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 [7] [12] [13], 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, 50 és 100-as 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 [4]. 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 [3]. A program függvények hívásából és ezek kiértékeléséből áll, nincs értékadás, csak érték kiszámítás [3]. A függvényt leginkább úgy értjük, hogy egy leképezés egy adott halmazról egy másik halmazra[8]. 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) [3].

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 [6]) alapjaiban imperatív, objektumorientált, de a nyolcas verziótól kezdve [7] megjelennek benne funkcionális

elemek, például a lambda kifejezések. A nyelv hivatalos oldala [7] "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 [12] [13].

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 [8] 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 [9]. 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 [9].

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 [32], 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, tuldajonságait és egy rövid példán keresztül a használatának könnyedségét.

1.3.1. JavaScript keretrendszerek

A Vue.js egy klines-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 [14]. 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. Elenghetetlen továbbá átgondoli 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 nyújtat 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 [22]. 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 [22]. 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 [15], ezzel a felhasználók 96,09%-ához jut el [16]. Megbízhatóságát biztosítja a több mint 1,5 millió felhasználó világszinten, de olyan nagyvállalatok is használják mint a NASA, Apple vagy a Microsoft [17].

A Vue.js az alábbi tulajdonságokkal rendelkezik:

Gyors - Különféle teljesítménytesztek igazolják, hogy gyorsabb keretrendszer mint a rivális Angular vagy React [18].

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 végős méretét [19].

Skálázható - A keretrendszer eszköztára lehetővé teszi nagyobb alkalmazások fejlesztését is, melyet a nagy fokú modularitásának köszönhet [20].

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 struktúrált fájlban megtalálhatók. Ez az úgy nevezett *Single-file Component* tulajdonság. A következőkben a keretrendszer dokumentációja [21] 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 interkaciók 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 újrafelhasználható cím komponensnek a forráskódja, amelyben a címben megjelenő szöveg paraméterezhetőA 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 tapasztalankon 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 ojektumorientált programozási nyelv és platform, a Java [24]. Az alkotójának, James Goslingnak eredeti célkitűzése az "írd meg egyszer, futtasd bárhol" volt [25], ami a hordozhatóság tulajdonságában meg is valósult. Működése a következő képpen alakul [26]:

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 [25]:

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 fentebbi Gosling idézetben szereplő "bárhol".

Biztonság - Ez fő szempont volt a nyelv kialakításának kezdeteitől fogva. A Java prograok 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 [6], 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 alkalamzásokat 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 [27]:

- 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 [27]:

> **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

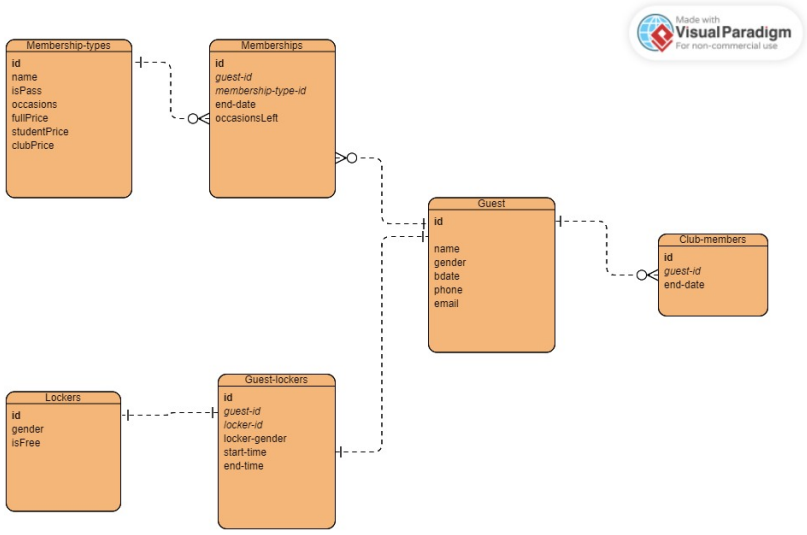
2.1. PureScript

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

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

Az első sor maga a PureScript, a második pedig Spago telepítéséhez szükséges. A Spago egyrészt a csomagokat is kezeli, illetve a szoftver életciklusát is menedzseli, például az *install, build, run* parancsok segítségével. Fejlesztő környezetként a Visual Studio Code-ot választottuk, mely rendelkezik PureScript kiterjesztésekkel és kiemeléssel is.

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.



1. ábra. A szoftver adatbázis modellje.

A következő 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 hatalmas hangsúlyt nem kapott ez a szakasz. Inkább arra figyeltünk, 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.

A szoftver alapvetően az MVC (model - nézet - vezérlő) architektúrális mintát követi. A modell megalkotásával kezdtük, ehhez készítettünk egy adatbázismodellt, mely az első ábrán látható. A PureScript kódban ez szolgált a típusok vázaként is. 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
    }
```

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

```
g = Guest {id:1, name: "Doma" gender:"male", bdate:"2005.04.26", phone:"+364012345

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
```

2.2. Java - Spring backend

2.3. Vue.js

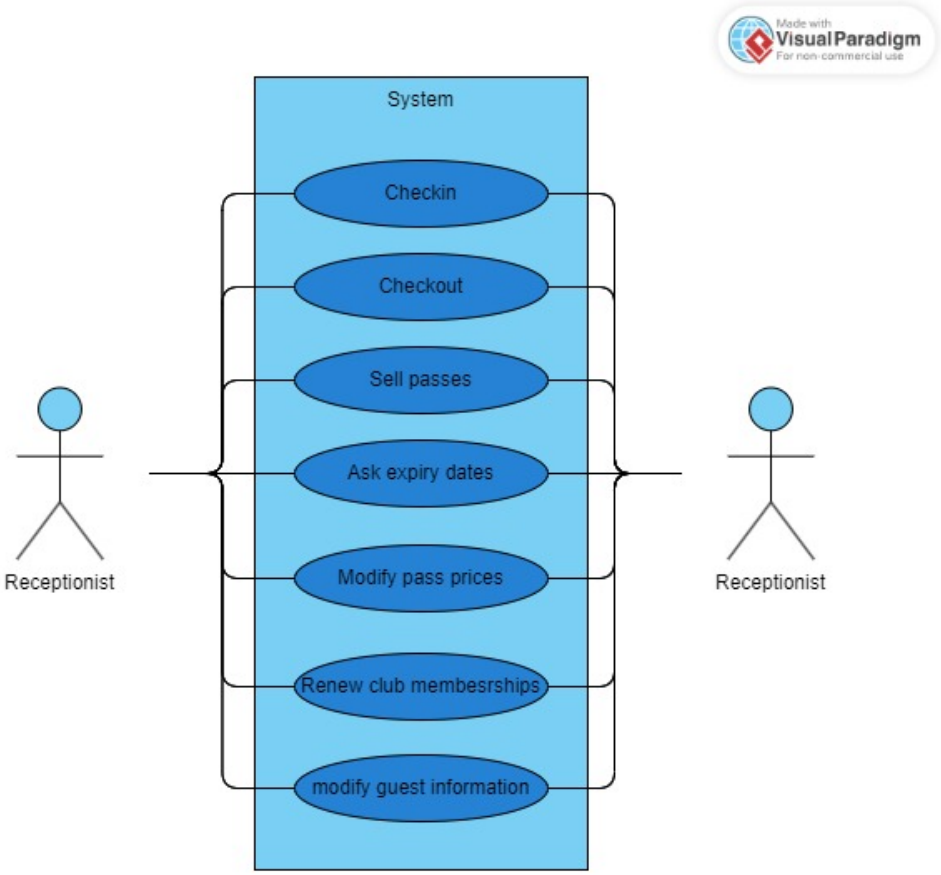
3. Az eredmények ismertetése

3.1. A fejlesztés tapasztalatai

3.2. Mérési eredmények

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.



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

Összefoglalás

Irodalomjegyzék

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

[2] P. Freeman. PureScript by Example. 2014 - 2017. https://book.purescript.org/

[3] Dr. Vadász Dénes: Programozási paradigmák, programozási nyelvek (letölthető egyetemi oktatási anyag) https://web.archive.org/web/20150501083657/http://www.iit.uni-miskolc.hu/iitweb/export/sites/default/users/DVadasz/GEIAL401/Progpar-4-fejezet.pdf# Hozzáférés dátuma: 2023.04.11.

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

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

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

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

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

[9] 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.

[10] https://www.purescript.org/

[11] https://vuejs.org/guide/introduction.html

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

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

D. Cravey.: Functional-Style Programming in C++. Hozzáférés dátuma: 2023.04.20.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

[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.