Általános információk

A diplomaterv szerkezete:

1. Diplomaterv feladatkiírás
2. Címoldal
3. Tartalomjegyzék
4. A diplomatervező nyilatkozata az önálló munkáról és az elektronikus adatok kezeléséről
5. Tartalmi összefoglaló magyarul és angolul
6. Bevezetés: a feladat értelmezése, a tervezés célja, a feladat indokoltsága, a diplomaterv felépítésének rövid összefoglalása
7. A feladatkiírás pontosítása és részletes elemzése
8. Előzmények (irodalomkutatás, hasonló alkotások), az ezekből levonható következtetések
9. A tervezés részletes leírása, a döntési lehetőségek értékelése és a választott megoldások indoklása
10. A megtervezett műszaki alkotás értékelése, kritikai elemzése, továbbfejlesztési lehetőségek
11. Esetleges köszönetnyilvánítások
12. Részletesés pontos irodalomjegyzék
13. Függelék(ek)

Felhasználható a következő oldaltól kezdődő Diplomaterv sablon dokumentum tartalma. Ügyeljen a konzulens nevét és a beadás évét jelölő szövegdobozokra, mert azokra külön ki kell adni a frissítést. A mezők tartalma a sablonban a dokumentum adatlapja alapján automatikusan kerül kitöltésre.

A diplomaterv szabványos méretű A4-es lapokra kerüljön. Az oldalak tükörmargóval készüljenek (mindenhol 2.5cm, baloldalon 1cm-es kötéssel). Az alapértelmezett betűkészlet a 12 pontos Times New Roman, másfeles sorközzel.

Minden oldalon - az első négy szerkezeti elem kivételével - szerepelnie kell az oldalszámnak.

A fejezeteket decimális beosztással kell ellátni. Az ábrákat a megfelelő helyre be kell illeszteni, fejezetenként decimális számmal és kifejező címmel kell ellátni. A fejezeteket decimális aláosztással számozzuk, maximálisan 3 aláosztás mélységben (pl. 2.3.4.1.). Az ábrákat, táblázatokat és képleteket célszerű fejezetenként külön számozni (pl. 2.4. ábra, 4.2 táblázat vagy képletnél (3.2)). A fejezetcímeket igazítsuk balra, a normál szövegnél viszont használjunk sorkiegyenlítést. Az ábrákat, táblázatokat és a hozzájuk tartozó címet igazítsuk középre. A cím a jelölt rész alatt helyezkedjen el.

A képeket lehetőleg rajzoló programmal készítsék el, az egyenleteket egyenlet-szerkesztő segítségével írják le.

Az irodalomjegyzék szövegközi hivatkozása történhet a Harvard-rendszerben (a szerző és az évszám megadásával) vagy sorszámozva. A teljes lista névsor szerinti sorrendben a szöveg végén szerepeljen (sorszámozott irodalmi hivatkozások esetén hivatkozási sorrendben). A szakirodalmi források címeit azonban mindig az eredeti nyelven kell megadni, esetleg zárójelben a fordítással. A listában szereplő valamennyi publikációra hivatkozni kell a szövegben. Minden publikáció a szerzők után a következő adatok szerepelnek: folyóirat cikkeknél a pontos cím, a folyóirat címe, évfolyam, szám, oldalszám tól-ig. A folyóirat címeket csak akkor rövidítsük, ha azok nagyon közismertek vagy nagyon hosszúak. Internet hivatkozások megadásakor fontos, hogy az elérési út előtt megadjuk az oldal tulajdonosát és tartalmát (mivel a link egy idő után akár elérhetetlenné is válhat), valamint az elérés időpontját.

Fontos:

* a szakdolgozat készítő/diplomatervező nyilatkozata (a jelen sablonban szereplő szövegtartalommal) kötelező előírás Karunkon, ennek hiányában a szakdolgozat/diplomaterv nem bírálható és nem védhető!
* mind a dolgozat, mind a melléklet maximálisan 15 MB méretű lehet!

Jó munkát, sikeres szakdolgozat készítést ill. diplomatervezést kívánunk!

FELADATKIÍRÁS

A feladatkiírást a **tanszék saját előírása szerint** vagy a tanszéki adminisztrációban lehet átvenni, és a tanszéki pecséttel ellátott, a tanszékvezető által aláírt lapot kell belefűzni a leadott munkába, vagy a tanszékvezető által elektronikusan jóváhagyott feladatkiírást kell a Diplomaterv Portálról letölteni és a leadott munkába belefűzni (ezen oldal HELYETT, ez az oldal csak útmutatás). Az elektronikusan feltöltött dolgozatban már nem kell megismételni a feladatkiírást.



Budapesti Műszaki és Gazdaságtudományi Egyetem

Villamosmérnöki és Informatikai Kar

Irányítástechnika és Informatika Tanszék

Orbán Tamás

Konzulens

Dr. Simon Balázs

BUDAPEST, 2020

Tartalomjegyzék

[Összefoglaló 3](#_Toc332798843)

[Abstract 4](#_Toc332798844)

[1 Bevezetés 4](#_Toc332798845)

[1.1 Formázási tudnivalók 4](#_Toc332798846)

[1.1.1 Címsorok 4](#_Toc332798847)

[1.1.2 Képek 4](#_Toc332798848)

[1.1.3 Kódrészletek 4](#_Toc332798849)

[1.1.4 Irodalomjegyzék 4](#_Toc332798850)

[2 Utolsó simítások 4](#_Toc332798851)

[Irodalomjegyzék 5](#_Toc332798852)

[Függelék 5](#_Toc332798853)

Hallgatói nyilatkozat

Alulírott **Orbán Tamás**, szigorló hallgató kijelentem, hogy ezt a szakdolgozatot 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, 2020. 05. 18.

...…………………………………………….

Orbán Tamás

# Összefoglaló

Ide jön a ½-1 oldalas magyar nyelvű összefoglaló, melynek szövege a Diplomaterv Portálra külön is feltöltésre kerül.

# Abstract

Ide jön a ½-1 oldalas angol nyelvű összefoglaló, amelynek szövege a Diplomaterv Portálra külön is feltöltésre kerül.

# A mikroszolgáltatások architektúra és a felhasznált technológiák bemutatása

## A monolit és mikroszolgáltatások architektúra

### Monolit architektúra

Szerveroldali alkalmazások fejlesztésekor a következő komponenseket kell megvalósítani:

* Nézet – felelős HTTP kérések és a rájuk adott válaszok (HTML, vagy webszolgáltatások esetén XML/JSON) kezeléséért
* Üzleti logika – az alkalmazás üzleti logikájának megvalósítása
* Adatbázis kezelés – SQL vagy NoSQL adatbázisok kezelésére létrehozott adatkezelési objektumok
* Alkalmazás integráció – integráció más szolgáltatásokkal (pl. JMS vagy REST API)

Az architektúra modularitásának ellenére az alkalmazás egy egységként van csomagolva és telepítve.

#### A monolit architektúra előnyei:

* Egyszerű fejleszteni.
* Könnyű tesztelni. Például egy teljes end-to-end teszt elvégezhető az alkalmazás felhasználói felületén végzett (manuális vagy automatizált) tesztekkel.
* Egyszerű telepíteni. Az összecsomagolt alkalmazás egy egységként telepíthető.
* Horizontálisan könnyen skálázható több alkalmazáspéldány futtatásával egy loadbalancer mögött.

#### A monolit architektúra hátrányai:

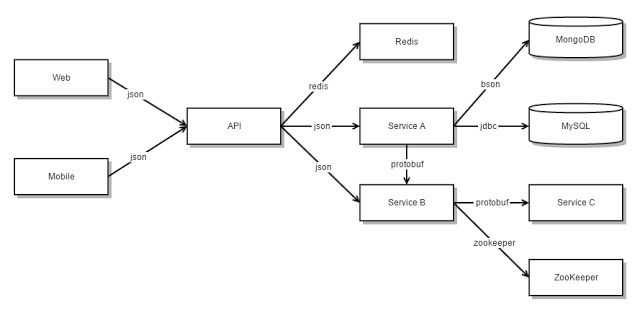
* Limitált az alkalmazás mérete és komplexitása.
* Az alkalmazás túl nagy ahhoz, hogy teljes egészében átlátható legyen, így nehéz gyorsan működőképes módosítást végezni rajta.
* Az alkalmazás mérete lelassíthatja az indulásának az idejét.
* Minden frissítéskor az alkalmazás egésze újratelepítendő.
* A módosítások hatása nem mindig egyértelmű és átlátható, így sok regressziós tesztet kell végezni.
* Az alkalmazás skálázása nehéz lehet, ha különböző modulok azonos erőforrásoktól függenek.
* Megbízhatósági szempontból is jelentkezhetnek problémák. Bármely modulban lévő hiba (pl. memória szivárgás) megállíthatja az egész folyamatot. Mivel az összes példány megegyezik, a hiba jelen van az egész alkalmazásban.
* Nehéz új technológiák bevezetése. A nyelvekben vagy keretrendszerekben történő módosítás hatással van az egész alkalmazásra, így ezek elvégzése rendkívül költséges (mind időben, mind pénzben).

### Mikroszolgáltatások architektúra

Az alapvető ötlet az, hogy az alkalmazás felbontható kisebb, egymással kommunikáló szolgáltatások csoportjára. Minden mikroszolgáltatás egy kis önálló alkalmazás, amely megvalósítja az üzleti logikát és a különböző adaptereket. Néhány mikroszolgáltatás kiajánlhat REST, RPC, vagy üzenet (messaging) alapú API-okat, és a legtöbb szolgáltatás más szolgáltatások API-ját használja. Más szolgáltatások megvalósíthatják a (pl. webes) felhasználói felületet.

A mikroszolgáltatások tervezési mintának jelentős hatása van az alkalmazás és az adatbázis kapcsolatára. Egy egységes, közös adatbázis séma helyett minden szolgáltatásnak meg van a saját adatbázis sémája. Ez a hozzáállás ellentmond a szervezeti szintű adatmodell lehetőségének., és előfordulhat adatok duplikációja. Ugyanakkor a szolgáltatásonkénti adatbázis séma szükséges a mikroszolgáltatások architektúrához, ugyanis ez biztosítja a szolgáltatások laza kapcsolatát. Minden szolgáltatás saját adatbázispéldánnyal rendelkezik, amely a legjobban megfelel az igényeinek.

Néhány API használható mobilalkalmazásból, webes böngészőből, webalkalmazásból, vagy asztali vastagkliensből. Ezen alkalmazásoknak nincsen direkt hozzáférésük a back-end szolgáltatásokhoz. Ehelyett a kommunikációt egy köztes komponens, az úgynevezett API Gateway (API útválasztó) kezeli. Ez felel a terhelés elosztásáért (load balancing), cache-elésért, hozzáfárés-kezelésért és monitorozásért.



. ábra: Egy példa mikroszolgáltatások architektúra használatára

#### A mikroszolgáltatások architektúra előnyei

* A komplexitási problémákat megoldja az alkalmazás kezelhető méretű részekre bontásával, amelyek gyorsabbal fejleszthetők, emellett sokkal egyszerűbben átláthatók és karbantarthatók.
* Lehetővé teszi, hogy minden szolgáltatás külön fejleszthető legyen azon csapat által, amely a legjobban ért az alkalmazás adott részéhez.
* Új technológiák bevezetése sokkal egyszerűbb. A fejlesztők szabadon válogathatnak a technológiák között, nem kell a projekt indulásakor rögzített technológiákhoz ragaszkodniuk.
* Az architektúra lehetővé teszi a szolgáltatások önálló telepíthetőségét, így a folyamatos telepítés lehetősége is meg van, még komplex alkalmazásoknál is.
* A mikroszolgáltatások külön-külön is skálázhatók.

#### A mikroszolgáltatások architektúra hátrányai

* Növeli a feladat komplexitását az elosztottsága által. Ki kell választani és implementálni egy folyamatok közötti kommunikációs mechanizmust, és kezelni kell a részleges és kommunikációs hibákat.
* Az alkalmazás adatbázisa partícionált. Azon üzleti folyamatok, amelyek több üzleti objektumon is módosítást végeznek, ebben az architektúrában különböző szolgáltatások különböző adatbázisát használják. Legtöbbször az elosztott tranzakciókezelés nem jelent megoldást, így a fejlesztőknek fokozatos konzisztencia (eventual consistency) alapú megoldást kell használniuk, ami sok kihívást hordoz magában.
* A szoftver tesztelése sokkal komplexebb a monolit alkalmazásokhoz képest. Egy szolgáltatás teszteléséhez az összes olyan szolgáltatás is futtatandó vagy mockolandó, amelyektől függ.
* Nehezebb azon igények implementálása, amelyek több szolgáltatást is érintenek. Monolit alkalmazásban egyszerűen módosítható egy modul, integrálhatók a változások és telepíthető, mindez egy folyamatként. Mikroszolgáltatásoknál körültekintő tervezést és lebonyolítást igényel egyes szolgáltatások módosítása.
* Komplexebb az alkalmazás telepítése. A monolit alkalmazásokkal szemben egy mikroszolgáltatás alapú szoftver többnyire rengeteg szolgáltatásból áll, szolgáltatásonként több példánnyal. Minden ilyen példányt konfigurálni, telepíteni, skálázni, és monitorozni kell. Ezen kívül implementálni kell egy szolgáltatás feltáró mechanizmust (service discovery). Ekkora komplexitásnál a telepítések automatizálása minden esetben szükséges.

### Összegzés

Komplex alkalmazások fejlesztése rendkívül nehéz és körülményes. A monolit architektúra jobban megfelel kisebb, egyszerűbb alkalmazásokra. Mikroszolgáltatások fejlesztését két irányból lehet elkezdeni: egy monolit alkalmazásból kiindulva, annak részekre bontásával (dekompozíció), vagy ezen lépés kihagyásával, közvetlenül. Mindkét esetben fontos a monolit architektúra ismerete, ugyanis ez az alapja a mikroszolgáltatások architektúrának, amelyben minden szolgáltatás az előbbi szerint van implementálva. A mikroszolgáltatások tervezési minta komplex, folyamatosan bővülő alkalmazásoknál használandó. Az architektúra fő célja a komplex rendszerek kezelése, ennek eléréséhez azonban újabb komplexitásokkal, és implementációs kihívásokkal jár.

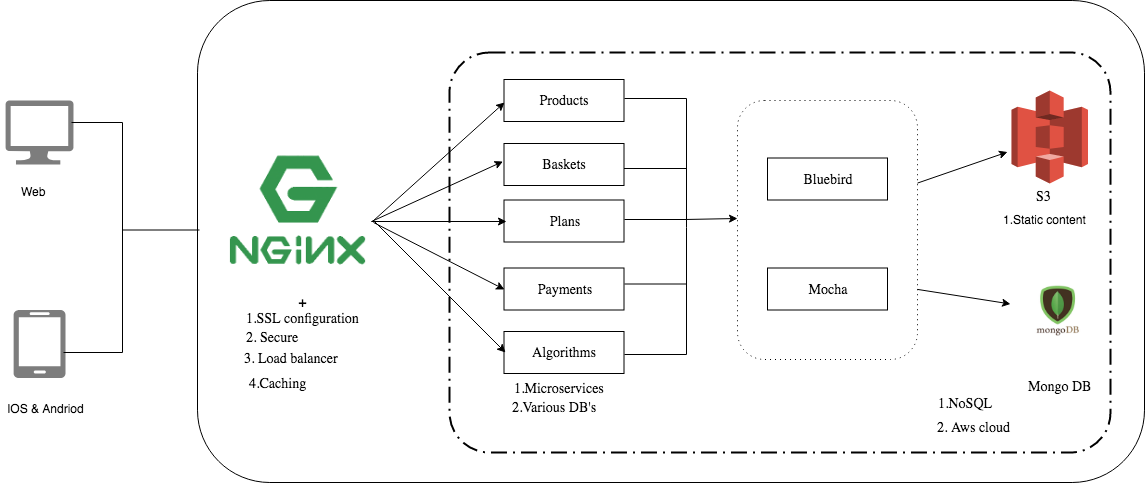
## Esettanulmány: Cadenza

A Cadenza.tech vállalati alkalmazás kezdeti időben egy monolit szoftver volt, amely MEAN (Mongo, Express, Angular, Node) stackre épült. 125 API-t szolgáltatott különböző kategóriákban: felhasználókezelés, termék leltár stb. A rendszer fenntartási költsége havi 40000 dollár volt az AWS Cloud platformon. Két fő probléma állt fent: magas költségek, és az alkalmazás időnkénti összeomlása.

**Első probléma: Alkalmazás összeomlása**

Az alkalmazást modulokra bontották, amelyek a következő entitások kezeléséért voltak felelősek:

* Felhasználó
* Termék
* Megrendelés
* Előfizetés
* Fizetés



. ábra: Cadenza 2.0 beta architektúra

Minden szolgáltatás 5 vagy 6 darab API-al dolgozott. A mikroszolgáltatások tervezési mintára való áttérésből következett:

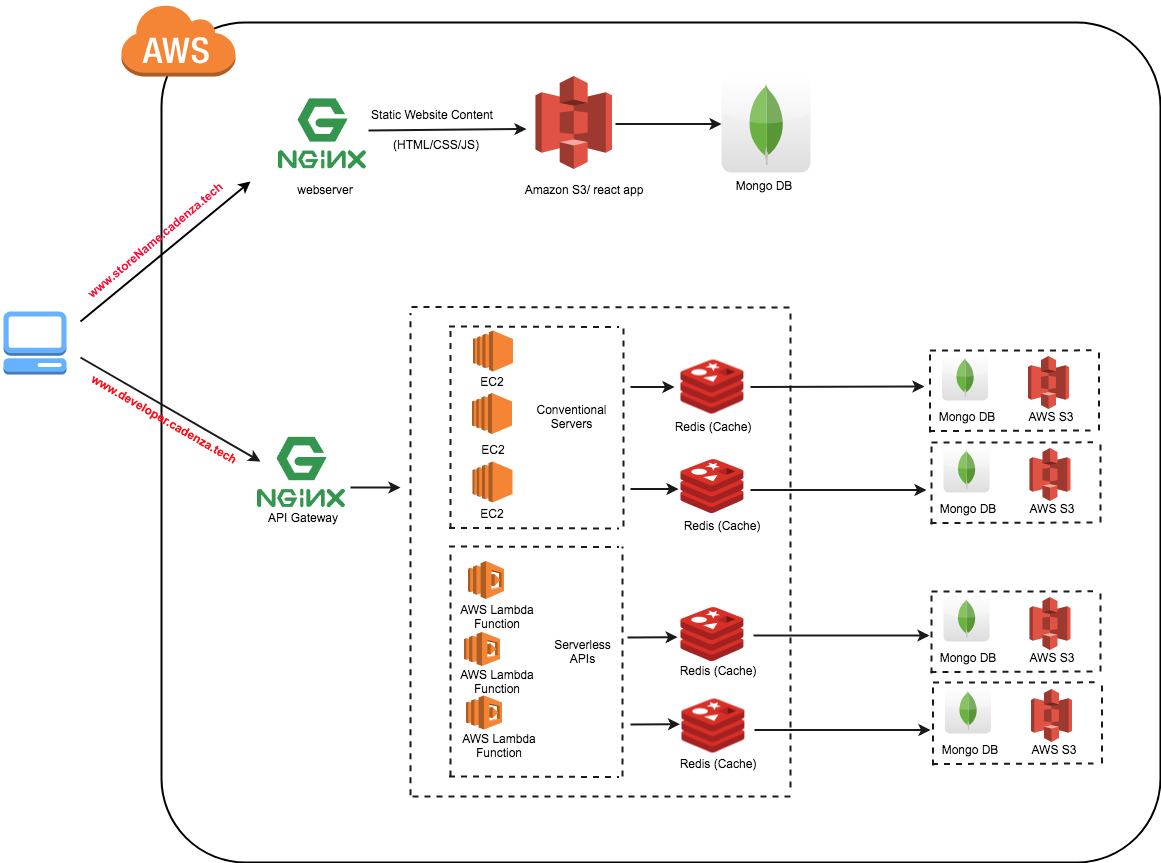
* Lehetőség nyílt a szolgáltatások nyelv- és adatbázisfüggetlen programozására. Megszűnt a hosszútávú függőség egy, a projekt elején választott technológiától. A pénzügyek kezelése Python-ban, a felhasználókezelés Node.js-ben íródott.
* A frontend, backend és DevOps csapatok párhuzamosan tudtak dolgozni. Könnyű volt a hibakeresés.
* Megvalósíthatók lettek a CI/CD folyamatok nagy méretű alkalmazásoknál is.

**Második probléma: Nagy költségek**

A mikroszolgáltatások könnyedén hosztolhatók felhőben, Dockeres környezetben, vagy bármely más szerver platformon, és szolgáltatásonként nagyon könnyen skálázhatók.

Monolit platform feldarabolásánál fontos, hogy a létrejött kisebb egységek könnyen és automatizáltan telepíthetők legyenek.

Docker használatával könnyen futtatható a mikroszolgáltatás alapú alkalmazás a saját fejlesztői gépeken, majd ugyan ilyen módon futtatható a teszt és éles környezeteken is.



. ábra: Cadenza 2.2 beta architektúra diagram

Hagyományos szervereket használtak NodeJS, Python és Java API-okhoz, és AWS Lambda-t a szervernélküliekhez. A hibrid architektúra segített nagy mértékben csökkenteni a költségeket: a havi 40 ezer dolláros kiadás 7 ezresre csökkent.

## Felhasznált technológiák

Az alábbi fejezet bemutatja a szakdolgozat elkészítéséhez használt technológiákat.

### REST API

A REST a Representational State Transfer rövidítése, amely reprezentációs állapot szállítást jelent. Ez egy architekturális stílus elosztott szoftverrendszerekhez, így jól működik a mikroszolgáltatások architektúrával is. Mint minden más tervezési mintának, a REST-nek is meg vannak a saját megszorításai, amelyeknek betartásával RESTful interfészeket tervezhetünk és implementálhatunk. Ezek a szabályok a következők:

* Kliens–szerver – A felhasználói felület és az adattárolás különválasztásával növelhető a felhasználói felület portabilitása különböző platformok között, illetve az alkalmazás skálázhatósága az egyszerűsített szerver komponensek miatt.
* Állapotmentes – A kliens összes szerver felé indított kérésének tartalmaznia kell mindent, ami ahhoz kell, hogy a szerver megértse azt. A szerveren nem tárolódik semmilyen, a klienssel kapcsolatos kontextus. Minden ilyen információt a kliens tárol.
* Gyorsítótár – A kérésre adott válaszokban explicit meg kell jelölni, hogy a visszaadott adat cache-elhető, vagy sem. Ha egy válasz cache-elhető, akkor a kliens oldali gyorsítótárban eltárolható az adat későbbi felhasználásra.
* Egységes interfész – Ennek eléréséhez az alkalmazás komponensei viselkedésének több architekturális megszorításnak kell megfelelniük. A REST négy megszorítással van definiálva: erőforrások azonosítása, erőforrások módosítása reprezentációkkal, önleíró üzenetek, illetve hipermédia, mint az alkalmazás állapotának motorja.
* Többrétegű architektúra – Az alkalmazás minden rétege csak a közvetlenül alatta vagy felette lévő rétegekkel kommunikálhat.

A REST szolgáltatások fő absztrakciós szintje az erőforrás. Erőforrás bármi lehet, aminek nevet adhatunk (pl. dokumentum, kép, más erőforrások halmaza, stb…). A REST egy úgynevezett erőforrás-azonosítót használ az erőforrások azonosítására.

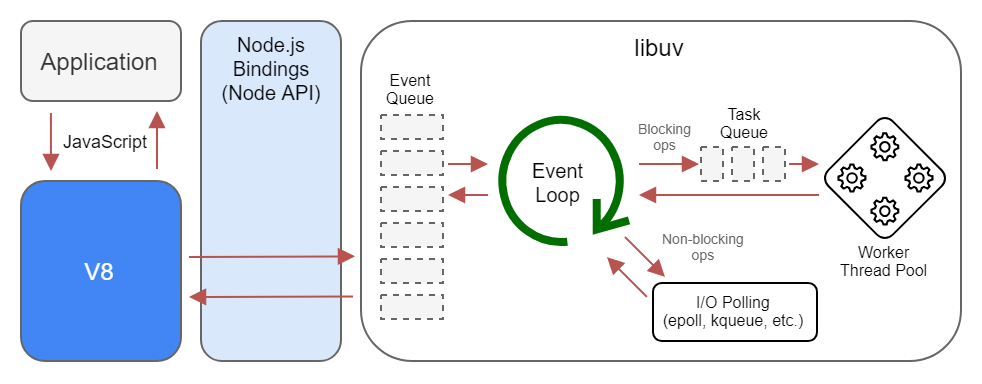
Egy erőforrás állapotát egy adott időpillanatban az erőforrás reprezentációnak hívják, amely adatokból, metaadatokból, és hipermédia linkekből áll. Ez utóbbi segíti a kliensek állapotátmenetét. A reprezentációnak önleírónak kell lennie: a kliensnek nem kell tudnia az adatok szerkezetét.

REST esetén definiálni kell úgynevezett erőforrás metódusokat. Ezek nem egyeznek meg a HTTP GET/POST/PUT/DELETE metódusokkal.

### Node.js

A Node.js egy nyílt forráskódú, cross-platform JavaScript futtatókörnyezet, amely JavaScript kódot futtat a böngészőn kívül. Ez lehetővé teszi a nyelv használatát CLI eszközök és szerveroldali szolgáltatások fejlesztéséhez. Nagy előnye, hogy ugyan az a programozási nyelv használható kliens-, és szerveroldali fejlesztéshez. A Node egy eseményvezérelt architektúrára épült, amely lehetővé teszi az aszinkron I/O műveleteket. Ezek célja, hogy optimalizálják az áteresztőképességet és a skálázhatóságot olyan webalkalmazásokban, ahol sok I/O műveletet kell végrehajtani. A futtatókörnyezet a Google V8 JavaScript motorjára épült.

A Node egyszálú eseménycikluson dolgozik, nem blokkoló I/O hívásokat használva, így akár több tízezer kérést is ki tud szolgálni, a szálak kontextusváltásának költsége nélkül. Minden kérés az eseményciklusba jut. Amennyiben a válasz előállításához szükség van I/O műveletre, az aszinkron módon kiszerveződik egy külön szálra, így párhuzamosan több I/O művelet is futhat.



. ábra: A NodeJS architektúrája

A Node.js csomagkezelője az npm (Node Package Manager), amely az npm registry-ből különböző csomagokat tud telepíteni.

### Express

Az express talán a legismertebb webes keretrendszer, amely lehetővé teszi webalkalmazások fejlesztését Node.js-ben. A következő főbb szolgáltatásokat nyújtja:

* HTTP kérések kezelése, különböző útvonalakon.
* Integrálható nézetrenderelő motorokkal, így dinamikus tartalmat tud válaszként adni.
* Konfigurálhatók a webalkalmazás csatlakozási pontjai port szinten, illetve azok a template-ek, amelyek a válasz generálásához szükségesek
* Hozzáadhatók személyre szabott middleware-ek, amelyek a kéréseket kezelik

Az Express önmagában egy nagyon egyszerű keretrendszer, ugyanakkor rengeteg különböző middleware csomag létezik, amelyek számos problémára nyújthatnak megoldást. Ezen csomagok segítségével könnyedén kezelhetők a cookie-k, session-ök, felhasználók bejelentkezései, URL paraméterek, POST adatok, biztonsági fejlécek és még sok más.

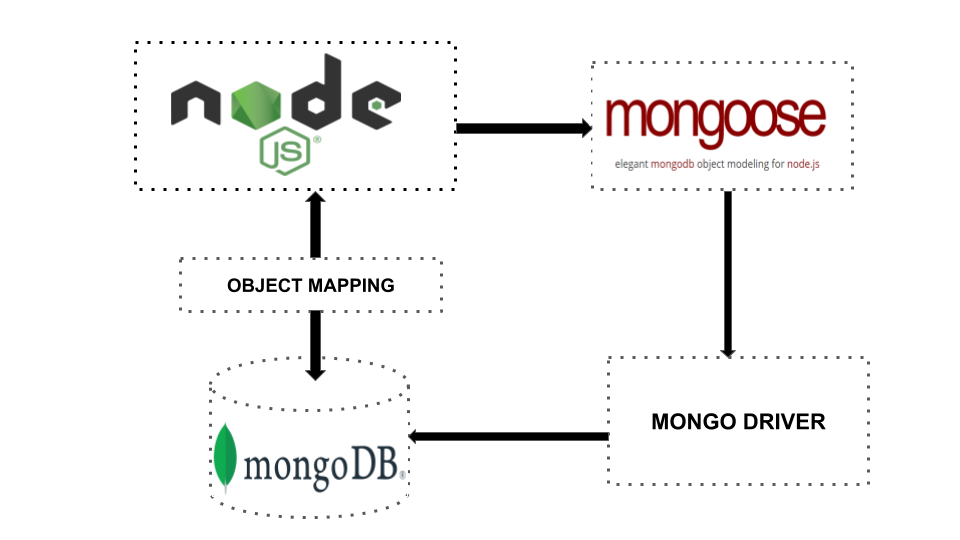
### MongoDB

A MongoDB egy skálázható és flexibilis dokumentum adatbázis (objektum tár). Lehetővé teszi például a személyre szabott indexek készítését és egymásba ágyazott dokumentumok kezelését. Az általa használt dokumentum modell egyszerű, könnyen megtanulható, mégis megoldást nyújthat komplex problémákra is. Főbb tulajdonságai:

* Az adatokat flexibilis, JSON-hoz hasonló dokumentumokként tárolja, ami azt eredményezi, hogy különböző dokumentumoknak lehetnek más mezői, és az adatok struktúrája az idő előrehaladásával változhat (nincs rögzített séma).
* Minden dokumentumnak össze van rendelve egy objektummal az alkalmazásban, így könnyű az adatokkal dolgozni.
* Ad-hoc módon lehet adatokat lekérni és indexelni, továbbá valós időben aggregálni, ezzel megkönnyítve az adatelemzést.
* Elosztott adatbázis lévén magas rendelkezésre állás, horizontális skálázhatóság és térbeli elosztottság könnyen megvalósítható.

### Mongoose

A Mongoose egy Objektum-Adat Modellező (Object Data Mapping, ODT) könyvtár MongoDB-hez és Node.js-hez. Kezeli az adatok közötti kapcsolatokat, sémavalidációt kínál és lefordítja az alkalmazás objektumait MongoDBdokumentumokra.



. ábra: A mongoose működése

### JWT (JSON Web Tokens)

A JWT egy szabvány (RFC 7519), amely egy olyan kompakt és zárt biztonsági megoldást definiál, amelynek célja a felek közti biztonságos információcsere. Ez az információ megerősíthető és megbízható a digitális aláírásnak köszönhetően. A JWT tokenek aláírhatók titok használatával (HMAC), vagy publikus és privát kulcspárokkal (RSA vagy ECDSA). A JWT a következőkre használható:

* Hozzáféréskezelés – Ez a leggyakrabban használt funkciója. Miután a felhasználó bejelentkezett, minden kérése tartalmazni fogja a tokent, amelynek segítségével védett útvonalakat, szolgáltatásokat és erőforrásokat érhet el.
* Információcsere – Felek közti biztonságos kommunikáció. Mivel a token aláírható (pl. publikus és privát kulcspárral), így biztosítható a küldő azonosítása. Ezen felül az aláírás a fejlécből és az adatból számítódik, így megerősíthető, hogy az üzenet a küldés óta nem változott meg.

Egy JWT token három részből áll: fejléc, adatok és aláírás. A fejléc tipikusan két komponensből áll: a használt algoritmusból és a token típusából. Az adatrész egy entitásra vonatkozó kijelentésekből, és további információkból áll. Végül maga a token (példaként HMAC SHA256 algoritmust használva) a következőképpen számítódik:

HMACSHA256(

base64UrlEncode(fejléc) + „.” +

base64UrlEncode(adat),

titok)

Amint ezt megkapta a kliens, minden kérésnél küldi a szerver felé a kérés fejlécében, az alábbi módon:

Authorization: Bearer <token>

### Axios

Az Axios egy JavaScript könyvtár, amelynek segítségével HTTP kéréseket küldhetünk Node.js alkalmazásból. Előnye, hogy támogatja az ES6 Pomise API-t. A klasszikus JavaScript fetch() függvényhez képest jobb hibakezelésre nyújt lehetőséget.

### Nginx

Az Nginx egy nyílt forráskódú szoftver, amelynek fő felhasználási módjai:

* webes kiszolgálás
* gyorsítótárazás.
* terheléselosztás
* média streaming

Ezen dolgozat keretében a szoftver egy egyszerű API kapuként szolgál, amelyen keresztül elérhetőek az alkalmazás szolgáltatásai.

### Vue.js

A Vue.js egy webes felhasználói felületek építésére szolgáló keretrendszer. A fő könyvtára kizárólag a megjelenítési rétegre fókuszál, így könnyű integrálni más könyvtárakkal, vagy már létező projektekkel is. Ezen felül készíthetők vele egy oldalas (single page) alkalmazások, amelyek megvalósításához a keretrendszer kliensoldalon renderelt, komponens alapú technológiát használ.

### Docker

A Docker egy olyan eszköz, amelyet arra fejlesztettek ki, hogy megkönnyítsék az alkalmazások fejlesztését, telepítését és futtatását konténerek használatával. A konténerek lehetővé teszik a fejlesztő számára az alkalmazás és függőségek (pl. könyvtárak) összecsomagolását. Ennek köszönhetően az alkalmazás bármely Linux gépen futni fog, a gép beállításaitól függetlenül. A Docker hasonlít egy virtuális gépre, ugyanakkor nem hoz létre egy teljes virtuális operációs rendszert, hanem a host gép kernelét és szolgáltatásait használja, ezzel csökkentve az alkalmazás méretét.

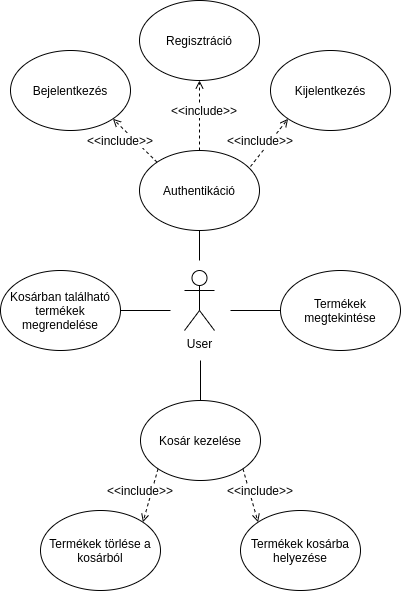
### Docker compose

A Docker compose egy többkonténeres alkalmazás futtatásához használható eszköz, ahol egy YAML fájl használatával konfigurálhatóak az alkalmazás szolgáltatásai. Ezután egy egyszerű paranccsal létrehozható és elindítható az összes szolgáltatás.

# Tervezés

Az általam tervezett szoftver egy webáruház alapja, amelyben virtuális termékek tekinthetők és vásárolhatók meg. A cél egy minimális, később testre szabható, ugyanakkor minden fő funkciót megvalósító alkalmazás tervezése volt.

## Use case diagram



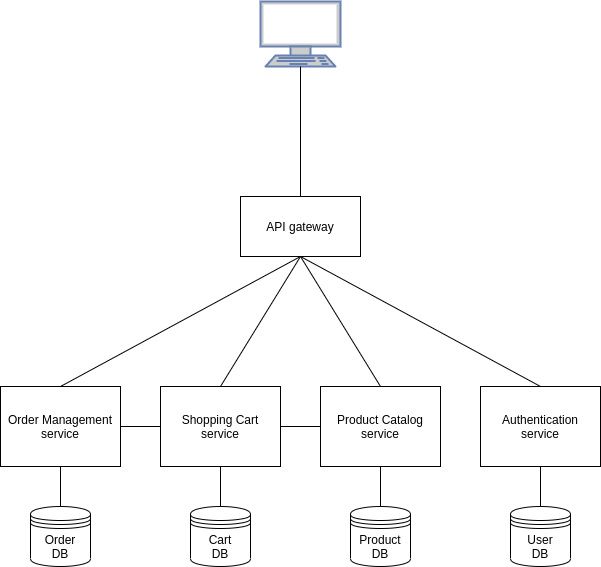
. ábra: Use case diagram

## Funkcionális követelmények

|  |  |  |
| --- | --- | --- |
| Követelmény | Aktor | Leírás |
| FREQ01 | Vásárló | A felhasználó regisztrálhat a szolgáltatásra, amely hatására létrejön a felhasználói fiókja a megadott email címmel, névvel és jelszóval. |
| FREQ02 | Vásárló | A felhasználó be tud jelentkezni létező felhasználói fiókjába az email címe és a jelszava megadásával. |
| FREQ03 | Vásárló | A felhasználó, kijelentkezhet abból a felhasználói fiókból, amelybe be van jelentkezve. |
| FREQ04 | Vásárló | A felhasználó meg tudja tekinteni a webáruház kínálatát. |
| FREQ05 | Vásárló | A felhasználó tetszőleges számú terméket el tud helyezni a kosarában. Minden felhasználónak külön kosara van. |
| FREQ06 | Vásárló | A felhasználó el tud távolítani termékeket a saját kosarából. |
| FREQ07 | Vásárló | A felhasználó meg tudja rendelni a kosarában található termékeket. |
| FREQ08 |  | Az alkalmazás robusztus, egy szolgáltatás leállása nem jár az egész alkalmazás leállásával. |
| FREQ09 |  | Felhasználói adatok biztonságos kezelése és tárolása. |

## Architektúra

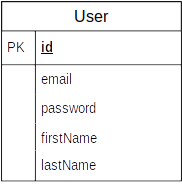
A követelmények felsorolása és megértése után elkészítettem az architektúra, a komponensek és az adatmodellek tervét. Ebben a szakaszban még nem szerepelnek konkrét technológiák, csupán magas absztrakciós szinten az alkalmazás komponensei és azok tulajdonságai. Az architektúrámat a mikroszolgáltatások architektúra alapján hoztam létre. Ebben az esetben, egy ilyen méretű alkalmazásnál alapvetően nem indokolt a használata, azonban a tervezési minta bemutatása volt a cél.



. ábra: Magas szintű architektúra

### Komponensek

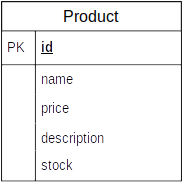
* **Kliens** – A kliens webes felületen tudja használni a webáruház által nyújtott szolgáltatásokat.
* **API gateway** – A webáruház szolgáltatásai ezen átjárón keresztül érhetők el. Konfigurálható, hogy kik férnek hozzá a szolgáltatásokhoz.
* **Authentikációs szolgáltatás** – Fő célja a felhasználók azonosítása, létrehozása és törlése. Az authentikációt olyan módon szerettem volna megvalósítani, hogy ezzel a szolgáltatással csak a kliensnek kelljen kommunikálnia.
* **Felhasználói adatbázis** – Ebben az adatbázisban egy -féle entitás entitás létezik, a felhasználó. Az jelszavak védelme érdekében a jelszó mező titkosítva lesz eltárolva. A felhasználó entitás attribútumai:



. ábra: Felhasználó entitás

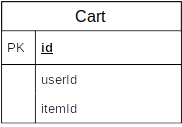
Az email mező értéke egyedi.

* **Termék katalógus szolgáltatás** – A termékek megjelenítéséért és kezeléséért felelős szolgáltatás. A termékeken értelmezettek az alapvető CRUD műveletek (létrehozás, olvasás, módosítás, törlés), azonban az olvasáson kívül ezt csak egy adminisztrátori jogosultsággal rendelkező felhasználó teheti meg egy külön felületen.
* **Termék adatbázis** – A webáruházban szereplő termékek tárolási helye. A felhasználói adatbázishoz hasonlóan egyetlen entitástípus jellemzi, a termék. A termékekről eltároljuk a nevüket, árukat, leírásukat és készletinformációjukat. Ezek az attribútumok a webáruház tematikájához igazodva tetszőlegesen bővíthetők:



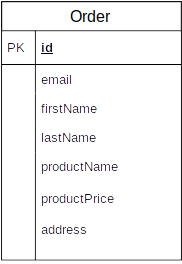
. ábra: Termék entitás

* **Bevásárlókosár szolgáltatás** – A felhasználó által kosárba helyezett termékeket kezeli. Azért született erre a funkcióra egy különálló szolgáltatás, hogy egy vásárló későbbi visszatérésekor ne vesszen el a kosár tartalma, az alkalmazás mégis állapotmentes maradjon.
* **Bevásárlókosár adatbázis** – Szintén egy entitástípus tárolására terveztem, ugyanakkor nem konkrét termékeket, hanem csak azok azonosítóját tárolja, így csökkentve az adatbázisok közti redundanciát:



. ábra: Bevásárlókosár entitás

* **Megrendeléskezelő szolgáltatás** – Amennyiben egy vásárló összeállította a rendelése tartalmát a bevásárlókocsijában és megrendelte azt, ezen szolgáltatás megkapja az összes, rendeléssel kapcsolatos információt, és elmenti a saját adatbázisába későbbi feldolgozásra.
* **Megrendelés adatbázis** – Itt tárolódik a megrendelések összegzése. A termékekre illetve felhasználókra nem referencia van, az attribútumok ki vannak fejtve, többek között a termékek árváltozásai miatt. A megrendelés egyedtípus attribútumai:



. ábra: Rendelés entitás

A cím attribútumot a felhasználó a rendelés véglegesítésénél adhatja meg.

## Technológiák kiválasztása

A magas absztrakciós szintű architektúra megtervezése után a következő lépés azon technológiák kiválasztása, amelyek legjobban illeszkednek az adott feladat megoldásához. A választott technológiákat és azok indoklását tárgyalja ez az alfejezet.

### Kliens/UI

A klienstechnológia kiválasztásánál törekedtem az egyszerűségre, egy egyoldalas (single page) alkalmazást készítése volt a terv, így három technológia került szóba: React, Angular és Vue.js. A választás végül a Vue.js-re esett, ugyanis a három keretrendszer közül ennek a legegyszerűbb a felépítése, kisebb alkalmazások készítésénél ez nagy előnyt tud jelenteni. A Vuetify könyvtár használata megkönnyíti a felhasználói felület stílusozását és reszponzivitást, így ezeket nem kell kézzel, külön implementálni. Ezek mellett tárolhatók kliens oldalon információk a Vuex segítségével, ez egyfajta gyorsítótárazást tesz lehetővé, csökkentve a szolgáltatások felé indított kérések számát.

### API gateway

Az API gateway megoldásra az egyik terv egy saját implementáció volt, amely egy egyszerű proxy-ként funkcionált: a klienstől érkező kéréseket szétosztotta a megfelelő szolgáltatások felé, majd a kapott eredményeket visszaadta a válaszban. Végül nem ezt a megoldást, hanem az nginx-et választottam, ugyanis nagyon egyszerűen konfigurálható a routing és a HTTPS használata is. Ezeknél sokkal többre képes az nginx, de esetemben ezeknél a megoldásoknál nem kellett több.

### Adatbázisok

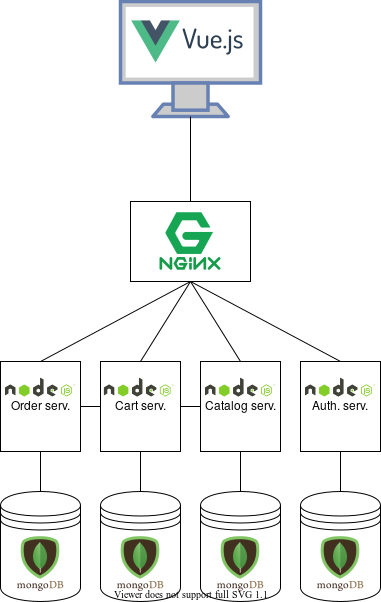
Addattárolásra valamilyen NoSQL adatbázist szerettem volna, ugyanis csak elemi CRUD műveleteket valósítok meg az adatokon, tárolt eljárásokat nem használok. Döntésem a MongoDB-re esett, ugyanis egy entitástípus tárolásához tökéletes megoldás, ugyanakkor lehetővé teszi az entitások attribútumainak bővülését. Ez utóbbi akkor lehet hasznos, ha az elkészített vázat valamilyen tematikus, teljes webáruházzá kellene alakítani.

### Szolgáltatások

Egy olyan technológiát kellett találnom, amellyel a REST-es HTTP üzeneteket könnyű kezelni, továbbá az adatbázisban lévő adatokat valamilyen ORM (Object Relation Mapping, Objektum Reláció Megfelelőség) keretrendszer segítségével lehet olvasni és módosítani. A felmerült technológiák a következők voltak: Spring Boot (Kotlin), .NET Core (C#) és Node.js (Javascript). Mivel minden szolgáltatásom egyszerű felépítésű, és főként demonstrációs célokat szolgál, a Node.js mellett döntöttem. Sokkal kevesebb (jelen esetben felesleges) konfigurációval elindítható a szolgáltatás. A REST-es végpontok implementációjához az express keretrendszert választottam. Az adatbázis ekkor már adott volt, így kiválasztottam a Mongoose könyvtárat, amely lényegesen leegyszerűsíti az adatok kezelését, és az adatbázissal való kapcsolat kezelését.

### Technológiai megoldásokkal kiegészített architektúra

Az előzőekben elkészített architekturális ábra a választott technológiákkal kiegészítve:



## REST végpontok tervezése

A REST-es API-ok interfészét szolgáltatásonként terveztem meg, külön-külön Swagger fájlokban, azonban nagy terjedelmük miatt itt csak a szöveges leírásukat adom meg.

### Authentikációs szolgáltatás

* GET /user – JWT token alapján visszaadja az éppen bejelentkezett felhasználót, ha van ilyen.
* POST /autchenticate – A kérésben érkező email és jelszó ellenőrzése után, amennyiben azok helyesek, visszadja a JWT tokent, és eltárolja a sütik közt.
* POST /register – Amennyiben a megadott email címmel és jelszóval nem létezik felhasználó, regisztrálja a felhasználót, majd visszatér a JWT tokennel, a belépéshez hasonlóan.
* POST /logout – Kijelentkezteti a felhasználót a JWT süti törlésével.

### Termék katalógus szolgáltatás

* GET /products – Visszatér az összes termékkel, termékek listája formájában.
* GET products/{id} – Visszatér az adott azonosítójú termékkel.

### Bevásárlókocsi szolgáltatás

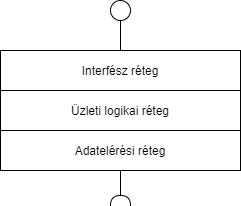
* GET /cart – JWT token alapján visszaadja a felhasználó kosarában lévő termékeket.
* PUT /cart – JWT token alapján hozzáadja a felhasználó kosarához a kérésben küldött terméket.
* DELETE /cart – JWT token alapján törli a felhasználó kosarából a kérésben küldött terméket.

### Megrendeléskezelő szolgáltatás

* PUT /orders – A megrendelés minden információja a kérésben érkezik, amit aztán a szolgáltatás eltárol az adatbázisában későbbi feldolgozásra.

## Szolgáltatások felépítése

Az egyes szolgáltatások egy monolit alkalmazáshoz hasonlóan követik a klasszikus háromrétegű REST esetén használt architektúrát.



. ábra: A szolgáltatások rétegei

Az egyes rétegek feladatai:

* Interfész réteg: a REST kérések és válaszok kezeléséért felel, vele kommunikál a kliens.
* Üzleti logikai réteg: itt vannak megvalósítva az üzleti objektumok transzformációi és az üzleti folyamatok algoritmusai. Kapcsolatban áll az interfész és az adatelérési réteggel is.
* Adatelérési réteg: kiszolgálja az üzleti logikai réteget azokkal az adatokkal, amelyeket valamilyen interfészen keresztül kap. Ez az interfész lehet valamilyen üzenet alapú kommunikációs csatorna (pl. REST vagy SOAP), vagy mint esetünkben, egy adatbázis csatlakozási pontja is.

# Megvalósítás

## Fejlesztői környezet konfigurálása

A fejlesztés első lépéseként összeállítottam a fejlesztőkörnyezetet. A kód írásához Visual Studio Code-ot használtam, az adatbázis kezeléséhez pedig MongoDB Compass-t. Mivel az elkészítendő szoftver több, egymástól független szolgáltatásból állt, így automatizáltam az adott komponensek újrafordítását, amennyiben a hozzájuk tartozó forráskód megváltozott. Ehhez a nodemon nevű szoftvert választottam, amely aktívan figyeli a forráskód változásait, és amennyiben módosulást észlel, újrafordítja és futtatja az adott szolgáltatást. Megírtam a Dockerfile-t a szolgáltatásokhoz (mindre egységes), a klienshez, és az nginx-hez:

FROM ubuntu:18.04

RUN apt-get update --fix-missing && apt-get upgrade -y

RUN apt-get install -y curl gnupg git

RUN groupadd -g 1000 app && useradd -m -u 1000 -g app app && chown app:app /home

RUN curl -sL https://deb.nodesource.com/setup\_10.x | bash -

RUN apt-get update --fix-missing && apt-get install -y nodejs build-essential

RUN npm install -g @vue/cli --unsafe

WORKDIR /home/frontend

USER app

CMD npm install ; npx vue-cli-service build --watch

FROM node

WORKDIR /usr/src/app

CMD ["sh", "-c", "npm install && npm run start"]

szolgáltatás Dockerfile

kliens Dockerfile

FROM nginx:latest

RUN chown nginx:nginx /home

COPY nginx.conf /etc/nginx/conf.d/default.conf

nginx Dockerfile

Készítettem egy docker-compose fájlt, amely tartalmazza az alkalmazásom konténereinek leírását, többek közt a bázis képet (base image), a konténeren belüli és kívüli portok megfeleltetését, a források felcsatolását a konténerekhez, a konténerek egymástól való függőségét stb. A konfigurációs fájl mérete miatt csupán az egyik szolgáltatást, illetve a hozzá tartozó adatbázist mutatom be, a többi szolgáltatásra és adatbázisra hasonlóan ismétlődik, eltérő elnevezésekkel és portokkal:

version: "3.1"

services:

auth\_server:

build: ./backend/auth\_server

container\_name: auth\_server

ports:

- "3000:3000"

depends\_on:

- mongo\_auth

environment:

- MONGO\_HOST=mongo\_auth

volumes:

- ./backend/auth\_server:/usr/src/app

mongo\_auth:

image: mongo

container\_name: mongo\_auth

ports:

- "27022:27017"

. . .

frontend:

environment:

- CHOKIDAR\_USEPOLLING=true

build:

context: ./frontend

container\_name: webshop\_frontend

volumes:

- ./frontend:/home/frontend

ports:

- "8080:8080"

nginx:

build:

context: ./nginx

container\_name: webshop\_nginx

ports:

- "80:80"

- "443:443"

volumes:

- ./frontend:/home

depends\_on:

- auth\_server

- product\_catalog

- shopping\_cart

- order\_management

docker-compose.yml részlet

Megírtam az nginx konfigurációs fájlt, amely azonosítja a szolgáltatások végpontjait, illetve szétosztja irányukba a gateway-en keresztül érkező kéréseket.

upstream auth\_server {

server auth\_server:3000;

keepalive 16;

}

upstream product\_catalog {

server product\_catalog:3001;

keepalive 16;

}

upstream shopping\_cart {

server shopping\_cart:3002;

keepalive 16;

}

upstream order\_management {

server order\_management:3003;

keepalive 16;

}

server {

listen 80 default\_server;

server\_name \_;

location / {

root /home/dist;

index index.html;

try\_files $uri $uri/ /index.html;

}

location /api/auth {

proxy\_pass http://auth\_server;

proxy\_set\_header Host $host;

expires -1;

add\_header Pragma "no-cache";

proxy\_http\_version 1.1;

proxy\_set\_header Connection "";

}

location /api/catalog {

proxy\_pass http://product\_catalog;

proxy\_set\_header Host $host;

expires -1;

add\_header Pragma "no-cache";

proxy\_http\_version 1.1;

proxy\_set\_header Connection "";

}

. . .

nginx.conf részlet

## Komponensek megvalósítása

### Szolgáltatások

A szolgáltatásokat a tervezésnél leírt (4.6 fejezet), három rétegű architektúrának megfelelően készítettem el. A lentebb lévő kódrészleteket az authentikációs szolgáltatásból emeltem ki, a többi komponenst is hasonló módon implementáltam.

. . .

let UserSchema = new Schema({

  email: {

    type: String,

    unique: true,

    required: true

  },

  password: {

    type: String,

    required: true

  },

  firstName: {

    type: String,

    required: true

  },

  lastName: {

    type: String,

    required: true

  }

});

export default model('User', UserSchema);

Az adatelérési réteget a mongoose könyvtár által szolgáltatott sémákkal írtam le.

User.js részlet

Az üzleti logikát kezelő modul részeit a feladat megvalósításánál szolgáltatásoknak (service) neveztem. Ezen szolgáltatások függvényeit használja a felette található, üzenetkezelésért felelős interfész réteg. A rétegek közötti adatok továbbadását Promise-ok használatával oldottam meg. Az üzleti logika algoritmusaiban megtalálhatók egyéb, külső könyvtárak (pl. bcrypt, md5, jwt) függvényei. Alább az egyik ilyen szolgáltatásból emeltem ki egy részt, amely az új felhasználók létrehozását végzi.

auth.js részlet

. . .

export const register = async (user) => {

  return new Promise((resolve, reject) => {

    bcrypt.hash(user.password, saltRounds, (err, hash) => {

      if (err) {

        reject(err);

      }

      let userToSave = new User({...user, password: hash});

      resolve(userToSave.save());

    });

  });

};

. . .

A legfelső, interfész réteget az express keretrendszer routing megoldásával készítettem el, amely lehetővé teszi a REST végpontok létrehozását, a kérésekre adott válaszok státuszának és a visszaadott adatoknak a specifikálását. Megadható, hogy melyik URL-en, milyen HTTP metódusra miként reagáljon a szolgáltatás.

. . .

router.post('/authenticate', async (req, res) => {

  const {

    email,

    password

  } = req.body;

  authenticate(email, password)

    .then((result) => {

      res.cookie('token', result.token, {

        secure: false,

        httpOnly: true

      });

      res.status(200).json({

        ...result.user

      });

    })

    .catch(() => {

      res.status(400).json({

        status: 400,

        message: 'Email or password is incorrect'

      });

    });

});

. . .

router.js részlet

Minden szolgáltatásnak van egy gyökér fájlja (index.js), amely elindítja az express szervert, illetve csatlakozik a saját adatbázisához a megadott beállításokkal. Mivel ezek a beállítások (a csatlakozási címen és porton kívül) nem változnak, ezért nem vezettem ki külön konfigurációs fájlba.

const app = express();

const port = process.env.PORT || 3000;

const mongo\_host = process.env.MONGO\_HOST || 'localhost';

app.use(cors());

app.use(bodyParser.json());

app.use(cookieParser());

app.use('/api/auth/', UserRouter);

const mongoOptions = {

  useNewUrlParser: true,

  useUnifiedTopology: true,

  useCreateIndex: true

};

mongoose.connect(`mongodb://${mongo\_host}:27017/users`, mongoOptions).then(

  () => {

    console.log("Connected to MongoDB")

    app.listen(port,() => console.log(`Listening on port: ${port}`));

  },

  (err) => console.log(err)

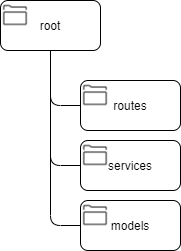
);

index.js részlet

Az adatbázis csatlakozási pontját környezeti változókból kapja a szerver, de van egy alap, fallback beállítása is. A következő middleware-eket használtam:

* CORS (Cross-Origin Resource Sharing) – olyan HTTP fejléceket használ, amely lehetővé teszi egy webalkalmazásnak, hogy elérjen olyan erőforrást, amely egy másik szerveren található meg.
* Body parser – a HTTP kérések body részét Javascript objektummá alakítja automatikusan, így könnyebb kezelni az ott érkező adatokat.
* Cookie parser – a http kérésben érkező sütiket teszi könnyen elérhetővé, a body parserhez hasonlóan.
* Router – az előzőekben bemutatott express router használata, a megadott gyökér ponttól (ezesetben /api/auth).

A három rétegnek megfelelően alakítottam ki a projekt struktúráját. Külön könyvtárat kaptak a routerek, a szolgáltatások, és a mongoose sémák is. Ezeken kívül a szolgáltatás gyökérkönyvtárában kapott helyet a belépési pont (index.js) és a Dockerfile.



. ábra: egy szolgáltatás könyvtárstruktúrája

### Kliens

# Tesztelés és értékelés

# Összefoglalás

# Irodalomjegyzék

1. https://articles.microservices.com/monolithic-vs-microservices-architecture-5c4848858f59
2. https://restfulapi.net/
3. https://developer.mozilla.org/en-US/docs/Learn/Server-side/Express\_Nodejs/Introduction
4. www.mongodb.com/what-is-mongodb
5. https://mongoosejs.com/
6. https://jwt.io/
7. https://www.nginx.com/resources/glossary/nginx/
8. https://vuejs.org/
9. https://opensource.com/resources/what-docker
10. <https://docs.docker.com/compose/>
11. <https://medium.com/@dearsikandarkhan/microservices-architecture-for-e-commerce-f8b49270e72f>

Irodalomjegyzék

1. Levendovszky, J., Jereb, L., Elek, Zs., Vesztergombi, Gy.: Adaptive statistical algorithms in network reliability analysis, Performance Evaluation - Elsevier, Vol. 48, 2002, pp. 225-236
2. National Istruments: LabVIEW grafikus fejlesztői környezet leírása, <http://www.ni.com/> (2010. nov.)
3. Fowler, M.: UML Distilled, 3rd edition, ISBN 0-321-19368-7, Addison-Wesley, 2004
4. Wikipedia: Evaluation strategy, <http://en.wikipedia.org/wiki/Evaluation_strategy> (revision 18:11, 31 July 2012)

Függelék