy komponenseket tartalmazó faként lehet leírni. A komponensek a legalapvetőbb építőelemei a felhasználói felületünknek. A komponensek mindig össze vannak kapcsolva egy template-el. Egy template egy eleméhez csakis kizárólag egy komponens köthető. Ahhoz, hogy egy komponenst más komponensekben vagy alkalmazásokban is tudjuk használni egy NgModule-hoz kell kapcsolni, ezt úgy tehetjük meg, hogy az NgModule metaadatai között megadjuk a komponensünket.

Angularban egy komponens definiálása eltérő attól, amit a React-nél láttunk. Itt dekorátorok segítségével kell definiálnunk őket. A dekorátorokban kell megadni a komponens különböző információit.[11]

- selector: komponenst azonosító név
- styleURL: komponenshez tartozó CSS fájlok útvonala
- templateURL: útvonal a komponens sablonjához

```
@Component({
  selector: 'app-my-example',
  templateUrl: './my-example.component.html',
  styleURL: './my-example.component.css'
})
export class MyExampleComponent { }
```

Ahogy a React.js esetében láthattuk itt is van a komponenseknek életciklusa. Ezek segítségével a szakaszokban futtathatjuk saját kódjainkat, véghez vihetjük folyamatinkat. Miután az alkalmazásunk példányosít egy komponenst a konstruktorral, az Angular meghívja az adott életciklushoz tartozó hook metódusokat a következő sorrendben.

Hook metódus	Leírás	Meghívás időpontja
ngOnChanges	Akkor hívódik meg, amikor beállítjuk vagy visszaállítjuk az adatokhoz kötött tulajdonságokat. Viszonylag sokszor hívódik, így teljesítmény igényes feladatok implementálása nem ajánlott.	Az ngOnInit előtt hívódik és akkor ha megváltoznak az adathoz kötött tulajdonságok.
ngOnInit	Inicializálja a komponenst és beállítja a bejövő adatokat.	Egyszer hívódik meg az első ngOnChanges metódus után.
ngDoCheck	Felismeri azokat a változásokat, amiket az Angular magától nem tud.	Minden ngOnChanges után meghívódik közvetlenül és az első futáskor az ngOnInit után is.

ngAfterContentInit	Akkor válaszol, amikor egy külső tartalom megjelenik a nézetben.	Egyszer hívódik meg az ngDoCheck után.
ngAfterContentChecked	Akkor válaszol, amikor az Angular ellenőrzi a kivetített tartalmat a komponensben.	Az ngAfterContentInit után és minden ngDoCheck után.
ngAfterViewInit	Akkor tér vissza, amikor a komponens nézetét inicializálta.	Csakis egyszer az ngAfterContentChecked után.
ngAfterViewChecked	Akkor tér vissza, ha az Angular ellenőrizte a komponens nézeteit.	Az ngAfterViewInit után és minden alhívás ngAfterContentChecked után.
ngOnDestroy	Mielőtt az Angular megsemmisíti a komponenst meghívódik és megtisztítja a komponenst, leiratkozik eseményekről stb. ezzel is megakadályozva a memória szivárgást.	Közvetlenül mielőtt egy komponenst megsemmisít az Angular.

4.6.3 Vue.js alkalmazások építő elemi

Vue.js keretrendszer alapvető koncepciója szintén a komponensekből történő építkezés. Alapvetően kétféle megközelítést szoktunk alkalmazni. Az egyik, hogy minden komponens egy fájlban van definiálva, ezt főleg kis és közepes méretű projekteknél lehet alkalmazni. Nagyobb projekteknél azonban ez nem járható út, több okból kifolyólag sem. Az egyik probléma a globális változókkal van. Amennyiben minden komponensünk egy fájlban található, minden változónak különböző nevet kell adnunk. Ezekre a problémákra megoldást nyújt az a megközelítési mód, amikor minden komponenst egy külön .vue kiterjesztésű fájlban definiálunk.

```
<script>
module.exports = { data() {
    return { greeting: "Hello" };}};
</script>
```

Ahogy a fenti példában láthatjuk, itt is megjelenik a data tulajdonság, azonban ebben az esetben ez egy metódus, ami egy objektumot ad vissza. Ez azért így működik, mivel, ha egy komponenst többször használunk fel, akkor minden komponensnek külön adatokkal kell rendelkeznie, nem globális adatokat szeretnénk használni.

A komponensek használatához azonban először be kell regisztrálnunk őket, hogy a vue ismerje és tudja használni őket. Ennek kétféle módja van. Lehet lokálisan vagy globálisan regisztrálni egy komponenst. A globális regisztráció hátránya, hogy ha már egy komponenst nem használunk akkor is benne van az alkalmazásban és így le kell töltenie a felhasználónak feleslegesen, ilyenkor érdemes a lokális regisztrációt alkalmazni. Azonban a lokálisan regisztrált komponensek nem elérhetőek alkomponensekben, így, ha használni akarjuk őket, oda is külön be kell regisztrálni.[10]

Globális Regisztráció	Lokális Regisztráció

`const app = Vue.createApp({})`
 	
`app.component('component-a', {`
`  /* ... */`
`})`
	`const app = Vue.createApp({`
	`  components: {`
	`  'component-a': ComponentA,`
	`  'component-b': ComponentB`
	`  }`
	`})`

Lehetőségünk van a komponenseket egymásba ágyazni, azaz szülő gyerek kapcsolatba állítani őket. Ilyen esetben sokszor kell adatokat átvinni egyik komponensből a másikba. Erre van egy props nevezetű tulajdonság. Ez a szülő -> gyerek irányba működik, ha a másik irányba szeretnénk adatokat küldeni, vagy valami változásról értesíteni a szülő komponenst azt esemény objektumokon keresztül tehetjük meg. Ennek a kétirányú kommunikáció megvalósításának megkönnyítésére hozták léte a v-model direktívát. A direktívákról a következő fejezetben lesz szó.

4.7 Direktívák

A direktívák lényegében arra szolgálnak, hogy egy speciális jelöléssel (egyedi attribútummal) megmondja a könyvtárnak, hogy csináljon valami a DOM elemekkel.

Lényegében egy egyedi HTML attribútum. Például, ha van egy listánk és minden elemét meg akarjuk jeleníteni, akkor erre jó megoldást nyújthat egy direktíva.

4.7.1 Direktívák a React.js-ben

A React.js-ben alapból nem találhatóak meg a direktívák. Ez nem is meglepő, mivel ez a másik kettő összehasonlításban szereplő technológiával ellentétben csak egy könyvtár nem egy egész keretrendszer, így nyilván kevesebb funkcióval is rendelkezik. Azonban ezt a hiányosságot egyszerűen lehet orvosolni, egy másik könyvtár alkalmazásával. Erre a react-directive nevezetű könyvtárat szoktuk alkalmazni. Ezt az npm segítségével egyszerűen tudjuk telepíteni és használni az alkalmazásunkban.

A react-directive könyvtárat a vue.js-ben található direktívák inspirálták. Ezáltal a használata és a működése nagyon hasonlít a vue.js-ben lévő direktívákéhoz, amit a később látni is fogunk, amikor megnézzük a vue.js-ben lévő direktívákat is. Ebben a könyvtárban kétféle direktíva található, az egyik a feltételek vizsgálatára a másik pedig a komponensek listázására használható.

```
import React from 'react';
import ReactDirective from 'react-directive';
const MyComponent = props => {
    return (
        <ReactDirective>
            {/* Az itt lévő komponenseknél lehet használni a  data-react-
if és a data-react-for direktívákat. */}
        </ReactDirective>
    );
}
export default MyComponent;
```

4.7.2 Angular.js-ben alkalmazott direktívák

Angular esetében a React-el ellentétben már alapból jelen vannak a direktívák, és eltérő a használatuknak mind a módja mind pedig a szintaktikája. Első lépésként meg kell értenünk, hogyan fordítja le az Angular a HTML-t és ezzel együtt a direktívákat. Ez három lépésben történik.[10]

1. Bejárja a DOM elemeket és azonosítja a direktívákat. Ha a fordító talál egy elemet, amihez van direktíva akkor a direktívát beleteszi egy listába, ebben a listában azok a direktívák vannak, amik az elemhez tartoznak.

2. Miután minden direktívát azonosított, rendezik őket a prioritásuk szerint és minden direktíva compile metódusa lefut, ezek módosítják a DOM elemet.
3. Összeköti a sablonnal, beregisztrálja a listener-eket. Létrejön a kapcsolat a scope és a DOM elem között.

Angularban alapvetően három típusra tudjuk bontani a direktívákat.

- Komponens direktíva: A sablon, azaz a HTML elemek számára. Ez a leggyakoribb.
- Attribútum direktíva: Megváltoztathatják egy elem, komponens vagy egy másik direktíva viselkedését, megjelenését.
- Strukturális direktíva: A DOM-ot módosítja, eltávolít vagy hozzáad elemeket.

```
<!-- toggle the "special" class on/off with a property -->
<div [ngClass]="isSpecial ? 'special' : ''">This div is special</div>
```

4.7.3 Vue.js direktívák

A Vue.js keretrendszerben az Angularhoz hasonlóan szintén alapból megtalálhatóak a direktívák, de szintaktikájuk és működésük eltérő, ebben a React-nél bemutatott kiegészítő könyvtárhoz hasonlóak. Azonban lényegesen több féle érhető el belőle. Vue.js-ben a direktívákat a következőképpen kell megadni a DOM elemeknek. Elején egy v- prefixet kell használnunk, ezzel jelezzük, hogy egy direktívát akarunk használni. Ezután jöhet a direktíva neve. Előfordulhat, hogy egy direktívának vannak argumentumai, ezeket egy kettőspont után lehet megadni. Van még lehetőségünk módosítók megadására is, ezekkel speciális viselkedést érhetünk el a DOM elemeknél, ezeket egy pont megadása után kell meghatározni. Vannak előre definiált direktívák, de lehetőségünk van nekünk is létrehozni újakat. Vannak eseményekre vonatkozó direktívák például egy kattintás eseményre egy direktíva a következőképpen néz ki.[9]

```
<button v-on:click="Submit">Submit form</button>
```

Továbbá vannak úgynevezett feltételes direktívák is, amikkel dinamikusan eldönthetjük, hogy egy elemet meg szeretnénk-e jeleníteni. Itt három direktívát kell mindenképpen megemlíteni a v-if, a v-else és a v-show direktívákat. Az első kettő szerintem nem igényel magyarázatott, a már megszokott módon működnek. A v-show

abban különbözik a v-if-től, hogy ebben az esetben legenerálódik az elem, csak az inline tulajdonság módosításával jelenítjük meg vagy tűntetjük el az elemet.

```
<div id="app">
     <button v-on:click="change">Change button</button>
     <p v-if="condition">Condition true</p>
     <p v-else>Condition false</p>
     <p v-show="condition2">Condition2 true</p>
</div>
```

Vannak még direktívák egy DOM elem egy attribútumhoz való kapcsolásához, ez a v-bind direktíva, illetve van még a v-for direktíva, ami a data-react-for direktívának feleltethető meg.

4.8 Események kezelése

Egy webes vagy bármilyen alkalmazás fejlesztésekor elengedhetetlen az események kezelése. Ezek az események jöhetnek a felhasználói felületről, azaz a felhasználótól vagy jöhetnek magából a programból is, amikor például lejár egy időzítő és az triggerel egy eseményt. Ezeknek az eseményeknek a kezelése mind a három keretrendszerben meg van oldva, de különböző módszerekkel. Ebben a fejezetben ezeket a megoldásokat hasonlítom össze.

4.8.1 React.js események

Reactben az események kezelése nagyon hasonló a DOM-ban megszokotthoz, csak kisebb szintaktikai eltérések vannak. Egyik ilyen, hogy JSX használata esetén az eseménykezelőnek egy string helyett magát a függvényt adjuk át. Az eseménykezelők még kapnak alapból úgynevezett szintetikus eseményeket is. A szintetikus események hasonló tulajdonságokkal rendelkeznek, mint a böngészőben szerepló natív események, de ezek nem függenek attól, hogy milyen böngészőből van futtatva az alkalmazás. Ezek a szintetikus események a Reactben a W3C szabvány alapján van megvalósítva, így nem kell kompatibilitási problémáktól tartanunk. Azonban, ha mégis szükségünk lenne egy natív eseményre, akkor azt is el tudjuk érni.

Ha egy DOM elemhez eseménykezelőt szeretnénk hozzárendelni azt általában már az elem létrehozásánál hozzá szoktuk adni, azonban lehetőség van utólag is hozzáadni az addEventListener() metódus meghívásával.[6]

```
<button onClick={(e) => this.showMore(id, e)}>További infó</button>

ReactDOM.findDOMNode(this).addEventListener('showMoreButton', this.handleClick);
```

4.8.2 Angular.js események kezelése

Angular.js-ben szintén elég hasonlóan működik az események kezelése, mint a DOM-ban. A HTML elemeknél kell megadnunk, hogy egy esemény bekövetkezte esetén, milyen metódus hívódjon meg. Szintaktikailag azonban eltérőek, erre egy példa alább látható.

```
<input [value]="currentItem.name"
       (input)="currentItem.name=getValue($event.target)">
```

Ha ezt a példát nézzük itt átadjuk az eseményt kezelő metódusnak az event objektumot, ami tartalmazz egy csomó információt magáról az eseményről, például, hogy melyik gomb lett lenyomva. A példánknál maradva ebben az esetben a következők történnek.[11]

1. Az input HTML elem input eseményéhez kötjük a getValue() metódust.
2. Amikor a felhasználó megváltoztatja az input elembe írt adatot az esemény meghívódik.
3. Végrehajtódik az utasítás.
4. Lekérdezzük a megváltozott nevet a getValue() metódussal.

4.8.3 Vue.js események kezelése

Vue.js esetében is hasonló módon történik az események kezelése, mint az előző két technológiánál, azonban szintaktikailag itt is eltér ez előzőkétől. Itt is a HTML elemeinknél kell megadni az eseményt és a hozzá kapcsolódó funkcionalitást, de ezt ebben az esetben a v-on direktíva segítségével tudjuk megvalósítani. A direktívának lehet megadni egyszerűbb feladatokat is vagy bonyolultabb folyamatok esetében egy metódust.

```
<div id="example-2">
  <!-- `greet` a metódus neve-->
  <button v-on:click="greet">Greet</button>
</div>
```

Sokszor előfordul az eseményeknél, hogy meg kell hívjuk az esemény interfész valamely metódusát, például az event.preventDefault() metódust. Ezt egyszerűen megtehetjük a handler metódusban is, de van erre lehetőségünk a direktíváknál is, ezzel is átláthatóbbá téve a kódunkat. A következő esemény módosító kifejezések érhetőek el a v-on direktíváknál[9]:

- stop
- prevent
- capture
- self
- once
- passive

```
<!-- the submit event will no longer reload the page -->
<form v-on:submit.prevent="onSubmit"></form>
```

4.9 Routing

A single-page alkalmazásoknál nem az egész oldalt töltjük be a szerverről, hanem a meglévő oldalt írjuk felül az új adatokkal. Ennek az oka, hogy gyorsabban történjen a váltás. Ezt a nézetek közötti váltást lehetővé tévő megoldást szoktuk routingnak nevezni. Routingra minden technológia esetében szükség van, ebben a fejezetben bemutatom milyen megoldások vannak rá az általam vizsgált három technológiában.

4.9.1 React Router

React.js-ben alapból nincs megoldás a routingra, ezt egy külső könyvtár segítségével tudjuk megvalósítani, ami a React Router. Ennek telepítése nagyon egyszerűen az npm segítségével megoldható. Jelen esetben mi egy web alkalamzást szeretnénk készíteni így a react-router-dom-ot kell telepítenünk és importálnunk.[16]

Három fő típusa van a komponenseknek a React Routerben, amit feltétlenül meg kell említenünk.

- routers: Ez a fő routing komponens, minden alkalmazásba mely megvalósítja a routingot kell, hogy szerepeljen. Kétféle router található a könyvtárban a <BrowserRouter> és a <HashRouter>. A különbség a kettő között az URL tárolásának és a webkiszolgálóval történő kommunikációjának módjában van.

    - BrowserRouter: A megszokott URL útvonalakat használja, ezért megköveteli, hogy a szerver jól legyen konfigurálva. Pontosabban a szervernek mindig ugyan azt az oldalt kell kiszolgálnia, minden olyan címen, amit a kliens oldalon a React Routerrel kezelünk.
    - HashRouter: Az aktuális helyet az URL hash részében tárolja. Mivel a hash-t nem küldjük el a szervernek az nem igényel konfigurációt.

- route matcher: Kétféle létezik belőle, ezek a <Route> és a <Switch>. A switch végigmegy a route elemeken és megkeresi azt amelyik a legjobban passzol az URL-hez és azt jeleníti meg a többit nem. Ezért fontos, hogy a speciálisabb címeket tartalmazó route-okat előrébb tegyük a kevésbé specifikusakhoz képest. Ha nincs találat semmit nem jelenít meg.
- navigation: Ide tartozik a <Link> , <NavLink>, <Redirect>. A Link egy <a> HTML elemet generál a háttérben a NavLink pedig a link speciális típusa, ami aktívra tudja állítani magát, ha megfelelő helyen vagyunk. A Redirect pedig átnavigál minket oda, amit megadtunk neki attribútumként.

```
import { BrowserRouter } from "react-router-dom";
ReactDOM.render(
  <BrowserRouter>
    <App />
  </BrowserRouter>,
  el
);
const App = () => (
  <div>
    <nav>
      <Link to="/dashboard">Dashboard</Link>
    </nav>
    <div>
      <Route path="/dashboard" component={Dashboard} />
    </div>
  </div>
);
```

A React Router megközelítésben is különbözik a Vue.js és az Angular.js által használt routingtól, míg azok statikus routing-ot alkalmaznak a React Router dinamukust. A statikus routing azt jelenti, hogy az útvonalakat még az app inicializálása közben definiálni kell még mielőtt bármit megjelenítenénk. A dinamikus routing esetében pedig

a routing a megjelenítés közben történik nem pedig a futó alkalmazáson kívüli konfigurációban.

4.9.2 Angular.js routing

Angularban a React-el ellentétben van beépített megoldás a routingra, az Angular Router. Ennek a használatához első lépésként be kell importálni a Routes-t és a RouterModule-t a @angular/router könyvtárból. A Router egy szolgáltatás, ami biztosítja a nézetek közötti váltást és a linkek használatát. A Route egy konfigurációs objektum, ami egyet egy útvonalat tartalmaz. A Routes egy Route-okat tartalmazó tömb, ami egy útvonal konfigurációt tartalmaz magának a Router szolgáltatásnak. Egy alkalmazásnak egy Router példánya van, amikor a böngészőben az URL megváltozik a router megkeresi a megfelelő útvonalat és ez alapján a megjelenítendő komponenseket. Egy router konfigurációja a következőképpen néz ki[11].

```
const appRoutes: Routes = [
  { path: 'crisis-center', component: CrisisListComponent },
  { path: 'hero/:id',      component: HeroDetailComponent },
  {
    path: 'heroes',
    component: HeroListComponent,
    data: { title: 'Heroes List' }
  },
  { path: '',
    redirectTo: '/heroes',
    pathMatch: 'full'
  },
  { path: '**', component: PageNotFoundComponent }
];
```

A példában használt paraméterek magyarázata:

- path: útvonal, megadható egy stringként, a ** paraméter a minden további lehetőséget jelenti
- component: a megjeleníteni kívánt komponens
- data: az útvonalhoz tartozó adatokat adhatjuk itt meg
- redirectTo: továbbküldi az alkalmazást a megadott címre
- pathMatch: azt mondja meg, hogy az URL-t hogyan illessze az útvonalhoz, lehet full vagy prefix

További funkciók, melyek hasznosak lehetne az útvonalak definiálása során:

- Jelenleg bárki bármelyik oldalra navigálhat az alkalmazásban, de előfordulhat, hogy nem szabad mindenkinek elérnie egy oldalt, erre a guard-ok alkalmazása jelent megoldást.
- Nagyobb alkalmazások esetén sokáig eltarthat, mire betölt mindent a böngésző, ezért érdemes aszinkron betöltést használni, amit az angular lusta betöltésnek hív. Az útvonalaknál meg lehet adni, hogy az adott útvonal az egy lusta útvonal-e.
- Előfordulhat, hogy egy komponenshez további gyerek komponensek szükségesek, ekkor ezeknek az útvonalát a szülő komponensben egy újabb routes tömböt kell megadni, amiben lehet definiálni a gyerek komponensek útvonalát.

4.9.3 Vue.js routing

Vue.js-ben is található hivatalos támogatott megoldás a routingre, ami a vue-router könyvtár. Ennek működése hasonló az Angular.js-ben található routinggal. Első lépésként telepítenünk kell a könyvtárat, amit megtehetünk a parancssorból az npm segítségével. Ezután importálni kell a VueRouter-t.

Ahogy az Angularban úgy itt a Vue.js-ben is a route, a routes, és a router objektumokkal dolgozunk elsősorban. Első lépésként definiáljuk a route komponenseket. Ez a komponenst template-jét tartalmazza. Ezt követően létrehozzuk a routes tömböt, amiben megadjuk az útvonalat és a komponenseknél az útvonalhoz tartozó route komponenst, amit az első lépésnél hoztunk létre. Ha ezzel is megvagyunk, akkor létrehozunk egy példányt a router objektumból és átadjuk neki a routes tömböt. Végül pedig létrehozzuk az alkalmazást a new Vue() metódussal, ennek paraméterként átadjuk a router objektumot, így már tudjuk használni a router példányt a this.$router kulcsszóval, ami azért hasznos mert így nem szükséges minden komponensbe importálni a router-t.

```
const alma = { template: '<div>alma</div>' }
const barack = { template: '<div>barack</div>' }
const routes = [
  { path: '/alma, component: alma },
  { path: '/barack, component: barack }
]
const router = new VueRouter({
  routes: routes
})
```

```
const app = new Vue({
  router
}).$mount('#app')
```

A fenti példa a legegyszerűbb megvalósítása a routingnak, ennél sokkal bonyolultabb megoldások vannak, melyek sok hasznos funkciót kihasználnak, amivel a vue rendelkezik, melyek közül a legfontosabbak a következők:

- Guardok: Angolur.js-hoz hasonlóan itt is tudunk használni guardokat. Többféle guardot tudunk definiálni, attól függően, hogy mikor szeretnénk, hogy lefusson. Például vannak beforeRouteLeave guardok, amik elsőként futnak, a deaktivált komponensben, aztán lehet beforeRouteEnter, ami pedig az új komponensben fut, mielőtt belépünk oda. Ezeken kívül van még számos más lehetőség is.
- Nested routes: Ahogy Angularban láttuk itt is van lehetőség beágyazott vagy gyerek útvonalak felvételére.
- Lusta betöltés: Angularhoz hasonlóan az teljesítmény javítására szolgál, csak akkor töltjük be a komponenseket, amikor az útvonalra megyünk.
- Programból történő navigálás: Deklaratív módszer volt a példánkban, amikor konkrétan meg volt adva, hogy melyik útvonalra navigáljunk, de lehetőségünk van ezt a programból megadni. Ehhez a router.push() metódust kell alkalmazni.

4.10 Tesztelési lehetőségek

Minden informatikai rendszer fejlesztésének elengedhetetlen része a tesztelés, mivel így tudjuk ellenőrizni, hogy valóban az elvártak szerint működik-e az alkalmazásunk. Erre a különböző keretrendszerek esetében eltérő megoldások vannak. Ebben a fejezetben ezekre fogok összpontosítani és megnézem a különbségeket és hasonlóságokat a megoldások között.

React esetében a tesztelés hasonlóan történik, mint más JavaScript kódok esetén. Leginkább kétféle módon szoktuk tesztelni az alkalmazásokat. Az egyik, hogy a komponenseket megjelenítjük egy fa struktúrába és a kimenetét teszteljük, hogy az az elvárt legyen. A másik, hogy az egész alkalmazást futtatjuk egy környezetben. A React

alapvetően kétféle eszközt ajánl számunkra a teszteléshez, az egyik a Jest a másik pedig a React Testing Library.

Egy teszt a következőképpen épül fel:

- beforeEach(): Felállítunk egy DOM elemet, amit tesztelni szeretnénk, vagy a hozzá kapcsolódó elemeket.
- afterEach(): A teszt végezetével feltakarít.
- act(): Ebben rendereljük ki a kívánt komponenst, biztosítja, hogy minden változtatást megtörténjen a komponensen mielőtt megtörténik az assert.
- expect(): Elvárt és a kapott eredmény összehasonlítása.

```
import Hello from "./hello";
let container = null;
beforeEach(() => {
  container = document.createElement("div");
  document.body.appendChild(container);
});

afterEach(() => {
  unmountComponentAtNode(container);
  container.remove();
  container = null;
});

it("renders with or without a name", () => {
  act(()     =>     {     render(<Hello     />,     container);          });
expect(container.textContent).toBe("Hey, stranger");
  act(() => {
    render(<Hello name="Jenny" />, container);
  });
  expect(container.textContent).toBe("Hello, Jenny!");
 });
```

Mint a legtöbb esetben az Angularban szintén megtaláljuk a tesztelésre szolgáló funkciókat. Egyszerűen az Angular CLI segítségével fel tudjuk telepíteni a szükséges fájlokat. Maga a tesztelés itt is unit tesztek segítségével történik.

A leggyakrabban a TestBed API segítségével történik a tesztelés Angular alkalmazásoknál. Ez egy környezetet biztosít számunkra, metódusokkal, amikkel komponenseket és szolgáltatásokat tudunk készíteni a unit tesztünkhez.

Maga a unit teszt felépítése hasonló, mint a Reactnél, ugyan úgy létrehozzuk vagy futtatjuk a komponenseket, szolgáltatásokat és a végén ellenőrizzük a kapott eredményt.

Ugyan úgy megtalálható a beforeEach() metódus, mint Reactnél, egy példa az alább látható, ahol a TestBed használatával tesztelünk egy szolgáltatást.

```
beforeEach(() => {
    TestBed.configureTestingModule({ providers: [ValueService] });
    service = TestBed.inject(ValueService);
});
```

A Vue.js alapvetően három típusú tesztelést ajánl az alkalmazásokhoz, ez a három a következő:

- Unit tesztek: különálló kódok tesztelésére jó.
- Komponens tesztek: komponensek tesztelésére.
- End-to-End tesztek: Teljes alkalmazás tesztelése.

Azonban a teszteléshez csak külső eszközöket ajánl nekünk, mint például a Vue Testing Library vagy a Reactnél már megismert Jest. A tesztelés menete itt is ahhoz hasonló, mint amit a Reactnél vagy az Angularnál láttunk. Leggyakrabban létrehozunk egy komponenst és annak a kimenetét ellenőrizzük.

Összességében elmondhatjuk, hogy mind a három keretrendszer hasonló megközelítést alkalmaz az alkalmazások tesztelésénél. Főként a React.js és a Vue.js esetén nagy a hasonlóság, hiszen akár a Jest tesztekkel mindkét keretrendszer alkalmazásait tudjuk tesztelni. Angularban, ahogy láthattuk kicsit eltérőbb módon történik a tesztelés, de a filozófia itt is hasonló.

5 Node.js technológia

Ebben a fejezetben az általam választott szerver oldali technológiát a Node.js-t mutatom be részletesebben. Szó lesz a Node.js-en belül elérhető keretrendszerekről, ezeknek sajátosságairól, és bemutatom az általam használni kívánt keretrendszert az ExpressJs-t.

5.1 Node.js

A Node.js egy nyílt forráskódú, platformokon átívelő JavaScript alapú környezet. Ez elsőre meglepő lehet, mivel JavaScriptet többnyire a kliens oldalon szoktunk használni, ami a böngészőben fut, ebben az esetben azonban a szerveren fogjuk futtatni a kódunkat. Ahhoz, hogy ez működjön biztosítanunk kell neki egy JavaScript motort, például ez a Google Chrome böngésző esetében lehet a beépített V8-as motor. Ez a motor fordítja le a kódunkat gépi kódra.

A Node.js alkalmazások egy szálon futnak, több kérés esetén sincs szükség új szálak létrehozására, mivel a standard könyvtárakban olyan folyamatok vannak, melyek megakadályozzák a blokkolódást, így ez csak ritkán fordulhat elő. Például amennyiben egy hálózatról szeretnénk adatokat lekérni, akkor ameddig megérkeznek az adatok nem fogja blokkolni a folyamatot, hanem majd visszatér rá, miután megérkeztek a kívánt adatok. Ez a tulajdonsága miatt sok kérést ki tudunk szolgálni egyetlen szerverről is. További előnye a Node.js technológiának, hogy nem kell új nyelvet megtanulnia például egy frontend fejlesztőnek, hanem tudja kamatoztatni a JavaScript tudását[17].

A standard könyvtárak mellett lehetőségünk van további könyvtárakat is használni a fejlesztés megkönnyítésére. Szinte végtelen az elérhető könyvtárak száma, amelyekkel sokszor le lehet rövidíteni a fejlesztési időt. Mivel a Node.js egy alacsony szintű architektúra, annak érdekében, hogy a fejlesztés egyszerűbb legyen vannak hozzá különböző keretrendszerek, ilyen például a Nest.js vagy az Express.js.

5.1.1 Alap modulok

A Node.js-be alapból is találhatóak bizonyos modulok, ezeket az alábbi táblázatban gyűjtöttem össze.[17]

Név	Funkcionalitás
http	Szerver indítása, kérések küldése.
https	Szerver indítása SSL-el.
fs	Fájlok menedzselése.
path	Mappák és útvonalak menedzselése.
os	Információt biztosít a számítógép operációs rendszeréről.

Az alapmodulok használata rendkívül egyszerű, csak be kell őket importálni az alkalmazásunkban és már használhatjuk is őket. Készíthetünk még saját modulokat is és ezekből exportálhatunk saját API-kat. Ezt a module és az exports globális változókkal tudjuk megtenni.

```
var http=require('http');
var fs=require('fs');
http.createServer(function (req,res){
 fs.readFile('test.html', function(err,data){
    res.writeHead(200,{'Content-Type': 'text/html'});
   res.write(data);
    return res.end();
 });
}).listen(8080);
```

5. ábra Alap modulok használata Node.js alkalmazásban

5.1.2 Külső modulok használata

A Node.js-ben megtalálható alap modulokon kívül lehetőségünk van használni külsős modulokat is, például a felhasználói input ellenőrzésére vagy egy kérés feldolgozására. Ezek a külsős modulok vagy csomagok az npm repositryba találhatóak és onnan tudjuk telepíteni az npm segítségével. A telepítést a konzolból tudjuk elvégezni a következő paranccsal.

```
npm install nodemon –save-dev
```

Az első része az npm maga a package manager, a második az install parancs ezután következik a package neve, amit fel szeretnénk telepíteni. Majd a végén meg kell adni neki, hogy ezt a package-t hogyan szeretnénk használni. A –save-dev azt jelenti, hogy csak a fejlesztéshez használjuk a packaget, a szerveren nem fog futni élesbe. Ha csak sima –save -et adunk meg akkor azt használni fogjuk a szerver futása alatt. Végük van még egy lehetőségünk, amikor csak egy sima -g -t adunk meg akkor az azt jelenti, hogy globálisan telepíti az npm, ilyenkor bármelyik alkalmazásba lehet használni az adott packaget.

5.1.3 A Node.js alkalmazások életciklusa

Miután létrehoztunk egy alap alkalmazást például example.js néven, akkor ezt a node paranccsal tudjuk futtatni. Ekkor első lépésként elindul egy szkript. Ez a szkript lefordítja a kódunkat, végig megy rajta és közben regisztrálja a változókat és a metódusokat. Ez után pedig egy úgynevezett Event Loopba kerül. Ezáltal a programunk addig fog futni, ameddig van regisztrált esemény. Erre azért van szükség, mivel a Node.js szervernek akár több ezer kérést is ki kell szolgálnia, miközben csak egyetlen szálon fut. Az Event loop felelős a callback metódusok futtatásáért és az események kezeléséért. Ha egy nagyobb feladat érkezik például egy fájlból kell kiolvasnunk valamit, akkor az a Worker poolba kerül. A Worker poolba a folyamatok már több szálon futnak és ha végeztek akkor felhívják a callback metódusokat, amikért ugye az Event loop a felelős. Ezeknek a kéréseknek a kiszolgálása nagyon gyorsan történik és a kialakításnak köszönhetően nem blokkolja egy kérés a többit. Ha a http.createServer() metódust tekintjük, akkor ez egy esemény és ezáltal folyamatosan futni fog szerverünk.

Ha meg akarjuk szakítania futást, arra is van lehetőségünk a process.exit() metódus meghívásával.

5.1.4 Az Event loop működése

Az event loop minden Node.js alkalmazás szíve. Ezt egy folyamatosan futó folyamatként tudjuk elképzelni, ami egészen addig fut, ameddig van hozzá kapcsolt esemény, azaz event. Az event loop felelős elsősorban a callback függvények és az események menedzseléséért. A loopon belül van egy sorrend, ami alapján végrehajtja a különböző függvényeket. A sorrend a következő.[17]

1. Időzítők és a hozzájuk kapcsolódó callback metódusok. Ezek alatt elsősorban a setTimeout() és a setInterval() metódusokat értem.
2. Folyamatban lévő callback metódusok.
3. Poll fejezet. Itt új, futtatható callback metódusokat keresünk. Ha találunk olyan és nem tudjuk még egyből futtatni akkor a 2-es pontban lévő éppen futó callback-ekhez csatoljuk, illetve, ha időzítőket találunk, akkor az 1-es pontban lévőkhöz.
4. Check. Ebben a részben a setImmediate() metódusok. Hasonló a setTimeout() metódushoz. A lényeges különbség, hogy a setTimeout esetében egy konkrét időt adunk meg, míg a setImmediate esetén a poll után fog futni.
5. Close callbacks

5.1.5 Streamek

A stream egy absztrakt interfész, amivel a Node.js adatfolyamát tudjuk kezelni. A stream modulban található egy API, melynek segítségével megvalósíthatjuk az interfészt. A Node.js számos stream objektumot biztosít. Például egy HTTP-kiszolgálóhoz intézett kérés és a process.stdout egyaránt egy stream példány. Egy stream lehet olvasható, írható vagy mindkettő. A stream modult legtöbbször új típusú streamek készítéséhez használjuk egyéb esetben nem szükséges a stream modult alkalmazni.

Streamek használata azért fontos, mert például, ha egy fájlt kérdez le egy kliens és a fájlnak a mérete elég nagy, akkor az nagyon nagy memóriaigénnyel járhat, ugyanis, ekkor az egész fájlt betöltjük. Ekkor érdemes stream-eket használni. Streamek esetében a pipe kezeli az adatokat és így egyből a klienshez lesz küldve a fájl, így nem használunk annyi memóriát.

5.1.6 Események

A Node.js alap API készletének nagy része aszinkron eseményvezérelt architektúra alapján készült. Ebben vannak bizonyos objektumok, amiket emittereknek hívunk és ezek bocsájtanak ki eseményeket, melyek végül egy másik objektumot, ami hívnak meg. Ezt az objektumot, amit meghívnak listenernek nevezzük. Minden eseményt létrehozó objektumnak az EventEmitter nevezetű osztályból kell származnia. Ezekhez az

on() metódus segítségével tudunk hozzákapcsolni egy vagy több függvényt, az emit() metódussal pedig triggerelni tudjuk az eseményt. Erre egy példa az alább látható. Létrehozunk egy EventEmitter példányt és megadjuk neki, hogy mi történjen, ha végrehajtódik.

```
const EventEmitter = require('events');
class MyEventEmitter extends EventEmitter {}
const myEmitterinstance = new MyEventEmitter ();
myEmitterinstance.on('event', () => {
  console.log('an event occurred!');
});
myEmitterinstance.emit('event');
```

5.2 Express.js

Az Express.js egy keretrendszer a Node.js alkalmazásokhoz, melynek célja, hogy a fejlesztéseket egyszerűbbé, biztonságosabbá és gyorsabbá tegye. Ha tisztán csak Node.js-t használnánk, akkor sok minden nekünk kellene megírni minden alkalommal, ami nem csak sok időbe telne, de növelné a hibák lehetőségének kockázatát is, mi leginkább az üzleti logikával akarunk foglalkozni.

A keretrendszer telepítése egyszerűen megoldjató az npm segítségével. Minden ugyan úgy működik, mint a modulok esetében. Az Express.js sok hasznos funkcióval rendelkezik egyik ilyen például a Routing.

Routing alatt azt értjük, amikor az alkalmazás eldönti, hogy milyen válaszokat küldünk vissza a kliensnek az alapján, hogy milyen végpontra érkezett a kérés. Például egy bejelentkező oldalra más adatokat kell visszaküldeni, mint egy beállítások oldalra. Ezt egyszerűsíti le nekünk az Express.js routing funkciója. Egy route meghatározása a következőképpen néz ki.[18]

```
app.METHOD(PATH,HANDLER)
```

A method a kérés típusát jelenti, ugye ez lehet get, post, put, delete. A path az útvonal a szerveren. A handler pedig az a metódus, amit végrehajtunk, ha a route egyezzik a megadottal.

Ahhoz, hogy képeket, vagy fájlokat vagy a saját kódunkat, például egy CSS-t el tudjunk érni a szerveren az express.static() metódust kell használnunk. A metódusba meg kell adni azt a root mappát, amelyből a fájlokat szeretnénk használni. Amennyiben több mappából is kell fájlokat használnunk, arra is van lehetőség, egyszerűen újból meg kell hívni a metódust a mappára.

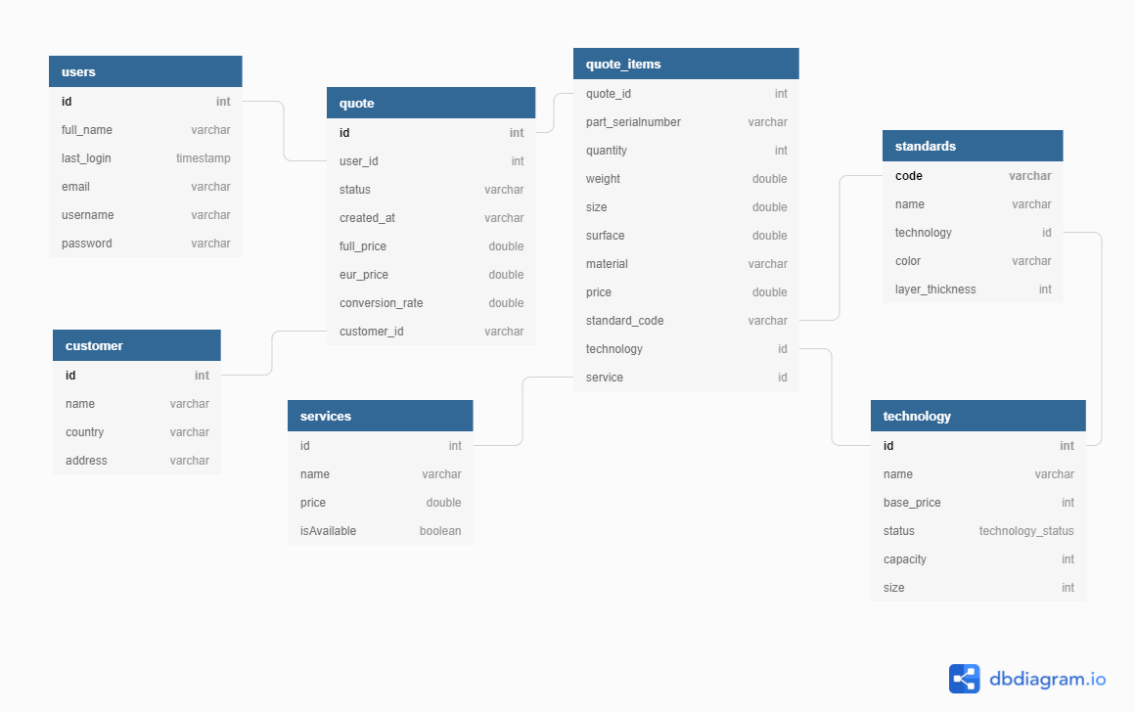
6 Adatbázis megtervezése

Ebben a fejeztben az alkalmazásom tervezett adatbázis architektúráját szeretném bemutatni. Azonosítom, hogy milyen táblákra lesz szükségem az adatok tárolására és ezek között milyen kapcsolatokra lesz szükség. Alapvető attribútumokat is meghatározom, de ezeknek a száma a fejlesztés során növekedhet vagy csökkenhet. Az adatbázis implementációját majd MySQL környezetben fogom elvégezni.

6.1 Szükséges Adatbázis táblák meghatározása

Ebben a fejezetben az adatbázis tábláit fogom meghatározni. A tervezett táblákat és leírásait egy táblázatban foglalom össze, valamint a köztük lévő kapcsolatokat és a tervezett attribútumokat, amik később bővülhetnek még, egy diagrammon ábrázolom.

Tábla név	Leírás
User	Felhasználók adatainak tárolására szolgál.
Standard	Szabványok tárolására szolgál.
Quote	Ajánlatok tárolására szolgál.
Services	A nyújtott plusz szolgáltatások tárolására. Például a csomagolás, vizsgálás stb. .
Technology	A rendelkezésre álló felületkezelési technológiák adatainak tárolására.
Quote_items	Az ajánlaton szereplő tételek, maga az alkatrészek adatainak tárolására szolgál.
Customer	Vevők adatainak tárolására szolgál.



6. ábra Tevezett adatbázis táblák és kapcsolataik

Irodalomjegyzék

[1] Gajdos Sándor Adatbázisok előadás PPT
https://db.bme.hu/~gajdos/2012adatb2/3.%20eloadas%20NoSQL%20ppt.pdf
Megtekintve: 2021.03.05

[2] Gajdos Sándor Adatbázisok előadás PPT
https://db.bme.hu/~gajdos/2012adatb2/3.%20eloadas%20NoSQL%20adatb%E1zisok%20doc.pdf
Megtekintve: 2021.03.05

[3] NoSQL vs SQL összehasonlítás
https://hun.small-business-tracker.com/what-is-nosql-nosql-databases-explained-789967
Megtekintve: 2021.03.05

[4] Microsoft NoSQL
https://azure.microsoft.com/hu-hu/overview/nosql-database/
Megtekintve: 2021.03.05

[5] Szemléletető ábra a join típusokra
https://www.google.com/url?sa=i&url=https%3A%2F%2Fwww.dofactory.com%2Fsql%2Fjoin&psig=AOvVaw1o90QceFDAUcujdcLkMu4b&ust=1620224564134000&source=images&cd=vfe&ved=0CAIQjRxqFwoTCOjLnYCdsPACFQAAAAAdAAAAABAD
Megtekintve: 2021.03.05

[6] React Fejlesztői dokumentáció
https://hu.reactjs.org/docs/rendering-elements.html
Megtekintve: 2021.03.18

[7] React komponensek életciklusa ábra
https://projects.wojtekmaj.pl/react-lifecycle-methods-diagram/
Megtekintve: 2021.03.18

[8] React Dokumentáció az űrlapokról
https://hu.reactjs.org/docs/forms.html
Megtekintve: 2021.03.18

[9] Vue.js
http://www.inf.u-szeged.hu/~tarib/javascript/vuejs.html#az-adat-es-a-metodus-tulajdonsagok
Megtekintve: 2021.03.20

[10] Vue.js dokument
https://v3.vuejs.org/guide/introduction.html#declarative-rendering
Megtekintve: 2021.03.20

[11] Angular.js dokumentáció
https://docs.angularjs.org/guide/directive
Megtekintve: 2021.03.26

[12] JavaScript leírás
http://nyelvek.inf.elte.hu/leirasok/JavaScript/index.php?chapter=27
Megtekintve: 2021.04.01

[13] Keretrendszerek összehasonlítása
https://academind.com/tutorials/angular-vs-react-vs-vue-my-thoughts/#framework-backgrounds
Megtekintve: 2021.04.03

[14] Keretrendszerek összehasonlítása szempontok szerint
https://code.tutsplus.com/hu/articles/angular-vs-react-7-key-features-compared--cms-29044
Megtekintve: 2021.04.04

[15] Angulat vs React
https://www.cuelogic.com/blog/what-are-the-differences-between-angular-and-react
Megtekintve: 2021.04.06

[16] React Router dokumentáció
https://reactrouter.com/web/guides/primary-components
Megtekintve: 2021.04.12

[17] Node.js dokumentáció
https://nodejs.org/api/stream.html#stream_stream
Megtekintve: 2021.04.16

[18] Express.js dokumentáció
https://expressjs.com/en/guide/routing.html
Megtekintve: 2021.04.17