Diplomamunka

Mezei Botond, Szabó Benedek

Debrecen
2023

Debreceni Egyetem
Informatikai Kar
Számítógéptudományi Tanszék

Edzőterem működését támogató szoftver PureScript és Vue.js alapokon

Diplomamunka

Készítette:

Mezei Botond és Szabó Benedek
programtervező informatika szakos hallgatók

Témavezető:

Dr. Battyányi Péter
adjunktus

Debrecen
2023

Tartalomjegyzék

Bevezetés

A funkcionális programozás egy kevésbé elterjedt programozási paradigma. Noha számos népszerű nyelv használ funkcionális elemeket [10] [15] [16], a teljesen funkcionális fejlesztés nem annyira népszerű. Megismerése az oktatásban sem igazán hangsúlyos. Ennek ellenére számos előnnyel rendelkezik, melyeket a későbbi fejezetekben fogunk részletezni. Másfajta gondolkosámódot igényel, mint az imperatív megközelítés: a "hogyan?" helyett a "mit csináljon a program?" kérdésre adja meg a választ.

A dolgozatban tárgyalt szoftver backend komponensének elkészítésére a PureScript nyelvet választottuk. Ez egy tisztán funkcionális és erősen típusos programozási nyelv, mely Javascript-re fordul. Azért választottuk ezt a nyelvet, mert tisztán funkcionális, a közelmúltban megjelent nyelv, mely a többi funkcionális nyelvhez (Haskell, Elm) képest modernebb, erőteljesebb, könnyebben használható és teljesebb [1].

Munkánk elsődleges célja egy edzőterem működését támogató szoftver elkészítése PureScript backend és Vue.js frontend használatával. Ezen felül szeretnénk megvizsgálni és bemutatni a funkcionális nyelvek használatának előnyeit és lehetőségeit leginkább webalkalmazások fejlesztése során. A szoftver elkészültét követően implementálásra kerül ugyanez a backend Java nyelven a Spring keretrendszer használatával. Célunk a két verzió sebességének és fejlesztési tapasztalatainak összehasonlítása. A sebesség összehasonlítására többféle operációs rendszerrel és különböző specifikációkkal rendelkező számítógépeken ugyanazokat a kéréseket 10, 100 és 1000-es kötegekben küldjük el a két különböző szervernek, majd elemezzük az így kapott eredményeket.

A huszonegyedik században egy korszerű edzőteremnek szüksége van egy szoftverre, amely a mindennapi üzletmenetet támogatja. Választásunk azért erre a területre esett, mert egyrészt mindketten szeretünk edzeni járni, másrészt pedig a szoftver összetettsége már alkalmas lehet a PureScript nyelv megismerésére, lehetőségeinek és előnyeinek bemutatására, illetve egy elterjedtebb technológiával (Java, Spring) készült változatával való összehasonlításra.

Az alkalmazás felhasználói az edzőterem recepciós kollégái. A rendszerben lehetőség van rögzíteni:

- vendégeket
- bérlettípusokat (jegytípusokat)
- a felhasználók jegyeit, bérleteit
- öltözőszekrényeket

A *Vendégek* oldalon listázhatjuk a regisztrált vendégeket, újat regisztrálhatunk, a meglévők adatait módosíthatjuk, vendéget törölhetünk. A vendégek számára bérleteket vagy jegyeket lehet vásárolni. A továbbiakban bérletnek nevezzük azt a tagásgot, ami egy időtartamra engedélyezi a terem használatát, jegynek pedig ami alkalmakhoz kötött. A jegyeknek is van egy elévülési ideje. Minden belépőnek három ára van: teljes, diák és klub ár. A felhasználóknak klubtagság vásárlására van lehetőségük, amely különböző előnyökkel jár, például kedvezőbb bérletárak.

A vezérlőpulton lehetőség van a vendégek beléptetésére: a regisztrált vendégek listájából ki kell választani az illetőt. A listában csak az aktív bérlettel vagy érvényes jeggyel rendelkezők jelennek meg. A illetőhöz hozzárendelődik a neme szerint egy szabad öltözőszekrény. Jeggyel rendelkező vendégek számára a jegyen található fennmaradó alkalmak száma eggyel csökken. Az öltözőszekrények megtekintésére egy külön lap szolgál. Itt nemek szerint meg lehet tekinteni minden szekrény állapotát, illetve az előzménytörténetét, vagyis mikor és ki használta. A foglalt szekrények piros színnel jelennek meg.

A vezérlőpulton a beléptetett vendégek kijelentkeztetésére is van lehetőség. Ezt úgy tehetjük meg, hogy a feljövő listából kiválasztjuk az illetőt. Ilyenkor felszabadul a szekrénye.

Minden oldal tetején egy navigációs sáv található, mellyel a megfelelő lapra ugorhatunk.

A közös munka során az egyik legnépszerűbb verziókezelő rendszert, a git-et használtuk a Github szolgáltatóval. Az extrém programozás módszerei közül többet is alkalmaztunk, például a páros programozást, mely hatékonynak bizonyult. Lényege, hogy az egyik fejlesztő írja a kódot, a másik pedig segíti közben.

1. Szakirodalmi áttekintés

1.1. Funkcionális programozás

A funkcionális programozás egy programozási paradigma. Több programozási nyelv is tartozik ide, melyeket bizonyos tulajdonságok, módszereik, lehetőségeik, gondolkodási logikájuk köti össze. Az egyik korai funkcionális nyelv a Lisp, melyet John McCarthy alkotott meg az 1950-es évek végén [3]. Ismertebb funkcionális nyelvek például Haskell, Elm, Erlang, Scala vagy a PureScript. A funkcionális programozás során a fejlesztő azt specifikálja a programban, hogy mit kell kiszámítani, és nem azt, hogy hogyan, milyen lépésekben [8]. A program függvények hívásából és ezek kiértékeléséből áll, nincs értékadás, csak érték kiszámítás [8]. A függvényt leginkább úgy értjük, hogy egy leképezés egy adott halmazról egy másik halmazra[11]. A listák (vagy halmazok) kezelésének ezért kiemelt szerep jut a funkcionális nyelvekben. A program tartalmazhat a nyelvben előre definiált, és a programozó által definiált függvényeket. A függvények névvel és opcionálisan argumentumokkal rendelkeznek. Az előállított érték(ek) megegyező paraméterekkel mindig ugyanaz. A rekurzió egy nagyon gyakori koncepció funkcionális nyelvekben.

Egy másik nagy paradigma az imperatív programozás. Az imperatív nyelvek lényege, hogy a programozó a lépéseket definiálja a kódban, melyet a számítógépnek el kell végeznie (utasítások). A legismertebb imperatív nyelv a C.

A logikai programozási paradigma lényege, hogy a programozó állításokat, szabályokat rögzít, melyek használatával a gép automatikusan kikövetkezteti, hogy egy kérdéses állítás igaz, vagy sem. Ilyen nyelv például a Prolog.

Manapság az egyik legelterjedtebb paradigma az objektumorientált programozás. Ez egy gondolkodásmód, tervezési módszer is egyben. A valós világot osztályok formájára képezi le, melynek egyedeit objektumok személyesítenek meg. Legfőbb elvei az egységbezárás (encapsulation: az adatmodell és az eljárásmodell szétválaszthatatlansága), öröklődés (újrafelhasználhatóság kiterjesztése), hozzáférés-szabályozás és többalakúság (polimorphism: lehetővé teszi, hogy ugyanarra az üzenetre különböző objektumok a saját módjukon válaszoljanak) [8].

A ma leginkább használatos nyelvekre általánosságban igaz az, hogy nem csupán egyetlen paradigmát követnek tisztán, hanem többet vegyítenek. Ennek az az oka, hogy minden paradigmának megvan a maga előnye (és persze hátránya is), és ezeket az előnyöket érdemes kihasználni. Például a Java (2023. áprilisában a harmadik legtöbbet használt nyelv [9]) alapjaiban imperatív, objektumorientált, de a nyolcas verziótól kezdve [10] megjelennek benne funkcionális

elemek, például a lambda kifejezések. A nyelv hivatalos oldala [10] "erőteljes kiterjesztés"-nek nevezi ezt a lépést. A Java mellett több elterjedt, nem tisztán funkcionális nyelv, mint például a Python vagy a C++ is épít be funkcionális elemeket [15] [16].

1.1.1. A funkcionális programozás előnyei

A funkcionális programozás, mint paradigma, számos előnnyel rendelkezik. Az imperatív gondolkodás kötöttségét egy másfajta megközelítéssel oldja fel. A lényeg, hogy *mit* csináljon a program, és nem az, hogy *hogyan*. Ez az ötlet a programozótól is másfajta megközelítést, gondolkodásmódot kíván, és máshogy strukturált kódot is fog eredményezni.

Ránézésre jobban érthető, átláthatóbb, esztétikusabb kód. A szoftver "viselkedése" jobban olvasható [1]. Ez az előny egyszerűen a funkcionális szintaktikából és gondolkodásmódból adódik. A következő Java kódrészletek Boris Radojicic 2022-es cikkéből [11] származnak. Az elsőn az iteratív megközelítés látható:

```
public List<String> getAddresses(List<Person> persons) {
    List<String> addresses = new ArrayList<>();
    for (int i = 0; i < persons.size(); i++) {
        Person person = persons.get(i);
        if (person.hasValidData()) {
            String address = person.getAddress();
            addresses.add(address.trim());
        }
    }
    return addresses;
}
```

A következő kódrészlet pedig a funkcionális megközelítést alkalmazza:

```
public List<String> getAddresses(List<Person> persons) {
    return persons.stream()
            .filter(person -> person.hasValidData())
            .map(person -> person.getAddress())
            .map(address -> address.trim())
            .collect(Collectors.toList());
}
```

Míg a két kód ugyanazt a viselkedést eredményezi, a kettőre ránézve elmondható, hogy a második olvashatóbb és tömörebb.

Nagy fokú újrafelhasználhatóság. Mivel egy funkcionális program gyakorlatilag függvények deklarálásából és függvényhívásokból áll, ezeket a függvényeket nagy mértékben újra lehet hasznosítani, elkerülve a feleslegesen duplikált kódrészleteket. A felesleges kód rontja az átláthatóságot és több erőforrást is igényel.

Könnyebb tesztelhetőség. Mivel a funkcionális programban nincsenek állapotok és mellékhatások, így könnyebb lefedni az összes esetet. Illetve ebből kifolyóan, a függvénynek bizonyos bemenő paraméterekre mindig ugyanazt az egyértelmű kimenetet kell előállítania.

A rekurzió hatékony használata. A rekurzió a funkcionális nyelvek gyakori eleme. Átláthatóvá és egyértelművé teszi a kódot teszi a kódot bizonyos problémák esetében, ahol a rekurzív definiálás a természetesebb megoldás. A rekurziónak egy speciális esete a farokrekurzió. Ez azt jelenti, hogy a rekurzív hívás a legutolsó művelet, amit a függvény végrehajt. Nagy előnye, hogy idő és erőforrás takarékos a nem farokrekurzív függvényekkel szemben, ugyanis a fordító kevesebb információt kell, hogy tároljon ilyen módon a veremben.

1.1.2. A funkcionális programozás nehézségei

Az előnyök között első helyen állt az átláthatóság és érthetőség. Ehhez viszont hozzátartozik az, hogy a fejlesztő előtte megismerje és elsajátítsa a funkcionális programozás gondolkodásmódját, ami néha bizony nem egyszerű és nem a legtermészetesebb megoldásnak tűnhet.

Memóriahasználat és teljesítmény. Mivel a funkcionális nyelvek nem használnak érték-átadást, sokszor egy változó értékének megváltozása helyett létrejön egy újabb, ami így több erőforrást igényel [12]. A legtöbb funkcionális nyelv ennek ellenére számos módon ígyekszik ezt kompenzálni: farokrekurzió, lusta kiértékelés, "smart linking". A több paradigmát támogató vagy OOP nyelvek nem feltétlen implementálják ezeket az optimalizációkat [12].

Mivel a funkcionális nyelvek használata kevésbé elterjedt, kevesebb eszköz, keretrendszer áll rendelkezésre és kevesebb a felhasználók, szakértők száma. Ennek következtében a fejlesztést nehezíti néha a dokumentáció hiányossága vagy teljes hiánya, illetve az interneten fellelhető minta kódok hiánya és elavultsága.

1.2. PureScript

A nyelvet Phil Freeman tervezte 2013-ban, ugyanis nem volt megelégedve a Haskell-t Javascript-re fordító próbálkozásokkal (például Fay, Haste, vagy GHCJS) [29]. A nyelv teljes forráskódja és dokumentációja megtalálható a GitHubon [30]. Hasznos dokumentációs forrás továbbá a Pursuit [33], illetve a nyelv alkotója által publikált PureScript by Example könyv [2].

A PureScript egy erősen típusos, tisztán funkcionális nyelv. Javascript kódra fordul. Ennek előnye, hogy a kód írására egy szép, könnyen áttekinthető és egyértelmű nyelvet haszálunk, amiből egy hatékony Javascript kód generálódik. Amit Javascriptben meg lehet írni, azt nagyjából PureScriptben is [1]. A böngészőben és a szerveren egyaránt futtatható.

A nyelv alapja a Haskell, nagyban hasonlít rá. Maga a szintkaxis annyira hasonló, hogy néhány fejlesztői eszköz a kód megjelenítéséhez a Haskell kiemelését használja. A nyelv fordítója is Haskell nyelven íródott [1]. Charles Scalfani [1] szerint erősebb, de egyben könnyebben is használható, mint a Haskell.

A nyelv komoly előnye, hogy frontend és backend fejlesztésre egyaránt alkalmas lehet [1].

A PureScriptnek saját csomagkezelő - fordító szoftvere van, a Spago [31]. Forráskódja szintén elérhető a GitHubon. Munkánk során mi is ezt használtuk, véleményünk szerint nagyon egyszerűen használható, és probléma sem merült fel vele kapcsolatban.

A PureScript rendelkezik egy parancsoros végrehajtási móddal, ezt PSCI-nek vagy REPL-nek nevezik. Soronként történik a kód begépelése és végrehajtása. Több sort is megadhatunk egyszerre a *:paste* paranccsal (kilépés: ctrl + D). A nyelvvel való ismerkedés és a tesztelés során hasznos lehet, ugyanis lépésről lépésre tudjuk irányítani a kódot, például típusok lekérdezéséhez.

A következő kódcsipet a *Hello, world!* program PureScript változata:

```
module Main where

import Prelude

import Effect (Effect)
import Effect.Console (log)

main :: Effect Unit
main = log "Hello World!"
```

Előre definiált könyvtárakból lehetőség van függvényeket importálni, ezt a fájl elején tehetjük meg. Elég explicit módon kell megadni a konkrét behívandó

függvényeket és komponenseket. Ezután (mint tiszta funkcionális nyelvhez illik) függvények deklarációja és definíciója következik. A következő kódcsipet egy függvényt ábrázol:

```
add :: Int -> Int -> Int
add a b = a + b
```

Az első sor nem kötelező, de az átláthatóság érdekében érdemes megadni. Azt lehet belőle leolvasni, hogy az add egy olyan függvény, amely két Int típusú input paraméterrel rendelkezik, és kimeneti értékként egy Int-et állít elő. A függvény a következőt csinálja: [1]:

- kap egy vagy több bemeneti értéket
- elvégez bizonyos számításokat
- visszaad egy értéket

Egy tiszta függvény mellékhatás mente, és mindig ugyanazt a kimenetet állítja elő[1]. A tiszta funkcionális elvek szerint egy függvénynek csak egy kimeneti értéket adhat vissza. Ez nehézségnek tűnhet, de az úgy nevezett *Currying* koncepció segíthet: egy több paraméteres függvény egy paraméteres függvények halmazára cserélését jelenti.

A nyelv további jellemzője, hogy minden változó "Immutable", vagyis a létrejöttük után nem módosulhatnak. Ez is egy érdekes tulajdonság az imperatív világban járatosabb fejlesztők számára. Az imperatív nyelvekben is van lehetőség objektumoknak ilyen tulajdonság beállítására. PureScriptben a *let* kulcsszó gyakran segítségünkre lehet ilyenkor, például:

```
add :: Int -> Int
add x = do
  let y = 20
    x + 20
```

A fenti függvény visszaadja a bemeneti értéknél hússzal nagyobb integert. PureScriptben a sorok behúzása is lényeges, a Pythonéhoz hasonló elven működik. Ott, ahol a legtöbb nyelv zárójeleket használna, a PureScriptben behúzás található: eggyel bentebbi behúzás egy zárójelezett blokknak felel meg.

Az imperatív nyelvek egyik alap építőelemét, a ciklusokat is el kell felejteni PureScriptben. Más megközelítés helyettük például a rekurzió használata, mely a funkcionális nyelvekben gyakori koncepció. Példaként Charles Scalfani könyvéből [1] mutatok két Javascript kódcsipetet, melyek ugyanazt a célt szolgálják. A ciklusos megközelítés:

```
const factorial = n => {
    var result = 1;
    for (var i = 1; i <= n; ++i)
        result = result * i;
    return result;
};
```

Funkcionális módon:

```
const factorial = n => n === 0 ? 1 : n * factorial (n - 1);
```

A második kódcsipet úgy definiálja a függvényt, hogy azt mondja meg, *mit* számoljon ki, míg az első a *hogyan* kérdésre adja meg a választ. Ugyanez a kód PureScriptben:

```
factorial :: Int -> Int
factorial n =
   if n = 1 then 1 else factorial $ n - 1
```

A fenti két funkcionális függvény farokrekurzív, mivel a rekurzív hívás maga a függvényben az utolsó hívás. A rekurzió egy előnyös fajtája, mivel kevesebb memóriát igényel. Ez annak köszönhető, hogy ily módon a veremben kevesebb információ tárolódik, és a sebesség is szignifikánsan jobb lesz egy nem farokrekurzív függvényhez képest.

A PureScript továbbá statikus nyelv. Ez azt jelenti, hogy a típusok ellenőrzése fordítás alatt zajlik, és nem futásidő alatt. Előnye, hogy a hibák már fordítás közben kiderülnek, kevesebb tesztelés szükséges. Hátránya pedig, hogy a típusokra több figyelmet kell fordítani, limitálja a programozói szabadságot [1].

A nyelvvel kapcsolatban kissé negatívum élményként tapasztaltuk, hogy az általános dokumentáltság (nem magának PureScript nyelvnek a dokumentációja, inkább a rendelkezésre álló egyéb oktatási anyagok, könyvek, példák, megválaszolt kérdésekre gondolva) hiányos, sokszor nehéz volt emiatt haladni a fejlesztéssel. Ez véleményünk szerint egyrészt a kevésbé népes funkcionális fejlesztői közösségnek köszönhető, másrészt pedig a nyelv fiatal korának.

1.3. Vue.js

Dolgozatunk frontend része Vue.js-ben készült, ezért röviden szeretnénk összefoglalni annak alapjait, tulajdonságait és egy rövid példán keresztül a használatának könnyedségét.

1.3.1. JavaScript keretrendszerek

A Vue.js egy kliens-oldali keretrendszer, mely a JavaScript programozási nyelvre épül. A JavaScript 1996-os indulása óta mára már megkerülhetetlen része a webnek, a weben található oldalak több mint 95%-án használatban van valamilyen formában [17]. Az évek során a nyelvvel dolgozó fejlesztőknek köszönhetően számos különböző problémára megoldást nyújtó könyvtár született, mely mind a web mai formájának alakulásához járult hozzá.

A keretrendszerek is ilyen könyvtárak. Meghatározzák egy abban írt szoftver felépítését, mely olyan előnyökkel jár mint a kiszámíthatóság, fenntarthatóság illetve a skálázhatóság. A keretrendszerek megépítésének motivációja a fejlesztési munkamenet egyszerűsítésében keresendő. Nem vértezi fel új képességekkel az eredeti programozási nyelvet (jelen esetünkben a JavaScriptet), hanem átláthatóbbá, valamint könnyebben elérhetővé teszi azt. Általuk a fejlesztőknek csak azt kell meghatározniuk, hogy hogyan nézzen ki az általuk elkészíteni kívánt felhasználói felület, a megvalósításról pedig már a keretrendszer gondoskodik.

Az alább felsorolt tulajdonságok szintén a fejlesztési folyamatok megkönnyítéséhez járulnak hozzá:

- tesztelési és kódellenőrzési eszközök biztosítása,
- felhasználói felület részekre (komponensekre) osztása,
- útvonalkezelés megkönnyítése.

Az elvitathatatlan előnyökön felül természetesen van más nézőpont is ami felől vizsgálódnia szükséges egy fejlesztőnek, mielőtt egy keretrendszer használatába kezd, az alap JavaScriptet elfeledve. Fontos észben tartani, hogy a keretrendszerek elsajátításához időre van szükség, hogy az általa biztosított eszközöket a lehető leghatékonyabban tudja felhasználni a fejlesztő a céljai elérésére. Elengedhetetlen továbbá átgondolni az elkészítendő program célját. Egy olyan alkalmazás, amely semmilyen felhasználói interakcióval nem rendelkezik, nem biztos, hogy igényel egy olyan professzionális fejlesztői eszközt, mint egy keretrendszer, mely komplex struktúrája feleslegesen bonyolítja a fejlesztési folyamatot.

Ahhoz, hogy az elérhető keretrendszerek közül a legmegfelelőbb kerüljön kiválasztásra, érdemes szem előtt tartani néhány egyszerű szempontot. A választott keretrendszer dokumentációjának minősége és elérhetősége. Minél részletesebb, minél több példával rendelkezik, annál könnyebb lesz a fejlesztés során felmerülő problémákra megoldást találni vagy az implementációs kérdésekben döntést hozni. A dokumentációt kiegészítő szempont a fejlesztői közösség aktivitása, amely a speciális esetekben nyújthat segítséget, akár korábban feltett és megválaszolt kérdéseken keresztül.

1.3.2. A Vue.js rövid története és tulajdonságai

A Vue.js ötlete Evan You Google alkalmazott fejében született meg [5]. A munkája során szüksége lett volna egy olyan hatékony eszközre, ami gyorsan képes nagy mennyiségű HTML dokumentumot előállítani adatok és prototípusok felhasználásával. A már létező eszközök nem tudták teljes mértékben kielégíteni az igényeit [5]. Volt olyan, amelyik csupán struktúrát biztosított, de az adatkezelése nem volt ideális (Backbone), de volt olyan is, amelyik túl szigorú, kötött szabályrendszerével nem csak a fejlesztendő alkalmazás struktúráját, de a kódolási folyamatot is túlságosan nagy mértékben kívánta befolyásolni (Angular). A megfelelő megoldás hiánya arra késztette, hogy saját maga hozza létre az általa keresett keretrendszert. Így jelent meg a keretrendszer 2014-ben, mely azóta egy lelkes fejlesztő kis magánprojektjéből széles fejlesztői körökben elterjedt és folyamatosan fejlődő keretrendszerré nőtte ki magát.

A Vue.js figyelemre méltó eredménye, hogy fel tudta venni a versenyt olyan elterjedt, szintén JavaScriptre vagy TypeScriptre épülő keretrendszerekkel mint az Angular vagy a React. Ebben hatalmas szerepe van az elérhetőségének. Minden olyan böngésző képes a Vue.js-t kezelni, amelyik támogatja az ES2015 (ES6) JavaScript szabványt [18], ezzel a felhasználók 96,09%-ához jut el [19]. Megbízhatóságát biztosítja a több mint 1,5 millió felhasználó világszinten, de olyan nagyvállalatok is használják, mint a NASA, Apple vagy a Microsoft [20].

A Vue.js az alábbi tulajdonságokkal rendelkezik:

Gyors - Különféle teljesítménytesztek igazolják, hogy gyorsabb keretrendszer mint a rivális Angular vagy React [21].

Könnyed - Egy Vue.js-ben fejlesztett alkalmazás az úgynevezett *tree-shaking*nek köszönhetően nem tartalmaz olyan beépített könyvtárakat, amiket a fejlesztés során nem használnak, ezzel csökkentve az alkalmazás végső méretét [22].

Skálázható - A keretrendszer eszköztára lehetővé teszi nagyobb alkalmazások fejlesztését is, melyet a nagy fokú modularitásának köszönhet [23].

Gyorsan tanulható - A részletes dokumentáció, az aktív közösség és az alapos oktató anyagoknak köszönhetően gyorsan elsajátítható a Vue.js használata, melyet a fejlesztés során is megtapasztalhattunk.

Ezeken a tulajdonságokon felül nagy előnye a keretrendszernek, hogy egy komponenshez tartozó forráskódok egyetlen jól strukturált fájlban megtalálhatók. Ez az úgy nevezett *Single-file Component* tulajdonság. A következőkben a keretrendszer dokumentációja [24] és saját fejlesztési tapasztalataink alapján ismertetjük annak részletesebb tulajdonságait. Egy fájl tartalma a következőképpen alakulhat:

Sablon - Először a komponens vázát tartalmazó sablont kell megadnunk, ez minden komponens kötelező eleme. Egy HTML alapú sablon szintaxisban adjuk meg a komponensünk vázát, amelyet az alkalmazás futása során fogunk adatokkal megtölteni. Ezt a sablont a *<template></template>* HTML címkék között tudjuk megadni.

Szkript - A komponens dinamikusságáért felel, az abban megjelenő adatot szolgáltatja a sablon számára. Itt végezhetünk adatlekérést, kezelhetjük a komponenshez tartozó interakciók mögötti műveleteket. Megadhatjuk a komponens által felhasznált más komponenseket, változókat, de azt is meg tudjuk pontosan határozni, hogy az adott komponens bizonyos életciklusaiban milyen műveletek hajtódjanak végre. A kódokat a már jól ismert, de opcionális *<script></script>* HTML címkék között tudjuk elhelyezni.

Stílus - A komponensekhez természetesen stílust is rendelhetünk, a Cascading Style Sheet (CSS) stílusleíró nyelv segítségével. Arról is dönthetünk, hogy az itt megírt stílusdefiníciók az egész alkalmazásra legyenek érvényesek, vagy csak az adott komponensre a *scoped* kulcsszó megadásával. Webfejlesztői tapasztalataink szerint ez egy igen erős eszköz. A különféle keretrendszerek nélküli fejlesztés során a stílusok nyomonkövetése egy növekvő alkalmazásban egyre nehézkesebb, nem fenntartható kódot tud eredményezni, mely problémát ezzel az apró megoldással könnyen meg lehet előzni. A stílusdefiníciókat az opcionális *<style></style>* HTML címkék között tudjuk megadni.

A **sablon** és a **stílus** kapcsolata a HTML és CSS kapcsolatából kézenfekvő, egy alap weblaphoz hasonlóan itt is attribútumokat adhatunk a HTML elemekhez,

melyekhez általunk definiált stílus szabályok tartoznak. Az adatot szolgáltató **Szkript** rész kapcsolata a **sablon**nal már kicsit bonyolultabb, de egy egyszerű példán keresztül könnyen át lehet látni a működést.

1.3.3. A Vue.js működés közben

Mielőtt létrehoznánk első komponensünket, először inicializálnunk kell egy Vue.js-nek megfelelő könyvtárstruktúrát. A fejlett eszközök segítségével ez egy nagyon egyszerű feladat. Először szükségünk lesz a Node.js JavaScript futtató környezet legalább 16.0-s verziójára. Amint ez rendelkezésre áll, a környezethez tartozó **npm** csomagkezelő segítségével egy parancsban tudjuk telepíteni a függőségeket és inicializálni a példa projektünket:

```
npm init vue@latest
```

Itt meg kell adnunk a készítendő projekt néhány alaptulajdonságát. Majd belépve az újonnan létrehozott projekt könyvtárába már létre is tudunk hozni komponenseket. Nézzük meg egy alapvető komponens forráskódját tartalmazó fájlt az *PageTitle.vue*-t:

```
<template>
    <h1>{{ title }}</h1>
</template>

<script>
export default {
    props: ['title']
}
</script>

<style scoped>
h1 {
    margin-top: 50px;
    margin-bottom: 50px;
    text-align: center;
}
</style>
```

Ez egy olyan újra felhasználható cím komponensnek a forráskódja, amelyben a címben megjelenő szöveg paraméterezhető. A fájl elején található a **sablon**. Ebben található egy *h1* HTML elem. Az elem szövege egy dupla kapcsos

zárójelben, egy úgynevezett *mustache* sablonban helyezkedik el. Az ott található *title* kulcsszó egy változó, melynek értéke a **szkript** részben kerül majd meghatározásra. A dupla kapcsos zárójelben egyébként bármilyen érvényes JavaScript kód elhelyezhető, ami további komplex, de mégis egyszerűen érthető megoldásokat tesz lehetővé.

A **szkript** részben találhatjuk a komponens számára elérhető változókat függvényeket. Ebben a példakódban egyetlen fajta változót láthatunk. A *props* listában olyan változó neveket sorolunk fel, amelyeknek értékét a komponens más komponensekben való használatakor kell megadni. Például ezt a példa komponenst szeretnénk egy oldal komponensében felhasználni, hogy ott egy címet jelenítsen meg, az általunk választott szöveggel. Ezt a következő féle képpen tudjuk megtenni a *PageComponent.vue* fájl sablon részében:

```
<template>
    <PageTitle title="Példa cím"></PageTitle>
</template>
```

Ekkor a *PageTitle* komponens beágyazódik a *PageComponent* oldal komponensbe, és a hozzá kapcsolt **title** változó értéke "*Példa cím*", ez a cím fog megjelenni a kirenderelt oldalon. Így egy komponens több oldalon is felhasználható, csupán a **title** változó értékét kell módosítanunk.

A **stílus** rész pedig egyértelműen megadja, hogy milyen stílusú legyen a *h1* elemünk. Figyeljük meg a *<style></style>* címkében szereplő *scoped* kulcsszót, mely a már említett módon segít a komponensek stílusainak elkülönítésében.

Az elkészült oldal komponenst importálnunk kell az alkalmazásunk belépési pontjául szolgáló **App.vue** fájlba, melyet JavaScript segítségével fel kell csatolunk egy HTML fájlban lévő elemre. Ezt követően a következő parancsok segítségével tudjuk elindítani az alkalmazásunkat, mely alapértelmezés szerint egy Vite [**?**] által lokálisan futtatott szerveren teszi elérhetővé alkalmazásunkat:

```
npm install
npm run dev
```

Ennek további hatalmas előnye, hogy így nem kell minden kód változtatás után újraindítanunk a szervert, hanem az automatikusan képes felismerni és megjeleníteni a változásokat, és a fejlesztőnek már csak az alkalmazása további kialakításával kell foglalkoznia.

1.4. Spring Boot

Ahhoz, hogy a dolgozatunk PureScript nyelven írt backend részét igazán értékelni tudjuk, szükségünk volt egy másik nyelven írt implementációra, amely

jó összehasonlítási alapot nyújthat számunkra. Választásunk azért esett a Spring Bootra, mert ezt egyetemi tanulmányaink során már több ízben is megismerhettük, illetve szakmai tapasztalatainkon keresztül is találkoztunk ezzel a teljesen más nyelven írt eszközzel. Ebben az alfejezetben ezt szeretnénk bemutatni.

1.4.1. Java

A Spring alapja az 1995-ben megjelent, jelenleg az Oracle vállalat kezelésében működő ojektum orientált programozási nyelv és platform, a Java [26]. Az alkotójának, James Goslingnak eredeti célkitűzése az "írd meg egyszer, futtasd bárhol" volt [6], ami a hordozhatóság tulajdonságában meg is valósult. Működése a következő képpen alakul [27]:

1. A fejlesztő megírja a forráskódot a *.java* kiterjesztésű szöveges fájlba.
2. A *javac* fordító a megírt kódból *.class* kiterjesztésű bájtkódokat tartalmazó fájlokat készít
3. E bájtkódokat fogja a Java Virtual Machine (JVM) a számítógép számára érthető utasításokká formálni, ami így végrehajtja a megírt programot.

A nyelv fontosabb tulajdonságai a következők [6]:

Hordozhatóság - A már említett JVM adja a nyelv hordozhatóságát, mely segítségével bármilyen számítógépen lehetséges Java kód futtatása. Mára a Java támogatása szinte magától értetődő és nem csak a fő operációs rendszerek támogatják, de böngészők, telefonok és IoT eszközök is. Így már szinte szó szerint értelmezhető a fenti Gosling idézetben szereplő "bárhol".

Biztonság - Ez fő szempont volt a nyelv kialakításának kezdeteitől fogva. A Java programok egy alaposan testreszabható jogosultságokkal rendelkező izolált környezetben futnak, ami lehetővé teszi nem megbízható forrásból származó forráskódok, csomagok használatát, anélkül, hogy az kárt okozna a számítógépünkben.

Dinamikusság és kiterjeszthetőség - A Java kódok osztályokba szerveződnek, melyeket dinamikusan képes betölteni a fordító, akkor amikor azokra szükség van.

A nyelv jelentősége tagadhatatlan. A TIOBE indexen az előkelő harmadik helyen szerepel [9], és bár népszerűsége az utóbbi években valamelyest csökkent, biztosan sokáig népszerű lesz még a már meglévő, ebben a nyelvben írt szoftverek és rendszerek miatt. A népszerűség egyik fontos hozadéka a nyelvvel foglalkozó aktív közösség, mely nagyban hozzájárul a nyelv elsajátításának megkönnyítéséhez.

Amennyire népszerű a programozással megismerkedők körében könnyen tanulhatósága miatt, egy-egy alkalmazás fejlesztése gyakran túlságosan bonyolulttá tud válni. E bonyolultság enyhítésére a Java nyelvhez is készültek keretrendszerek. Egy ilyen keretrendszer a Spring, dolgozatunk második backend implementációjának alapja.

1.4.2. Spring, a Java keretrendszer

A Spring egy 2003-ban megjelent keretrendszer Java alkalmazások fejlesztésének támogatásához. Széleskörű eszköztárral rendelkezik, hogy skálázható, nagy teljesítményű alkalmazásokat lehessen általa létre hozni. A keretrendszer legfőbb tulajdonsága a modularitás. Az alap modulon túl a fejlesztő kezében van a döntés, hogy az adott alkalmazásba mely számára szükséges modulokat építi be. A főbb modulok a következő területeket érintik [7]:

- adatelérés és kezelés,
- web,
- biztonság,
- tesztelés.

A keretrendszer alapelve a függőség befecskendezés vagy angolul Dependency Injection, melynek lényege, hogy a függőségek futási időben kerülnek átadásra. Ez két alapvető Java koncepció felhasználásával került implementálásra: az interfészek és a JavaBean-ek. Az interfészek és a függőség befecskendezésének kölcsönösen hasznos együttműködésének eredménye a rugalmas, de nem túl bonyolult alkalmazások. A függőség befecskendezés előnyei [7]:

> **Csökkent kódméret** - A függőség befecskendezés következtében sokkal kevesebb kódra van szükség ahhoz, hogy alkalmazásunk különböző részeit összeillesszük.
>
> **Alkalmazás konfigurációk egyszerűsödése** - A konfigurációk - hasonlóan a Spring alapgondolatához - modulárisan működnek, így ha egy részét le

szeretnénk cserélni az alkalmazásunknak, azt nagyon könnyen meg tudjuk tenni.

Tesztelhetőség fejlődése - Az előbb említett modularitás hozadéka a tesztelésben is tetten érhető. A tesztelési folyamat során a különböző objektumok mock-olása ennek következtében nagyban leegyszerűsödik.

1.4.3. A Spring Boot projekt

A Spring keretrendszer egy elterjedt modulja a Spring Boot. Erőssége, hogy a korábban említett konfigurációk terhének tetemes részét leveszi a fejlesztők válláról. Az alkalmazásfejlesztésben fontos szerepe van annak is, hogy a Spring boot rendelkezik beépített web szerver támogatással, mint a Tomcat vagy a Jetty, amely lehetővé teszi a Java alkalmazásokból készített JAR csomagok könnyed kihelyezését. Így a fejlesztőnek egy külön álló szerver felállításával sem kell törődnie. Alap csomagokkal is rendelkezik a keretrendszer, amelyek általános felhasználási eseteket fednek le, így egy alap váz létrehozása az alkalmazásunkhoz igazán egyszerű feladat.

Egy Spring Boot alkalmazás felépítése rétegekből áll, mely tartalmazza a következő fontosabb rétegeket:

Modell - Az alkalmazásban használt entitások modelljeit tartalmazza.

Perzisztencia - Az entitások és az adatbázis közötti kapcsolatot hozza létre.

Szolgáltatás - Az entitásokkal kapcsolatos műveleteket adja meg a Perzisztencia réteg felhasználásával.

Kontroller - Az alkalmazás végpontjaihoz rendeli a Szolgáltatás réteg műveleteit.

A Spring Boottal való fejlesztés egyszerűségének ékes példája a Spring által biztosított inicializáló eszköz, a Spring Initializr [28]. Ebben az eszközben magunk konfigurálhatjuk a készítendő projektünket a projekt menedzsment eszköz, a Spring Boot verzió, a Java verzió vagy az egyéb függőségek kiválasztásával. Ezt követően legenerálódik a projekt a paraméterek alapján, és kezdődhet is a fejlesztés.

E tulajdonságaival rengeteg időt spórol meg a fejlesztők számára, de természetesen ennek ára is. Mivel a lehető legtöbb mindent a fejlesztő helyett kíván megoldani, így a korábban felsorolt előnyöket kevésbé lesz képes kihasználni

a fejlesztő, amennyiben egy komplexebb alkalmazást kíván fejleszteni, ezért ilyenkor gyakran kell manuális konfigurációkra támaszkodni az alapértelmezettek helyett.

A hátránya ellenére is hatalmas népszerűségnek örvend, ma is gyakori felhasználási területe a REST API-k fejlesztése, ahogy ez a mi esetünkben is így történt. A saját fejlesztési tapasztalatokon keresztül pedig a fent említett előnyökre és hátrányokra is igazolást fogunk látni.

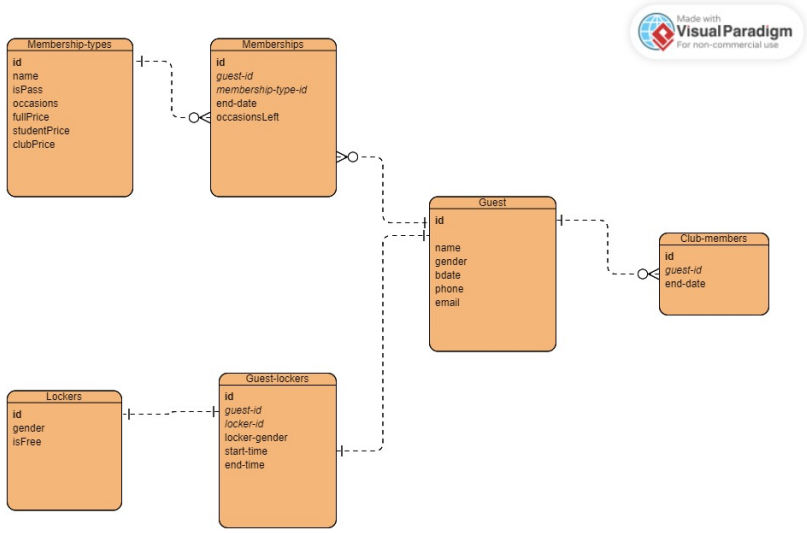
2. A fejlesztés bemutatása

A fejlesztés során szem előtt tartottuk azt az elvet, hogy a két alkalmazás majd nemcsak hogy ugyanazt a funkcionalitást lássa el, de lehetőleg (már amennyire a nyelvi sajátosságok engedik) ugyanúgy is épüljön fel, ugyanúgy viselkedjen. Mindkét alkalmazás ugyanazt a MySQL adatbázist és ugyanazt a frontendet használja.

Az első lépés az üzleti logika átgondolása, a funkcionális tervezés volt. Mivel a szoftver nem egy valós üzleti igényt elégít ki, hanem inkább kísérleti jellegű (Proof of Concept), így ez a szakasz kisebb hangsúlyt kapott.

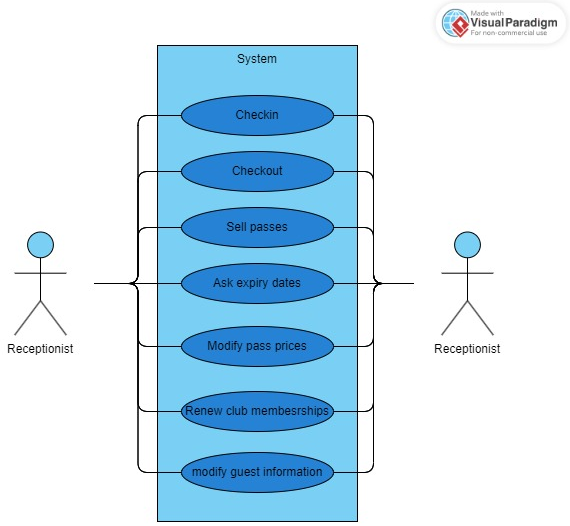
Az adatbázis megtervezésével kezdtük. A MySQL adatbáziskezelőt választottuk, mivel nyílt forráskodú, a tervezett funkcionalitást bőven ellátja és könnyen lehet használni. Elkészítettük a sémát, mely az első ábrán látható. Tartalmazza a táblák nevét, az attribútumokat, félkövér kiemeléssel az elsődleges kulcsot. A táblák közötti nyilak a külső kulcsokat és kapcsolattípusokat mutatják.

A *Vendégek* tábla tartalmazza egy vendég adatait stb.... ez kell?



1. ábra. A szoftver adatbázis modellje.

A Vue.js-ben implementált frontend ugyanazon a porton (3000, természetesen nem egyszerre) tud kommunikálni akár a PureScript, akár a Spring Boot REST API-val. Ez HTTP kérések használatával történik: mindkét backend ugyanazokra a kérésekre ugyanazt a választ és ugyanazt a működést eredményezi. Az első táblázatban néhány kérés látható a vendégekkel kapcsolatban. A szerver



2. ábra. Használati eset (use case) diagram.

erre a kérésre küld egy választ, mely ha szükséges, tartalmazhat a body-ban egy JSON-t.

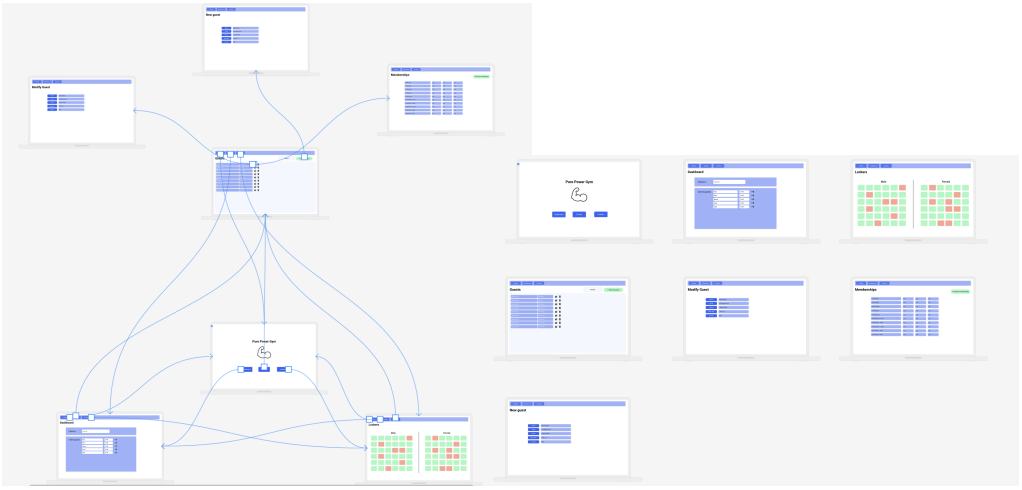
Az Uizard[**?**] nevű szoftverrel elkészítettük az oldalak kinézetének terveit, illetve az oldalak közötti navigációt modellező prototípust. Ez nagyon hasznos lehet a korai tesztelés szempontjából, ugyanis egy ilyen modell rövid idő alatt elkészíthető, így nagyon hamar egy kattintgatható prototípuson lehet kipróbálni, hogy az elképzeléseknek eleget tesz-e a szoftver. A harmadik ábrán az így elkészült navigációs térkép és mellette néhány képernyőkép terv látható.

A funkcionális tervezés részeként továbbá készítettünk egy UML használati eset diagramot, mely a második ábrán látható. Célja a szereplők és az általuk végezhető műveletek bemutatása. Az első két ábrát a Visual Pradigm szoftverrel készítettük.

A tervezés után a két backend implementálása következett, melyet a következőkben ismertetünk.

Metódus	**Útvonal**	**Leírás**
GET	/guest/getAll	Az összes vendég lekérése.
GET	/guest/getById/{id}	Egy vendég lekérése az azonosítója alapján.
POST	/guest/insertGuest	Egy új vendég hozzáadása.
PUT	/guest/updateGuest/{id}	Egy vendég kartonjának szerkesztése
DELETE	/guest/deleteGuest/{id}	Egy vendég törlése
...	...	...

1. táblázat. A kérések listájának részlete.



3. ábra. Az oldalak közötti navigáció és az oldalak tervezése.

2.1. PureScript

Az alkalmazás fejlesztéséhez első lépésként telepítettük a szükséges szoftvereket az npm csomagkezelő segítségével.

```
npm install -g purescript
npm install -g spago
```

Az első sor maga a PureScript, a második pedig Spago telepítése A *-g* kapcsoló a globális telepítés miatt szükséges. A Spago egyrészt a csomagokat, függőségeket is kezeli, illetve a szoftver életciklusát is menedzseli, például az *install, build, run* parancsok segítségével. A Spagoban előredefiniált könyvtárak importálása egyszerű, és ezek a könyvtárak naprakészek, jól karban vannak tartva. Hátránya, hogy nem feltétlen minden PureScript könyvtár van itt, amire szükség lenne.

Fejlesztő környezetként a Visual Studio Code-ot választottuk, mely rendelkezik PureScript kiterjesztésekkel és kiemeléssel is. A fejlesztés során ez egy jó döntésnek bizonyult.

Ezután jó darabig ismerkedtünk magával a nyelvvel, mivel valamennyi funkcionális tapasztalattal mindketten rendelkeztünk korábban, de nagyrészt imperatív nyelveket használunk. Ehhez jó segítség volt a PSCI, amit a *spago repl* paranccsal lehet elindítani. Lehetőség van egy alap kód megadására, ami minden indításnál lefut, ide helyeztük a szükséges importokat. Ezután soronként futtattuk a programot, példáuk gyakran lekértük az elemek típusát a *:type* paranccsal.

```
> x = 5
> :type x
Int
```

A szoftver alapvetően az MVC (model - nézet - vezérlő) architektúrális mintát követi. A modell leképezésével kezdtük, ehhez az első ábrán bemutatott adatbázis modellt vettük alapon, a PureScript kódban ez szolgált a típusok vázaként. Minden típushoz egy *.purs* fájlt készítettünk, melyben magának a típusnak és a hozzá tartozó függvényeknek a leírása található. A következő módon készítettünk egy ilyen típust:

```
newtype Guest = Guest
    { id   :: Int
    , name :: String
    , gender :: String
```

```
    , bdate :: String
    , phone :: String
    , email :: String
    }
```

Ahhoz, hogy egy Guest típusú értéket kiírassunk, szükség van a *Show* beépített típusosztály példányosítására, mely egy adott típusból egy sztringet készít.

```
instance showGuest :: Show Guest where
    show = genericShow
```

Továbbá szükség van ehhez a ReadForeign és WriteForeign típusosztályok felüldefiniálására:

```
derive instance genericGuest :: Generic Guest _
derive newtype instance readForeignGuest :: ReadForeign Guest
derive newtype instance writeForeignGuest :: WriteForeign Guest
```

Egy Guest típusú elemet a következőképp hozhatunk ezután létre repl-ben:

```
g = Guest {id:1, name: "Doma" gender:"male", bdate:"2005.04.26"
           , phone:"+36401234567", email:"doma@mail.com"}
```

Majd kiíratjuk:

```
Show g
```

Következő lépés a getter és setter függvények elkészítése volt minden típushoz:

```
getGuestId :: Guest -> Int
getGuestId (Guest guest) = guest.id

setGuestId :: Int -> Guest -> Guest
setGuestId s (Guest guest) = Guest {id:s, name:guest.name, gender:guest.gender, bo
```

Ez manuálisan kicsit hosszadalmas folyamat ahhoz képest, hogy más nyelvekben gyakran lehetőség van őket automatikusan generálni (például a Lombok szoftverrel).

A main függvényünk visszatérési értéke *Effect Unit*, ami az Effekt monádban való egységet jelenti. A számítások alapból az Effekt monádban futnak, de lehetőség van az *Aff* aszinkron monád használatára is.

```
main :: Effect Unit
main = do
  shutdown <- serve port router $ log $ "Server up running on port: " <> show port
  let shutdownServer = do
        log "Shutting down server..."
        shutdown $ log "Server shutdown."
  onSignal SIGINT shutdownServer
  onSignal SIGTERM shutdownServer
```

A do kulcsszóval lehetőség van egy olyan blokk létrehozására, melynek a visszatérési értéke unit, vagyis nem egy konkét érték, hanem egy cselekvés. Ezzel technikailag van visszatérési értékünk. A main annyit csinál, hogy a HTTPure csomagból származó *serve* függvény segítségével elindítja a szervert, bizonyos szignálok esetén pedig leállítja. A portot 3000-re állítottuk, és implementáltuk a router függvényt, mely gyakorlatilag a szoftver szíve.

Az adatbázissal való kapcsolat adatait egy purs fájlban tároltuk. A típusok adatbázisba való írása és kiolvasása végett szükségünk volt egy perzisztencia réteg kialakítására. Minden típushoz létrehoztunk a fájlrendszerben egy olyan purs fájlt, mely a perzsztálásra szolgáló függvényeket tartalmazza az adott típusra vonatkozóan. Ilyen függvények tipikusan a CRUD (create, read, update, delete - létrehozás, olvasás, módosítás, törlés) műveleteket implementáló függvények, illetve speciálisabbak, például az aktív felhasználók vagy az érvényes bérletek lekérdezése.

```
deleteMembershipQuery :: String
deleteMembershipQuery = "delete from membership where guestId = ?"

deleteMembership :: Int -> Connection -> Aff Unit
deleteMembership guestId conn = execute deleteMembershipQuery
                            [toQueryValue guestId] conn
```

A fenti példa egy bérlet törlését mutatja. A *deleteMembershipQuery-ben* megadjuk a végrehajtandó SQL szkriptet. A *deleteMembership* függvény egy Int és egy Connection típusú értékből Aff Unit-ot állít elő, mely aszinkron számítások végrehajtását jelenti. A *MySQL.Connection* könyvtár *execute* függvénye hajtja végre majd az adott műveletet. Szüksége van bemenetként az SQL szkriptre sztring formában, melybe a benne szereplő kérdőjelek helyére illeszti a következő tömb bemenet elemeit. Ezen felül szüksége van az adott kapcsolatra is.

A vezérlés megvalósításáért a *Server.purs*-ben található *router* függvény felel. Gyakorlatilag ebben van definiálva a szoftver működése.

```
router :: Request -> ResponseM
```

Egy kérést kap bemenetként, melyre előállítja a megfelelő választ. Az"M" betű a monádra utal, ugyanis a *ResponseM* típus egy HTTPure monádot (Aff) becsomagolva egy választ tartalmaz. A router működésére a legegyszerűbb példa:

```
router { method: Get, path: [ "guest", "getAll" ] } = do
  pool <- liftEffect $ createPool connectionInfo defaultPoolInfo
  guests <- flip withPool pool \conn -> getGuests conn
  liftEffect $ closePool pool
  ok' corsHeader $ writeJSON guests
```

A kérés típusát és az útvonalat definiáljuk a Request típusban. Az útvonalat tömbként kell megadni, melynek elemeire később ellenőrzés végett a következőképpen tudunk hivatkozni:

```
path !@ 0 == "guest"
```

A do kulcsszóval nyitunk egy blokkot, melyben létrehozzuk a kapcsolatot az adatbázissal. A *guests* tömbbe lekérjük a vendégek listáját. Ehhez egy lambda függvényen belül hívjuk a *getGuests* getter függvényt. Lezárjuk a kapcsolatot, majd a guest tömböt JSON formátummá alakítjuk, ez kerül a válaszba.

Komplikáltabb függvények esetén vizsgáljuk a paramétereket és esetleg a törzsben kapott JSON-t. Ezeknek szerinte a megfelelő hibával térünk vissza.

```
let lastParam = path !! 2
case lastParam of
  Nothing -> badRequest' corsHeader $ wrapMessageinJSON
               "Missing id parameter"
  Just idParam -> do
```

A fenti kódcsipetben az útvonal első elemét vizsgáljuk. Amennyiben hiányzik, erről hibaüzenetben tájékoztatjuk a frontendet. Ellenkező esetben a do utáni blokk kerül végrehajtásra. A *lastParam* visszatérési értéke *Maybe*, ami egy speciális kezelést igénylő típus. A *Maybe*, tulajdonsága, hogy például egy *Maybe Guest* típus lehet semmi, vagy *Guest*. Külön kezelést igényel, hogy az adott típus üres, vagy sem. Ezt a fenti példában nagyon jól ki lehet használni annak ellenőrzésére, hogy hiányzik-e a paraméter.

2.2. Java - Spring Boot backend

Ebben az alfejezetben az összehasonlítás alapjául szolgáló Java Spring Boot REST API fejlesztését fogjuk bemutatni. Fontos megjegyezni, hogy ha csupán a Spring segítségével szerettük volna elkészíteni a REST API-nkat, akkor valószínüleg nem mindenhol a most implementált megoldásokat alkalmaztuk volna. Mindent olyan módszerrel kívántunk megoldani, ami a legjobban hasonlít a PureScriptben használtakra, mindezt azért, hogy a dolgozatban szereplő összevetésben szereplő különbségek inkább a nyelvbeli különbségekből fakadjanak és ne a megoldások különbségéből. Ez növeli az összehasonlítás értékét, eredményének jelentőségét. Ennek tudatában vegyük végig a fejlesztés lépéseit.

2.2.1. A tényleges implementáció

A Java projekt első lépéseként a már említett Spring Initializr eszközt használtuk fel. Ennek három fontos részét kell kiemeljük:

Projekt menedzsment eszköz: Maven - Mivel mindkettőnk ezt az eszközt használta korábbi Java fejlesztéseink esetén, ezért ezt részesítettük előnyben.

Java verzió - 8 - Azért választottuk ezt a verziót, mert korábban ezzel már fejlesztettünk, illetve a magasabb verziók által kínált nyelvi eszközöket sem kívántunk igénybe venni.

Függőségek - Szükségünk volt a Spring Boot által biztosított kezdő csomagok közül a *web*-hez kapcsolódóra, az adatbázis eléréshez a *Java Persistence API*-ra (*JPA*), a *MySQL* adatbázishoz kapcsolódáshoz egy *kapcsoló* csomagra és a *Lombok Projekt* csomagjára, melynek a fejlesztési menetre való pozitív hatásáról említést fogunk tenni a későbbiekben.

Ezt követően rendelkezésünkre is állt a projekt váza. A következő lépésben kihasználva a Spring konfigurációinak erősségét, elvégezzük a szükséges konfigurálásokat. Meghatározzuk, hogy a HTTP szerver saját számítógépünk melyik portján legyen elérhető, illetve megadjuk a korábban létrehozott MySQL adatbázisunk elérési adatait és a perzisztálási stratégiát. amely a JPA adatbázisunkhoz való alapvető hozzáállását szabályozza.

Ezután kerültek implementálásra az adatbázisunk tábláiban szereplő entitások modelljei. Meghatároztuk azok attribútumait típusaikkal együtt, mindezeket privát eléréső változók formájában. Az adott entitásokat leképező osztályokhoz

szükséges konstruktorok, illetve a változók elérését és beállítását elvégző függvényeinket a már említett Lombok Projekt segítségével, csupán néhány annotáció felhasználásával meg is tudjuk adni. Így az entitások kódja tömör, kompakt marad. Ilyen annotációk a **@Data** vagy az **@AllArgsConstructor**. A Spring konfigurálhatósága az entitásoknál is tetten érhető, szintén annotációkon keresztül. Az **@Entity** annotáció jelzi, hogy az osztály leképezhető az adatbázis egy táblájára, illetve a változók közül az **@Id** annotáció segítségével adhatjuk meg a leképezett tábla elsődleges kulcsát. Speciális esetben előfordulhat, hogy egy tábla összetett, több mezőből álló elsődleges kulccsal rendelkezik, mely problémát szintén egy egyszerű annotációval, az osztályra helyezett **@IdClass**-szal tudjuk megoldani, így a változók közül már több kulcsot is ki tudunk jelölni. Nézzük meg a **Guest** entitást példaként, hogy milyen tömören és egyszerűen lehet lemodellezni az adatunkat:

```
@Data
@Entity
@AllArgsConstructor
@NoArgsConstructor
public class Guest {

    @Id
    private Long id;
    private String name;
    private String gender;
    private String bdate;
    private String phone;
    private String email;
}
```

Az entitásokat követik a hozzájuk tartozó perzisztencia rétegek, ezt minden entitáshoz szükséges megadnunk. Ezek olyan interfészek, mely a *JpaRepository* osztály terjesztik ki, és az entitás struktúrája alapján alapértelmezett SQL lekérdezéseket biztosítanak számunkra Java függvények formájában. Itt szükséges egy Spring annotáció, a **@Repository** alkalmazása. Természetesen egyéni lekérdezéseket is hozzá tudunk adni, azonban ennek két alapesete van:

> **1. A perzisztencia típusa megegyezik az egyéni lekérdezés eredményének típusával** - Ebben az esetben csak a függvényeket kell deklarálnunk és ellátnunk azokat a **@Query** annotációval, amiben az SQL lekérdezésünket tudjuk elhelyezni. Ezekben természetesen paramétereket is elhelyezhetünk, melyek a későbbiek folyamán töltődnek fel tartalommal.

2. A perzisztencia típusa nem egyezik meg az egyéni lekérdezés eredményének típusával - Ebben az esetben elegáns megoldásként egy úgynevezett csomagoló osztályt hozhatunk létre, amely a lekérdezési eredmény mezőinek megfelelő változókat tartalmazza, majd ezt követően egy konfigurációs annotáció, az **@SqlResultSetMapping** hozzáadásával tudathatjuk a Springgel, hogy miként szeretnénk leképezni a kapott eredményt a csomagoló osztály változóira. Ezen apró módosításokat követően viszont ugyanúgy tudjuk megadni az egyéni lekérdezéseinket, mint az előző esetben.

A következő réteg a szolgáltatás. Itt adhatjuk meg azokat a függvényeket minden egyes entitáshoz külön osztályban, amelyek a különböző URL-ekre való hívásokkor fognak lefutni. Felhasználjuk bennük a perzisztencia réteg biztosította függvényeket és az egyénileg írt SQL lekérdezéseket is az **EntityManager** osztályon segítségével. Ehhez a réteghez is kapcsolódik egy Spring annotáció, a **@Service**. Ebben a rétegben érhetjük tetten a Spring nagy fegyverét, a függőség befecskendezést is. Minden egyes a rétegben használt perzisztencia osztályt definiálnunk kell az **@Autowired** annotációval együtt. Így a függőségek, nevezetesen a perzisztencia osztályok futási időben lesznek elérhetőek a szolgáltatás osztályokban.

Az utolsó, legfelső réteg pedig a kontroller. Itt már több annotációval találkozhatunk, ezeket vizsgáljuk meg egy példán keresztül:

```
@CrossOrigin(origins = "http://localhost:5173")
@RestController
@RequestMapping("/guest")
public class GuestController {

    @Autowired
    GuestService guestService;

    @GetMapping(path = "/getAll")
    List<Guest> getGuests() {
        return guestService.getAllGuest();
    }

    ...
}
```

Az első annotáció a **@CrossOrigin**, melyben megadhatjuk, hogy milyen címekről fogadja a kontroller a kéréseket, más címekről érkező kérések elutasításra

kerülnek. Itt a mi frontend szerverünk címét adtuk meg, ezzel biztosítva annak a HTTP szerverhez való hozzáférést. A **@RestController** annotációval tudatjuk a Springgel, hogy az osztály egy kontroller feladatát hivatott ellátni. A **RequestMapping** annotáció pedig segít nekünk rendszerezni a kontrollerek által adott végpontokat. Itt a vendégekkel kapcsolatos API végpontjaink vannal, ezért az annotációban megadott "*/guest*" biztosítja azt, hogy a felsorolt végpontok ezzel az útvonallal fognak kezdődni a tényleges szerveren. Itt is találkozunk a függőség befecskendezéssel, de itt most a szolgáltatás osztályt szeretnénk befecskendezni, ugyanis az tartalmazza a szükséges függvényeket.

Alatta pedig pegig egy példa végpont látható. Szintén annotáción keresztül tudjuk megadni az elvárt HTTP metódust, illetve az erre a vépontra érvényes útvonalat. Ez a vépont például a szolgáltatás felhasználásával az adatbázisban megtalálható vendégek listáját adja vissza. A korábban említett **@RestController** annotáció azt is biztosítja számunkra, hogy a függvények eredménye képpen kapott objektumok a HTTP válaszban JSON formátumban jelenjenek meg.

Az összes réteget pedg maga az alkalmazás osztály fogja összekötni, mely osztály rendelkezik a **@SpringBootApplication** annotációval, és a *main* függvényben csupán egyetlen függvényhívás található:

```
SpringApplication.run(PuregymBackendApplication.class, args);
```

Az egyetlen hiányzó komponense a HTTP szerverünknek egy alkalmazásszerver. A mi megoldásunkban a *Jetty* alkalmazásszervert használtuk fel, és mivel a Spring rengeteg mindent ad alapértelmezetten, így ezt sem kellett külön konfigurálni vagy létrehozni.

Így már minden készen áll arra, hogy elindítsuk a szervert, melyet egy egyszerű *Maven* paranccsal tehetünk meg:

```
mvn clean install spring-boot:run
```

Így az alkalmazásunk néhány másodperc után már elérhető is az általunk konfigurált porton.

2.3. Vue.js

A most következő alfejezetben végül pedig a frontend alkalmazásunkként szolgáló Vue.js alkalmazás implementációját mutatjuk be. Mivel mindkét korábban ismertetett backend alkalmazásunk a számítógépünk 3000-es portján futtatható, így a Vue.js alkalmazás mindkét implementációval tökéletesen működik. Fontos megemlítenünk, hogy a dolgozatunknak nem az a célkitűzése, hogy egy tökéletes edzőtermi szoftvert hozzunk létre, hanem megvizsgáljuk a

PureScript nyelvű implementációnkat a Spring implementáció tükrében. Így az alkalmazás frontend részének feladata inkább a szemléltetés, mintsem egy létező edzőtermi menedzsment szoftver alternatívája.

2.3.1. A Vue.js implementáció

A Vue.js fejlett keretrendszer lévén szintén biztosít számunkra egy egyszerű módot, hogy a fejlesztés zökkenőmentesen kezdődhessen. A projekt vázát a már korábban említett módon kaptuk meg az előző fejezetben említett függőségek telepítése után:

```
npm init vue@latest
```

Alkalmazásunkban a következő főbb oldalakat szeretnénk létrehozni:

Kezdőlap - Ezzel az oldallal találkozik a felhasználó először, innen lehetősége van továbbnavigálni a többi oldalra.

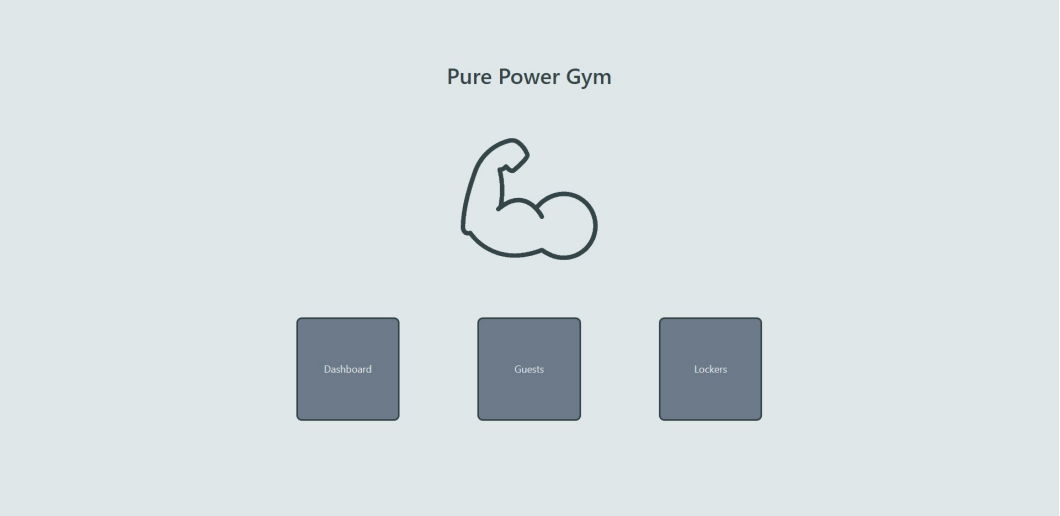
Vezérlőpult - Ezen az oldalon lehet nyomon követni az éppen aktív vendégeket és az aktív bérlettel rendelkező, de nem aktív vendégeket.

Vendégek - Itt láthatjuk az összes vendéget, aki valaha az edzőteremben járt. Lehet kezelni az adataikat, a Klub tagságukat, a bérleteiket és törölhetjük is őket. Ezen felül új vendég regisztrálására is van lehetőség.

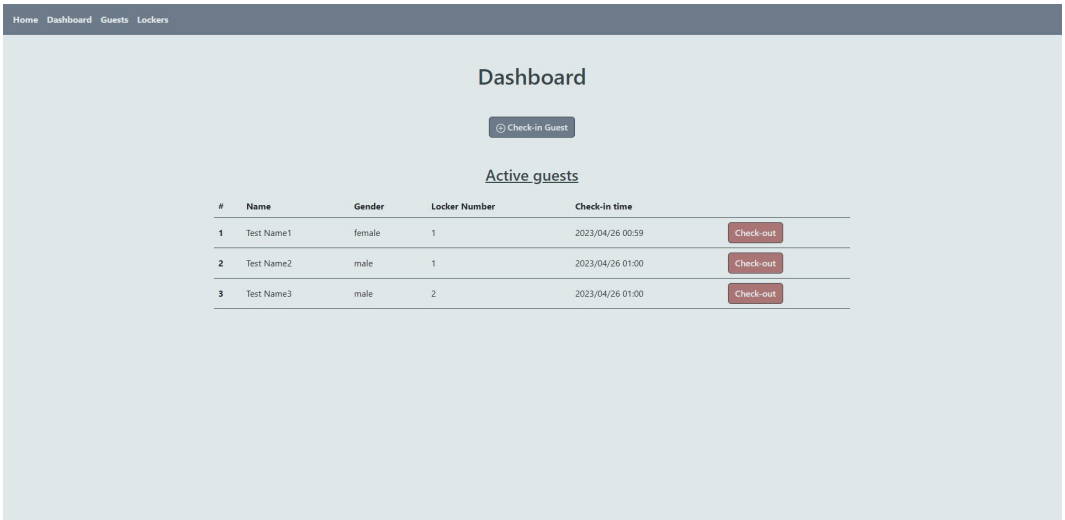
Öltözőszekrények - Végül itt pedig az öltözőszekrények állapotát tudjuk nyomonkövetni.

Ezekhez és az említett funkciókhoz tartozó aloldalak mindegyike és a bennük használt komponensek külön *.vue* kiterjesztéssel rendelkező fájlban került implementálásra. Az alkalmazás bizonyos vizuális elemeinek elkészítéséhez a **Bootstrap** frontend eszköztárat is felhasználtuk. Ez által előre elkészített stílusokkal és elemekkel is tudtunk dolgozni, felgyorsítva ezzel a fejlesztési folyamatot és egyszerűsítve a forráskódot. Vegyük végig a funkciókat és implementációjukat.

A **kezdőlap** csupán navigációs célokat szolgál. Itt találjuk meg a képzeletbeli edzőtermünk logóját, nevét és a navigációs komponenst, melyen keresztül a három másik fő oldal egyikére navigálhatunk. Ahhoz, hogy a navigálás az alkalmazás használata során gördülékeny legyen, minden egyéb oldalon elhelyeztünk egy navigációs sávot. Ez egy komponens formájában került implementálásra, így minden olyan oldalon, ahol szerettük volna felhasználni csak importálnunk kellett a komponens fájlt a *szkript* elemben és ezt a sort kellett megadnunk az adott oldal *sablon* elemében:



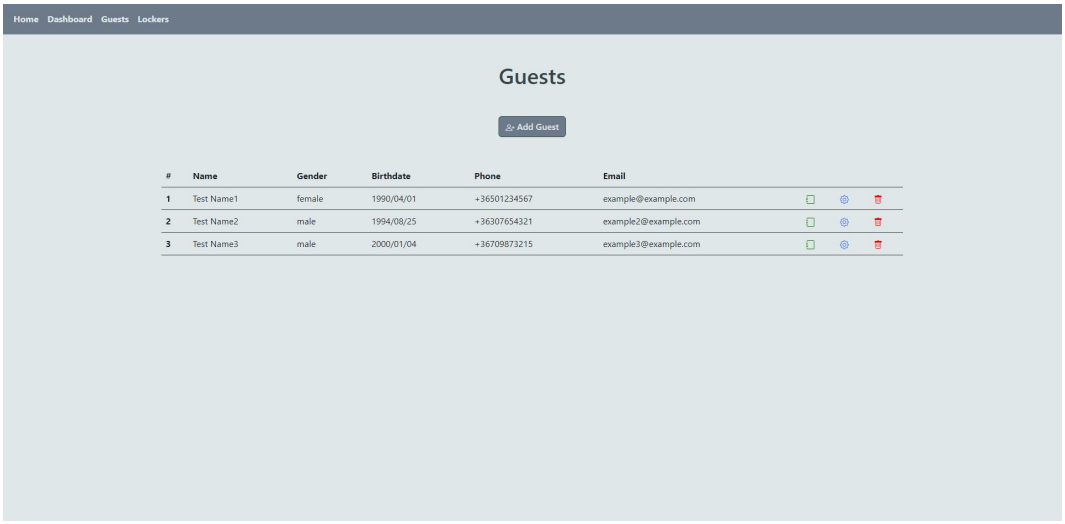
4. ábra. A kezdőlap.



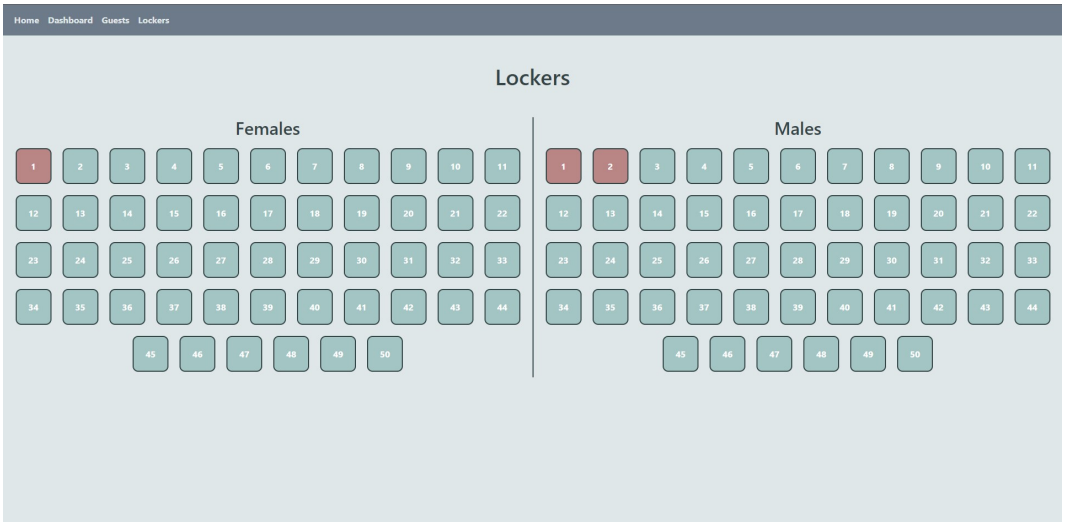
5. ábra. A vezérlőpult.

`<HeaderNav></HeaderNav>`

A következő oldal a **vezérlőpult**. Itt listába gyűjtve látjuk az edzőteremben tartózkodó vendégeinket, mellettük a hozzájuk rendelt öltözőszekrény számmal és a bent tartózkodásuk kezdetének idejével. Itt egyesével lehet őket kijelentkeztetni, ami a valós világ azon eseményének felel meg, amikor a vendég befejezte a tevékenységét az edzőteremben és távozni szeretne. Ekkor nem csak a



6. ábra. A vendégek kezelőfelülete.



7. ábra. Az öltözőszekrények státusz oldala.

vendég státusza változik, de az általa elfoglalt öltözőszekrény is felszabadul. Lehetőség van még vendégek bejelentkeztetésére is. Az ehhez tartozó gombra kattintva a felugró ablakban listázódnak ki azok a vendégek, akik aktív bérlettel rendelkeznek. A bejelentkezést követően megkapják öltözőszekrény számukat és engedélyezett számukra a belépés. Ez a háttérben a következő folyamat szerint történik:

- 1. Kérünk a rendszertől egy a vendég nemének megfelelő szabad öltözőszekrény számot.
- 2. Aktívra állítjuk a vendég státuszát és foglaltra állítjuk az öltözőszekrény állapotát.
- 3. Amennyiben a vendék alkalom alapú bérlettel rendelkezik, csökkentük a hátralévő alkalmainak számát.

A **vendégek kezelőfelülete** sok funkcióval rendelkezik. Az alap oldalon találhatjuk a regisztrált vendégeket, az adataikkal együtt. Minden vendéghez 3 akció tartozik. A vendég adatainak módosításához egy űrlapot tartalmazó oldal nyílik meg, ahol a kívánt adat szerkesztése után el tudjuk menteni az új állapotot. Megtekinthetjük a vendég bérleti és klub tagsági állapotát is. Ha a vendég rendelkezik ezekkel, úgy azok lejárati dátuma szerepel a megnyíló oldalon, illetve alkalom alapú bérlet esetén a hátralévő alkalmak száma. Amennyiben viszont a vendég nem rendelkezik ezekkel, úgy egy-egy gomb jelenik meg, ahol hozzá tudjuk adni ezeket a vendég profiljához. A bérlet esetén egy felugró ablakból választhatjuk ki, hogy milyen típust szeretnénk, a hozzá tartozó árlista pedig a klub tagság függvényében dinamikusan változik. A **vendégek kezelőfelületén** az utolsó adott vendéghez tartozó funkció a törlés, amely bár megítélésünk szerint egy ritkán használt funkció lenne valós esetben, azonban előfordulhat, hogy a vendég szeretné az adatait törölni. Ekkor egy megerősítést követően nem csak a vendég adatlapja, de tagsági története és aktivitási története is törlésre kerül. Az oldalon található még egy funkció, amely vendégek regisztrálására szolgál. Ezt egy üres űrlap kitöltését követően követően tudjuk megtenni. A regisztrációt követően egyből kezelhető az új vendég bérlet és klub tagság állapota.

Végül az **öltözőszekrények** kezelőfelületén láthatjuk nemek szerinti bontásban az öltöző szekrények állapotát. A szabad szekrények zöld, míg a foglalt szekrények piros színnel jelennek meg. Minden egyes szekrényt lehetőségünk van kiválasztani. A felugró oldalon megtekinthetjük az adott szekrény aktuális állapotát és a korábban a szekrényt elfoglaló vendégeket is. Minden egyes bejegyzésből a hozzá tartozó vendég adatlapjára navigálhatunk. Ennek a funkciónak például talált tárgyak esetén tudja az edzőterem hasznát venni, mivel így visszakövethető, hogy mely vendég használta a szekrényt az eseményt megelőző napokban.

Azt, hogy ezekben a megoldásokban, legyenek azok oldalak vagy komponensek, hogyan kötöttük össze a szkript réteg változóit a sablon réteg HTML elemeivel, ezzel dinamikussá téve a weblap tartalmát már a Vue.js-t ismertető fejezetben

láthattuk. Azonban arról, hogy miként kötöttük össze a frontend alkalmazásunkat a HTTP REST szerverünnkel még nem esett szó. A HTTP kérésekhez a Fetch API-t használtuk [34]. Ezzel minden szükséges HTTP metódust tudunk kezelni. A szükséges REST API hívásokat kezelő aszinkron függvények implementálása a JavaScriptben megszokott módon történik. Egy rövid példán keresztül nézzük meg, hogy milyen egyszerűen meg lehet ezt a feladatot oldani. A példánkban azt a függvényt vizsgáljuk, amelyik a regisztrált vendégek listájához szükséges adatokat tölti be:

```
async getData() {
    const res = await fetch("http://127.0.0.1:3000/guest/getAll");
    const finalRes = await res.json();
    this.guests = finalRes;
}
```

A komponens változója, a **guests** tartalma a függvény hívása után a vendégek adatainak listája lesz.

Ami érdekessé teszi az adatelérést az az, hogy lehetőségünk van irányítani, hogy az oldal kirenderelésének mely szakaszában szeretnénk lekérni az adatokat. Ennek a felhasználó szempontjából van jelentősége. Amennyiben túl későn kérjük le az adatot, az a felhasználó felület ugrálását eredményezi, ugyanis az adatok később jelennek meg mint az oldalunk. E probléma megoldására a Vue.js kínál lehetőségeket. Ez a lehetőség az Vue életciklusok használata. Ahhoz, hogy megértsük a sorrendiséget a különböző életciklusok között, említést kell tennünk a Vue.js által biztosított API-okról. A fejlesztés során két API használatára van lehetőség: **Composition API** és **Options API**. A Composition API-ban egyetlen hívásban van lehetőségünk megadni és egyénileg tagolni a *szkript* elemet, míg az Options API-ban különböző opciók szerint kell tagoljuk azt, mint például adatok, metódusok, külső változók [37]. A dokumentáció szerint az Options API kezdő barátabb, ezért a fejlesztés során ezzel dolgoztunk. A hivatalos oldalon lévő diagram szerint [38] az inicializálás első szakasza a következő:

- 1. A Composition API inicializálása - **setup** életciklus
- 2. **beforeCreate** életciklus
- 3. Options API inicializálása
- 4. **created** életciklus

Ennél a pontnál a diagram nem áll meg, de számunkra már látszik az az életciklus amire szükségünk van, ez pedig a **created**. Ez az a leghamarabbi pillanat az oldal generálása során, amikor már működik az általunk választott API. Így a komponensekhez tartozó változók hamarabb kapnak értéket, mint hogy az oldal betöltődne a felhasználó számára. Ezt a mechanizmust nagyon egyszerűen ki tudjuk használni a szkript rétegben:

```
created() {
    this.getData()
}
```

A *created* opcióban tudjuk megadni azokat a függvényeket, amelyeket szeretnénk az adott életciklusban végrehajtani.

Az implementált oldalak összekötéséhez a keretrendszer egy újabb eszközét használjuk fel, a **vue-router** nevű útválasztó csomagot. Itt a korábban bemutatott RestController-hez hasonlóan tudjuk meghatározni, hogy milyen útvonalakon legyenek elérhetőek az oldalaink. Egy útvonalhoz egy oldal komponensét tudjuk csatolni. E mögött az a gondolat található, hogy az útvonalválasztó egy nézettel rendelkezik és ebbe a nézetbe az útvonalak táblázata alapján tölti be a megfelelő komponenst. Ezzel egyetlen útválasztó lesz a közös pontja a felépített oldalainknak.

Minden Vue.js alkalmazásnak szüksége van egy központi, magát az alkalmazást megtestesítő komponensre, aminek *de facto* elnevezése *App.vue*. Ebben a mi esetünkben csak az útválasztó komponenst kell megadnunk. Az egyetlen hátralévő feladat pedig e komponens felcsatolása egy létező HTML elemre, az *index.html* fájlban, JavaScript segítségével.

3. Az eredmények ismertetése

3.1. A fejlesztés tapasztalatai

A fejlesztés eredménye 3 alkalmazás. Két REST API implementáció, az egyik PureScript míg a másik Java Spring Boot felhasználásával. Illetve egy frontend alkalmazás, mely a REST szerverekre támaszkodva egy edzőterem kezelő alkalmazást valósít meg. E nyelvekben való tudásaink nagyban eltértek, így a fejlesztési fázisokban nyelvenként különböző élményekben, tapasztalásokban volt részünk. Ebben a fejezetben a fejlesztési folyamat nehézségeiről és az ezeknek köszönhetően megszerzett hasznos tudásról szeretnénk röviden értekezni.

3.1.1. Fejlesztés funkcionális nyelven

Egyetemi tanulmányai során nagyon keveset talákozik a funkcionális nyelvvel a hallgató. Mindkettőnknek volt azonban olyan kötelezően választható tárgya az egyetemen, melynek tematikájában a funkcionális nyelvek is szerepeltek. Azonban ezen a tárgyon természetesen nem volt idő olyan mélységében elsajátítani egy funkcionális nyelvet, hogy azt követően könnyedén képesek legyünk egy komplett alkalamzást fejleszteni. Inkább a funkcionális nyelvek gondolkodásmódjának megértése, világának megismerése volt a feladat. Így az ezen e nyelven történő REST API implementáció izgalmas kihívást nyújtott számunkra. Mindkettőnk objektum orientált nyelvekkel dolgozott eddig, így egyik nagy feladatunk volt, hogy megértsük és elsajátítsuk a nyelvet legalább olyan szinten, hogy egy ilyen komplex célt tudjunk megvalósítani.

A PureScript nagyon fiatal nyelvnek számít. A nyelv iránt rajongó, azt aktívan fejlesztő közösség mérete jóval elmarad mondjuk a Java nyelv mögötti közösséghez képest. Ennek több olyan hozadéka is van, amely két, a nyelv elsajátítását a zászlójára tűző fiatal számára nem túl pozitív. Egyrészt egy adott problémára nem létezik olyan sok megoldást kínáló csomag, legtöbbször egy-két opció adódik. Ráadásul mivel ezeket a csomagokat a fejlesztők igen nagy százaléka szabadidejében, önszorgalomból fejleszti, ezért a csomagok bővülése hosszabb folyamat, sőt vannak olyan csomagok amik egy rövid aktivitás után gyorsan eltűnnek, mert egyszerűen a fejlesztőnek úgy alakul az élete, hogy nem tud tovább foglalkozni annak karbantartásával. Másrészről az elérhető dokumentáció erősen hiányos, a fórumokon kevés problémába és arra adott megoldásba lehet ütközni, így egy nyelvet tanulónak az lehet az érzése, hogy egy teljesen zárt közösségbe szeretne valahogy bejutni.

Például a MySQL kapcsolatos PureScript csomag semmilyen dokumentációval nem szolgál, így az azzal kapcsolatos munka során néhány fórum bejegyzésre és

teszt esetre tudtunk támaszkodni. Az úgynevezett elhagyatott csomagokkal is volt személyes tapasztalatunk. Egy néhány éve elkészített rövid összefoglaló egy adott eszközről már egyáltalán nem aktuális, mert azóta a benne felhasznált csomagokat már nem tartják karban.

Pozitívumot is tudunk azonban kiemelni a csomagokkal kapcsolatban. Az általunk használt *spago* csomagkezelő eszköz alapértelmezés szerint csomag halmazokkal működik. Ezek a halmazok kizárólag karbantartott csomagokat tartalmaznak, számos felhasználási célra. Ráadásul minden csomagnak olyan verziója található meg a halmazban, amelyek egymással kompatibilisek. Így amíg az alapértelmezett csomagokat használjuk, addig könnyen meg tudjuk adni a készülendő projektunk függőségeit. Munkánk során mi is ezekre a csomagokra támaszkodtunk.

Persze ahogy arra számítottunk is, a legtöbb időt a PureScript implementáció emésztette fel. Ennek oka nem kizárólag az említett dokumentáció hiány, vagy kis méretű közösség, hanem a funkcionális nyelvekben való járatlanságunk is.

Olyan esettel is találkoztunk a fejlesztés során, hogy bizonyos funkciók nincsenek olyan kézenfekvően megoldva PureScriptben, mint mondjuk a Spring Bootban. Az API fejlesztés során nem ismeretlen fogalom a CORS, azaz a Cross Origin Resource Sharing. Ez leegyszerűsítve azért felel, hogy a szerver meg tudja határozni, hogy a beérkező kéréseket milyen címről fogajda el, azok milyen HTPP metódusokat használhatnak és a kérések fejléc mezőit is megszabhatják. Ez a Spring Bootban végtelenül egyszerűen, annotációkkal szabályozható. Viszont PureScriptben ezekről magunknak kellett gondoskodni. Egyrészt elő kellett készítenünk egy úgynevezett *preflight* kérést, ami egy *OPTIONS* metódusú ellenőrző kérdés, ami arra szolgál, hogy tájékoztassa a szerver a klienst az érvényes elérési szabályokról, másrészt minden egyes válaszhoz egy szabáyloknak megfelelő fejlécmezőt kellett csatoljunk.

3.1.2. Fejlesztés Java nyelven

A Java nyelv viszont már egy nagyon jól ismert terület volt számunkra. Nem csak az egyetemi tanulmányok során, de a mindennapi munkában is rendszeresen foglalkozunk vele. Ráadásul egy kötelező egyetemi tárgy keretében az alapképzésen még a Spring keretrendszert is megismertük, így a Spring Boot alapú REST API implementálása jóval könnyebb feladatnak bizonyult a PureScript nyelvű társánál.

Nem meglepő módon az elérhető dokumentációval, a fejlesztő közösséggel és a megoldások sokszínűségével nem volt probléma a Java esetében. A népszerűségének köszönhetően nem találkoztunk olyan problémával, amire

ne lett volna korábban példa, így megoldás is. Sőt az esetek nagy részében több megoldási lehetőség adódott, így magunk választhattuk ki a számunkra megfelelőt.

A fejlesztés során azonban egy érdekes problémával találkoztunk, amivel kapcsolatban egyikünknek sem volt még tapasztalata. Az API-nak létezik egy olyan végpontja, amelyben az URL-ben megadandó paraméter tartalmaz '/' karaktereket, azonban a kontrollerünk nem tudta feldolgozni az ilyen kéréseket, annak ellenére, hogy karakterkódolást is alkalmaztunk. Sok egyéni URL feldolgozó megoldást kipróbáltunk, mielőtt megtaláltuk a probléma igazi forrását. Ez nem a Spring kontrollerben volt keresendő, hanem az automatikusan a Spring által konfigurált *Tomcat* alkalamzásszerverünkben. Ugyanis a Tomcat az ilyen jellegű kéréseket nem tudta kezelni. Ezért írtuk felül az alapértelmezett szervert, és cseréltük le a Tomcatet *Jetty*-re, mely meg is oldotta a problémánkat. Ezt a megoldást az aktív Java és Spring fejlesztő közösségnek köszönhetően találtuk meg.

3.1.3. Frontend Vue.js-ben

A Frontend fejlesztés területén is közös elősimereteink voltak, mindkettőnk foglalkozott már Angular keretrendszerben frontend alkalmazás implementálásával. Így ha magával a Vue.js-szel még nem találkoztunk, a komponens alapú, hasonló gondolatvilágú frontend fejlesztéssel már igen, mely hozzájárult ahhoz, hogy viszonylag gyorsan megismerkedjünk a keretrendszerrel.

A keretrendszer fiatal kora ellenére hatalmas közösség szerveződött köré, így szerencsére nem küzd a PureScript dokumentációbeli hátrányával. Természetesen abból a szempontból kicsit torz ez az összehasonlítás, hogy a Vue.js nem egy nyelv, hanem egy keretrendszer, és az alapjául szolgáló JavaScript sokkal nagyobb múltra tekint vissza.

3.2. Mérési eredmények

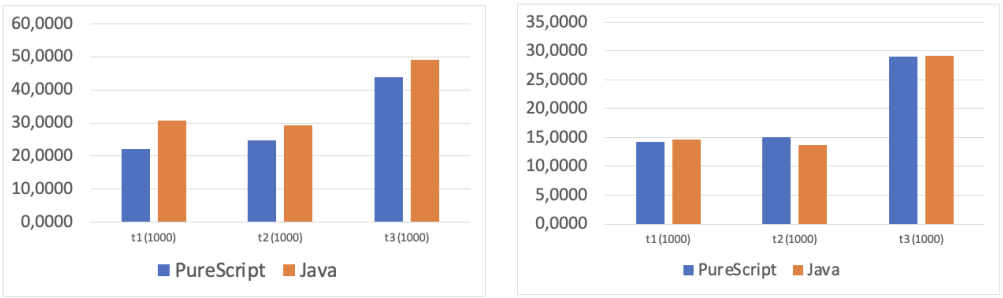
A két szoftver objektív összehasonlítására teszteseteket készítettünk Pythonban. A három különböző tesztesetben különböző kéréssorozatokat küldünk a szervernek feldolgozásra, és mérjük a közben eltelt időt. Például a hármas számú tesztesetben létrehozunk egy vendéget, vásárolunk neki tagságot, beléptetjük , kiléptetjük majd töröljük a vendégek közül. A tesztesetek összeállításánál arra ügyeltünk, hogy a teszt olyan állapotban hagyja az adatbázist, amilyenben találta. Ennek köszönhetően több konfigurációban is zökkenőmentesen tudtuk a teszteket egymás után futtatni.

A következő kódcsipet segítségével kommunikálunk a szerverrel:

```
url = "http://127.0.0.1:3000/guest/insertGuest"
payload = "{\"id\":1 , \"name\": \"Abel\", \"gender\": ...
response = requests.request("POST", url, headers=headers,
           data=payload)
```

Megadjuk az URL címet, útvonalat, opcionálisan a rakományt és a HTTP kérés típusát a *requests* Python könyvtárból származó függvénynek, mely a HTTP választ eredményezi.

A frontend alkalmazás implementálása és a tesztek elvégzése közben egy számunkra új ismeretet gyűjtöttünk a lokális fejlesztéssel kapcsolatban. Mivel tesztjeinket nem csak Windows operációs rendszeren végeztük, hanem MacOS operációs rendszeren is, arra lettünk figyelmesek, hogy a Windows-os mért értékek szignifikánsan nagyobbak MacOS-es társaiknál (nagyjából két nagyságrend különbség mutatkozott). Ez kérdéseket vetett fel bennünk, ugyanis a két számítógép számításteljesítményének különbsége nem indokolt ekkora eltéréseket. A probléma forrása egy egyszerű Windows mechanizmus volt. Eredetileg a frontend alkalmazásunkban lévő HTTP hívásokhoz és a tesztekhez is *http://localhost:3000* címet használtuk. Általános esetben ugyanazt jelenti Windows-os rendszereken, mintha a *localhost* név helyett a számítógép lokális IP címét, a *127.0.0.1*-et írnánk. Azonban van egy igen lényeges különbség a kettő között. Mégpedig az, hogy a *localhost* nevet a rendszernek még fel kell oldania, mielőtt a kérést elküldené. A MacOS rendszeren ez a mechanizmus ettől eltérően működik, így nem okozott problémát. Miután a címeket kicseréltük az IP címet tartalmazókra, utána már nagyságrendileg hasonló eredményeink voltak mind a két típusú operációs rendszer esetében.

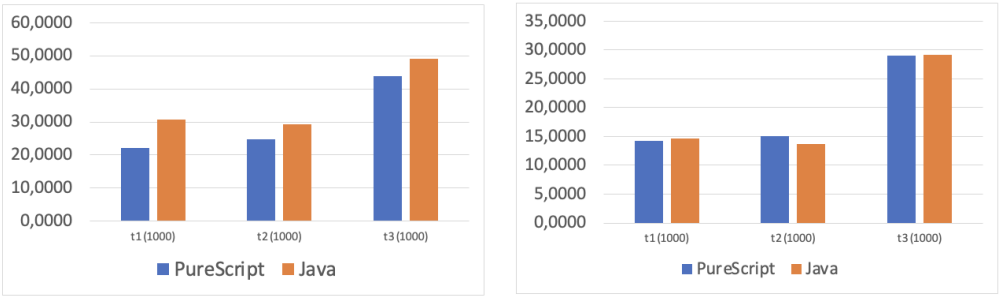


8. ábra. A tesztek eredménye másodpercben MacOS és Windows operációs rendszereken, tízes kötegmérettel.

A teszteseteket tízes, százas és ezres kötegekben futtattuk, minden köteget ötször, és ebből számoltunk átlagot. Azért választottunk több kötegméretet, hogy azt is lássul, hogy egyre több kérés esetén hogy viszonyul a sebesség.

A teszteket lefuttattuk Windows 10 és MacOS Ventura operációs rendszereken is (Intel(R) Core(TM) i5-9400F CPU @ 2,90 GHz és Apple M1 3,2 GHz processzorokkal rendelkező gépeken). Célunk nem a két rendszer sebességének összehasonlítása volt ezzel, csak megerősítés és kíváncsiság az eredmények konzisztenciájára vonatkozóan.

A mi esetünkben az általunk végzett PureScript tesztek átlagosan 13,25 százalékkal teljesítettek jobban. A ? ábrákon tízes és ezres kötegek átlageredményei láthatók másodpercben, bal oldalon MacOS Ventura, jobb oldalon Windows 10 operációs rendszer alatt. A többi diagram megtalálható a függelékben.



9. ábra. A tesztek eredménye másodpercben MacOS és Windows operációs rendszereken, ezres kötegmérettel.

A második táblázatban a következő tendencia figyelhető meg az eredmények között: a kötegméret növekedésével egyértelműen csökken a PureScript előnye. Tízes kötegnél még átlagosan 20,4 százalékkal, míg ezres kötegméretnél már csak 7,8 százalékkal kevesebb időre volt szüksége a PureScript szervernek.

Véleményünk szerint ez annak köszönhető, hogy ???

Ami még érdekes eredmény, hogy MacOS operációs rendszer alatt a sebességkülönbség jóval nagyobb, mint Windows-on: Mac-en 24,15 százalék, míg Windows-on csak 2,36 százalék átlagosan.

Kötegméret	**A PureScript átlagos sebességbeli előnye**
10	20,40%
100	11,57%
1000	7,80%

2. táblázat. Mérési eredmények kötegméret szerint

Összefoglalás

Irodalomjegyzék

[1] C. Scalfani. Functional Programming Made Easier: A Step-by-Step Guide. 2021.

[2] P. Freeman. PureScript by Example. 2014 - 2017.

[3] John McCarthy: The implementation of LISP. 1996. http://www-formal.stanford.edu/jmc/history/lisp/node3.html

[4] Szuromi Zs.: Programozási paradigmák, kézirat. ME, 1996.

[5] Olga Filipova: Learning Vue.js 2, 2016.

[6] David Flanagan: Java in a Nutshell, 2005

[7] Iuliana Cosmina, Rob Harrop, Chris Schaefe, Clarence Ho: Pro Spring 5: An In-Depth Guide to the Spring Framework and Its Tools, 2017

[8] Dr. Vadász Dénes: Programozási paradigmák, programozási nyelvek (letölthető egyetemi oktatási anyag).
Hozzáférés dátuma: 2023.04.11.

[9] The TIOBE Programming Community index.
`https://www.tiobe.com/tiobe-index/`
Hozzáférés dátuma: 2023.04.11.

[10] A Java nyelv hivatalos honlapja.
`https://dev.java/learn/lambdas/`
Hozzáférés dátuma: 2023.04.11.

[11] B. Radojicic.: Imperative to Functional Programming in Java. 2022
`https://symphony.is/blog/imperative-to-functional-programming-in-java`
Hozzáférés dátuma: 2023.04.11.

[12] J. Neumann.: Advantages and disadvantages of functional programming. 2022.
`https://medium.com/twodigits/advantages-and-disadvantages-of-functional\-programming-52a81c8bf446`
Hozzáférés dátuma: 2023.04.12.

[13] `https://www.purescript.org/`

[14] `https://vuejs.org/guide/introduction.html`

[15] J. Sturtz.: Functional Programming in Python: When and How to Use It.
https://realpython.com/python-functional-programming/
Hozzáférés dátuma: 2023.04.20.

[16] D. Cravey.: Functional-Style Programming in C++.
https://learn.microsoft.com/en-us/archive/msdn-magazine/2012/august/c-functional-style-programming-in-c
Hozzáférés dátuma: 2023.04.20.

[17] Kliens-oldali programozási nyelvek használatát bemutató statisztika.
https://w3techs.com/technologies/details/cp-javascript
Hozzáférés dátuma: 2023.04.23.

[18] A Vue.js által támogatott böngészők.
https://vuejs.org/about/faq.html#what-browsers-does-vue-support
Hozzáférés dátuma: 2023.04.24.

[19] Az ES2015-öt támogató böngészők. https://caniuse.com/es6
Hozzáférés dátuma: 2023.04.24.

[20] Vue.js-t használó nagyvállalatok.
https://vuejs.org/about/faq.html#is-vue-reliable
Hozzáférés dátuma: 2023.04.24.

[21] Különféle JavaScript alapú keretrendszerek benchmark eredményei.
https://rawgit.com/krausest/js-framework-benchmark/master/webdriver-ts-results/table.html
Hozzáférés dátuma: 2023.04.24.

[22] A Vue.js könnyed tulajdonságának leírása.
https://vuejs.org/about/faq.html#is-vue-lightweight
Hozzáférés dátuma: 2023.04.24.

[23] A Vue.js skálázhatóságának leírása.
https://vuejs.org/about/faq.html#does-vue-scale
Hozzáférés dátuma: 2023.04.24.

[24] A Vue.js hivatalos dokumentációja, útmutatója.
https://vuejs.org/guide/introduction.html
Hozzáférés dátuma: 2023.04.24.

[25] A Vue.js fejlesztéshez használt Vite szerver honlapja.
https://vitejs.dev/

[26] A Java nyelv rövid története.
https://www.java.com/en/download/help/whatis_java.html
Hozzáférés dátuma: 2023.04.24.

[27] A Java nyelv működése.
https://docs.oracle.com/javase/tutorial/getStarted/intro/definition.html

[28] A Spring Initializr eszköz.
https://start.spring.io/

[29] A PureScript nyelv hivatalos dokumentációja.
https://github.com/purescript/documentation
Hozzáférés dátuma: 2023.04.24.

[30] A PureScript nyelv hivatalos forráskódja.
https://github.com/purescript/purescript
Hozzáférés dátuma: 2023.04.24.

[31] A Spago hivatalos dokumentációja.
https://github.com/purescript/spago
Hozzáférés dátuma: 2023.04.24.

[32] Pursuit. Hivatalos PureScript dokumentáció.
https://pursuit.purescript.org
Hozzáférés dátuma: 2023.04.24.

[33] https://uizard.io

[34] A Fetch API hivatalos, MDN dokumentációja.
https://developer.mozilla.org/en-US/docs/Web/API/Fetch_API
Hozzáférés dátuma: 2023.04.26.

[35] A Vue.js API stílusainak bemutatása.
https://vuejs.org/guide/introduction.html#api-styles
Hozzáférés dátuma: 2023.04.26.

[36] A Vue.js életciklus diagramja.
https://vuejs.org/guide/essentials/lifecycle.html#lifecycle-diagram
Hozzáférés dátuma: 2023.04.26.

[37] A Vue.js API stílusainak bemutatása. https://vuejs.org/guide/introduction.html#api-styles Hozzáférés dátuma: 2023.04.26.

[38] A Vue.js életciklus diagramja. https://vuejs.org/guide/essentials/lifecycle.html#lifecycle-diagram Hozzáférés dátuma: 2023.04.26.

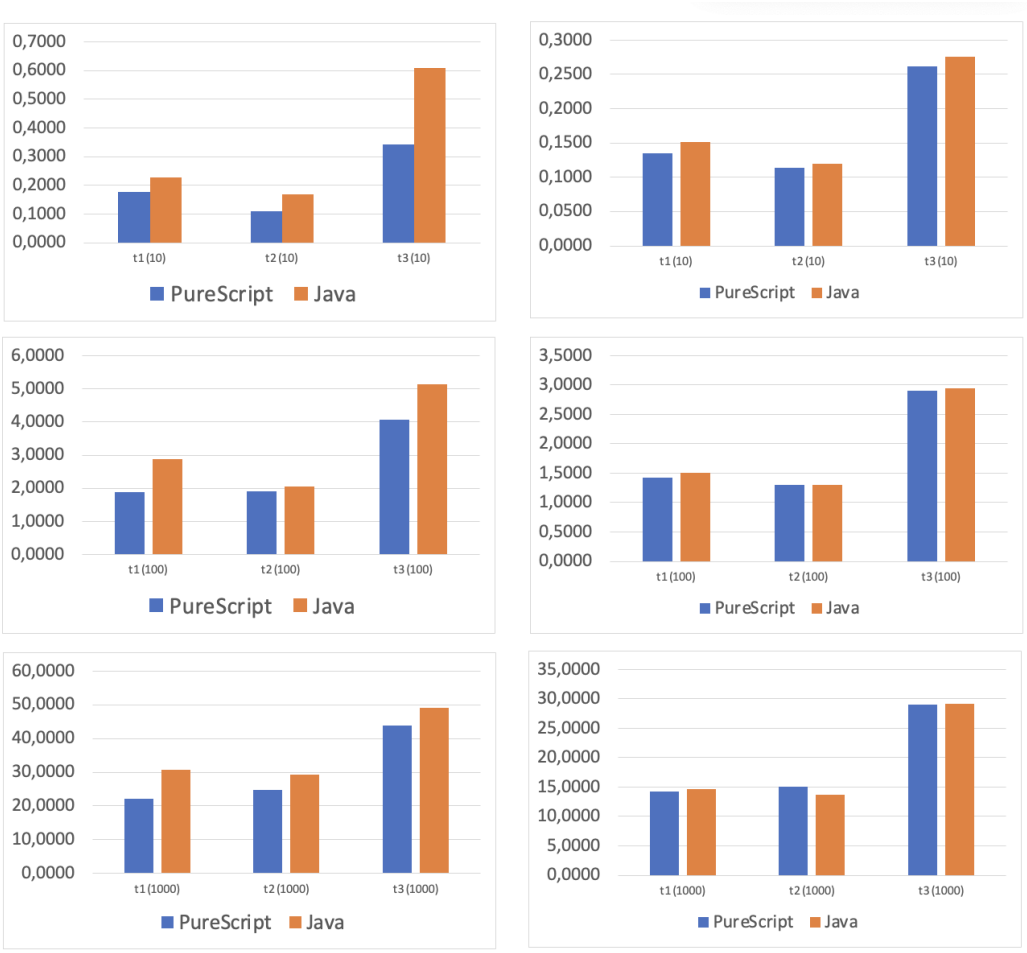
Függelék

Teszt	**Köteg**	**PureScript (mp)**	**Java (mp)**
1	10	0,1344	0,1515
2	10	0,1138	0,1198
3	10	0,2615	0,2754
1	100	1,4171	1,5040
2	100	1,2983	1,2965
3	100	2,9061	2,9475
1	1000	14,2713	14,6063
2	1000	15,0881	13,7380
3	1000	29,0169	29,1394

3. táblázat. Mérési eredmények Windows 10-en.

Teszt	**Köteg**	**PureScript (mp)**	**Java (mp)**
1	10	0,1763	0,2261
2	10	0,1091	0,1683
3	10	0,3421	0,6094
1	100	1,8837	2,8694
2	100	1,8984	2,0486
3	100	4,0703	5,1309
1	1000	22,1610	30,8191
2	1000	24,8790	29,3950
3	1000	43,8844	48,9996

4. táblázat. Mérési eredmények MacOS-en.



10. ábra. A tesztek eredménye másodpercben MacOS (bal oldal) és Windows operációs rendszereken (jobb oldal), különböző kötegméretekkel.