A megvalósítandó szoftver követelmény leírása

**Áttekintés**

Egy komplex rendszer kifejlesztése hosszú időt, tervezést és odafigyelést igényel, amit befolyásol a projekt mérete és az azon dolgozó csapatok száma. Minél összetettebb a rendszer, minél több fejlesztőcsapat dolgozik a projekten, annál komolyabb munka a csapatok lokális és globális menedzselése. A szoftver minősége és a határidők betartása függ az adott csapat készségétől, a tervezéstől és az előző fejlesztési ciklusok alapján kialakult tanulságoktól, valamint a csapattagok motiváltságától. A szakdolgozat témája egy olyan applikáció elkészítése, amelyben lehetőség nyílik egy agilis módszertan alapján dolgozó fejlesztőcsapat menedzselésére és támogatására, beleértve a munkafolyamatok megtervezésének gördülékennyé tételét, az adott fejlesztési ciklus, azaz ‘sprint’ kiértékelését csapat szinten, ezek visszakereshetőségét és feladatkörökre való osztását, a csapatmorál mérését és metrikák generálását a még hatékonyabb tervezés érdekében, valamint kompetencia mátrix előállítását a csapat tudását illetően.

Cél egy olyan alkalmazás lefejlesztése és prezentálása, ami egy valódi agilis csapatban jelentkező problémák megoldását teszi könnyebbé, gyorsabbá, és átláthatóbbá, a lehető legnagyobb transzparencia mellett. A Célközönség egy agilis módszertan alapján fejlesztő scrum csapat, ami a fejlesztőkből, egy Scrum masterből, és egy product ownerből áll.

Funkcionalitás szempontjából a legfontosabb, hogy átláthatóak és könnyen kezelhetőek legyenek, amiket a módszertan megkövetel, valamint az agilitáshoz megfelelően maximális rugalmasságot tegyen lehetővé. (A fent említett Scrum módszertanok később kerülnek kifejtésre). Az elkészített szoftver egy a böngészőből elérhető webes alkalmazás.

**Miben több ez az alkalmazás a piacon találhatóakhoz képest?**

A világon nagyon sok különböző igényt kielégítő alkalmazás található. Nincs ez máshogy a kis, közép, és nagyvállalati projektmenedzsment terén sem. Több szoftver és szoftvercsomag elérhető, amik segítségével monitorozni lehet egy csapat vagy projekt teljesítményét, esetleg a tervezést segítik, valamint vannak, amik a csapat folyamatos fejlődését segítik elő, vagy metrikák generálását végzik. A szakdolgozat témájában szereplő „Agile Team Management” alkalmazás célja pontosan az, hogy minél több funkció legyen elérhető egyetlen szoftvercsomagban, és ezek ne külön alkalmazások legyenek. Így egy szinergia áll elő, és a csapat teljes egészét egy felületen lehet kezelni. Ez megkönnyíti az átláthatóságot, valamint a csapatra vonatkozó összes adat egy helyen összpontosul. Ez a platform egyesíti a különálló híresebb alkalmazások funkcióit, mint például a csapatra vonatkozó metrikák generálását, a „*retorspective*” féle visszatekintést, valamint a csapattagok közötti kommunikációt „*Message* *Board*”.

**A dolgozat felépítése**

A dolgozat három fő részből áll, bevezető, felhasználói dokumentáció és fejlesztői dokumentáció. A felhasználói dokumentáció a szoftver felhasználó számára történő konfigurálásról szól, valamint az alkalmazás funkcionalitását hivatott bemutatni használat közben. A fejlesztői dokumentáció a tervezéstől az architektúrán át a végleges szoftver megvalósításáig, valamint annak a teszteléséig terjed. A dolgozat végén található egy köszönetnyilvánítás, valamint pár tapasztalatot és véleményt a témával kapcsolatban.

**Rendszer Funkciói**

Az elkészített web alkalmazás egy a legmodernebb szoftvertechnológiák által ötvözött úgynevezett web-stack. A web-stack szoftverek és technológiák kollekciója, amik egy komplex alkalmazás lefejlesztéséhez szükségesek. A Fő alkotó elemük egy Frontend service egy Backend service egy adatbázis motor, ami általában SQL adatbázis, tehát nem relációs adatbázis, ezek általában JSON alapúak. Ezen szakdolgozat témáját alkotó alkalmazás egy a manapság legmodernebb technológiák felhasználásával alkotott full-stack web alapú szoftver, ami a (MySQL, Express.js, React, Node.js)-ből áll.

* MySQL
  + A MySQ egy [többfelhasználós](https://hu.wikipedia.org/w/index.php?title=T%C3%B6bbfelhaszn%C3%A1l%C3%B3s&action=edit&redlink=1), [többszálú](https://hu.wikipedia.org/w/index.php?title=T%C3%B6bbsz%C3%A1l%C3%BAs%C3%A1g&action=edit&redlink=1), [SQL](https://hu.wikipedia.org/wiki/SQL)-alapú [relációs adatbázis-kezelő](https://hu.wikipedia.org/wiki/Rel%C3%A1ci%C3%B3s_adatb%C3%A1zis-kezel%C5%91) [szerver](https://hu.wikipedia.org/wiki/Szerver).
  + <https://hu.wikipedia.org/wiki/MySQL>
* Express.js
  + Az Express egy gyors, rugalmas, de mégis kis web framework, amiben könnyedén lehet robosztus web alkalmazásokat írni Node.js alapokon.
  + <https://expressjs.com/>
* React
  + A React egy komponens alapú javascript frontend könyvtár, felhasználói interfészek megjelenítésére. <https://reactjs.org/>
* Node.js:
  + A **Node.js** egy szoftverrendszer, melyet [skálázható](https://hu.wikipedia.org/wiki/Sk%C3%A1l%C3%A1zhat%C3%B3s%C3%A1g) internetes alkalmazások, mégpedig [webszerverek](https://hu.wikipedia.org/wiki/Webszerver) készítésére hoztak létre. A programok [JavaScript](https://hu.wikipedia.org/wiki/JavaScript)-ben írhatók, [eseményalapú](https://hu.wikipedia.org/wiki/Esem%C3%A9nyalap%C3%BA_szoftverarchitekt%C3%BAra), aszinkron I/O-val a túlterhelés minimalizálására és a [skálázhatóság](https://hu.wikipedia.org/wiki/Sk%C3%A1l%C3%A1zhat%C3%B3s%C3%A1g) maximalizálására.
  + <https://hu.wikipedia.org/wiki/Node.js>

Egyéb felhasznált modulok / fejlesztőeszközök / kód könyvtárak:

Ezeket a fejlesztés során külön kell installálni CLI (command line interface) segítségével, az *npm install* paranccsal. Használva a cli adta *--save* kapcsolót lehetőség van a függőségeket nem globálisan, hanem lokálisan az adott projekt mappájába telepíteni, ekkor a projekt *package.json* fájljába kerül elmentésre a *dependencies* kulcsszó alatt a függőség neve, valamint a projekt mappájában létrejövő *node\_modules* mappába történik az installálás. Az így létrejövő projektet, ha szeretnénk egy másikeszközön használni, akkor lehetőségünk nyílik egy paranccsal feltelepíteni az összes függőséget, feltéve, ha mindegyik szerepel a *package.json* fájlban. Ez esetben a megfelelő helyen állva a projekt mappában, kiadva az *npm install* parancsot az npm (node package manager) felolvassa a *package.json dependencies* alatt lévő listát, és feltelepíti mindegyiket.

Backend:

* cors
  + Middleware kiegészítő Express.js-hez, ami lehetővé teszi a CORS (Cross Origin Resource Sharing) módszert, ezáltal a modulok és rétegek közötti adatfolyamot.
  + <https://www.npmjs.com/package/cors>
* passport
  + Express.js kompatibilis middleware felhasználói autentikációra. Különböző előre definiált *Strategy* található meg benne, amikkel elérhetőek a legnépszerűbb közösségi belépési módszerek, mint a Facebook, Google, vagy az Instagram. JWT ( Json Web Token) alapú, ami azt jelenti, hogy ha a belépési kérés *request* sikeres, akkor a rendszer amit használunk a válaszban egy JWT-t küld *response* benne a felhasználó adataival, és a belépésre való jogosultsággal.
  + <https://www.npmjs.com/package/passport>
* chalk
  + Segítségével tudunk különböző színekkel üzeneteket kiírni a terminál képernyőjére, ezáltal elkülöníteni az eltérő üzeneteket.
  + <https://www.npmjs.com/package/chalk>
* bodyParser
  + Átalakítja a szerver felé beérkező kéréseket.
  + <https://www.npmjs.com/package/body-parser>
* Sequelize
  + Többirányú *Promise* alapú ORM (Object Relational Management) kezelő a legelterjedtebb SQL (Structured Query Language) verziókhoz, mint a Posgres, MySQL, MariDB.
* concurrently
  + Segítségével több CLI parancsot tudunk futtatni egyszerre, egyidőben. A szakdolgozat alapján elkészült alkalmazás esetén egyszerre egy parancs segítségével indul mind a server és a kliens alkalmazás is.
  + <https://www.npmjs.com/package/concurrently>
* mysql2
  + Gyorsaságot előnyben részesítő mysql kliens.
  + <https://www.npmjs.com/package/mysql2>
* nodemon
  + Minden a szerver fájlt ért változtatás után újra indítja a szervert az új frissítésekkel.
  + <https://www.npmjs.com/package/nodemon>

Frontend:

* Material-ui
  + Google által fejlesztett React komponensek gyűjteménye.
  + <https://www.npmjs.com/package/@material-ui/core>
* lodash
  + Gyors és könnyű adatkezelés és adat manipuláció.
  + <https://lodash.com/>
* react-router
  + Útvonal meghatározás az oldalak, és komponensek között frontend oldalon.
  + <https://www.npmjs.com/package/react-router>

**Környezet konfigurálása:**

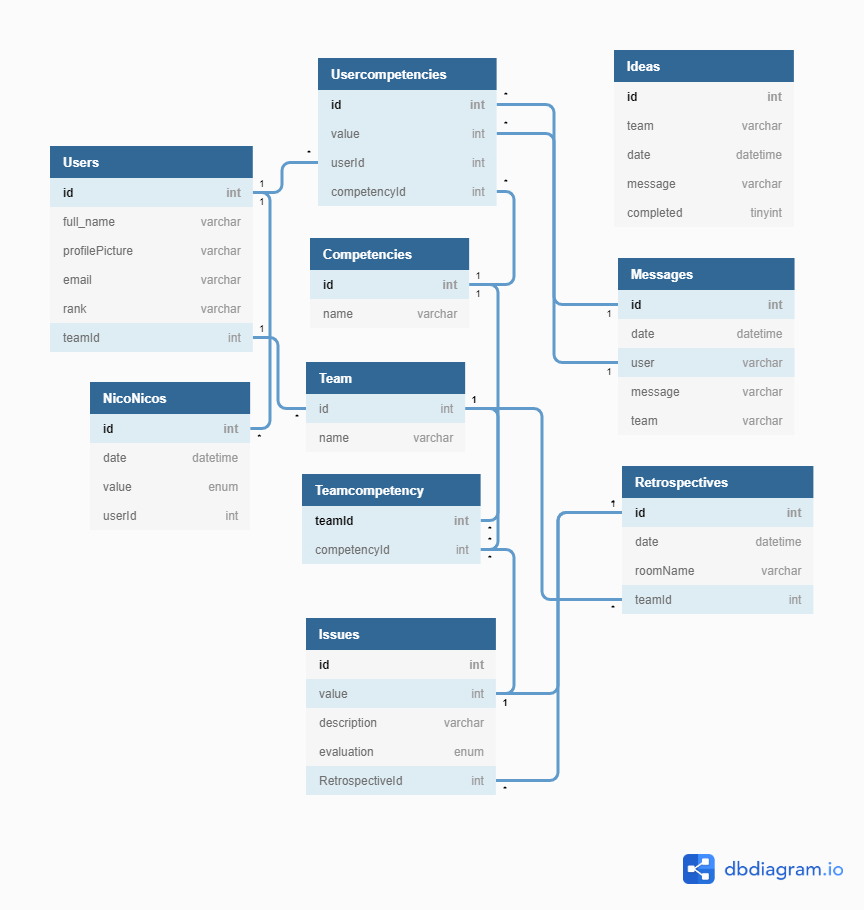
A projekt legelején beállításra kerül, külön a package.json fájlban egy új kulcs érték pár, ami a *’’proxy’’ : ’’http://localhost:5000’’*. Erre azért van szükség, hogy az API hívások esetén a szerver tudja, hogy honnan határozza meg a különböző API kéréseket. Így ha frontend oldalon a *fetch(’/api/team’)* függvény kerül meghívásra, akkor a szerver azt automatikusan átirányítja a *http://localhost:5000/api/team*-re.

A rekordok tárolása MySQL adatbázis szerveren történik. Node.js backend környezetben a Sequelize SQL ORM segítségével vannak a táblák és struktúrák definiálva, ez a *db*/*connections.js* fájlban található. Ebben található egy szülő objektum a *Model*, amiből származik az összes JavaScript osztály, ami az SQL táblák struktúráját írja le. Ezek a projekt elején, elkészülnek, ha még nincsenek definiálva, és SQL oldalon létrejön az adatbázis és a hozzá tartozó táblák, a megfelelő attribútumokkal, kezdőértékekkel.

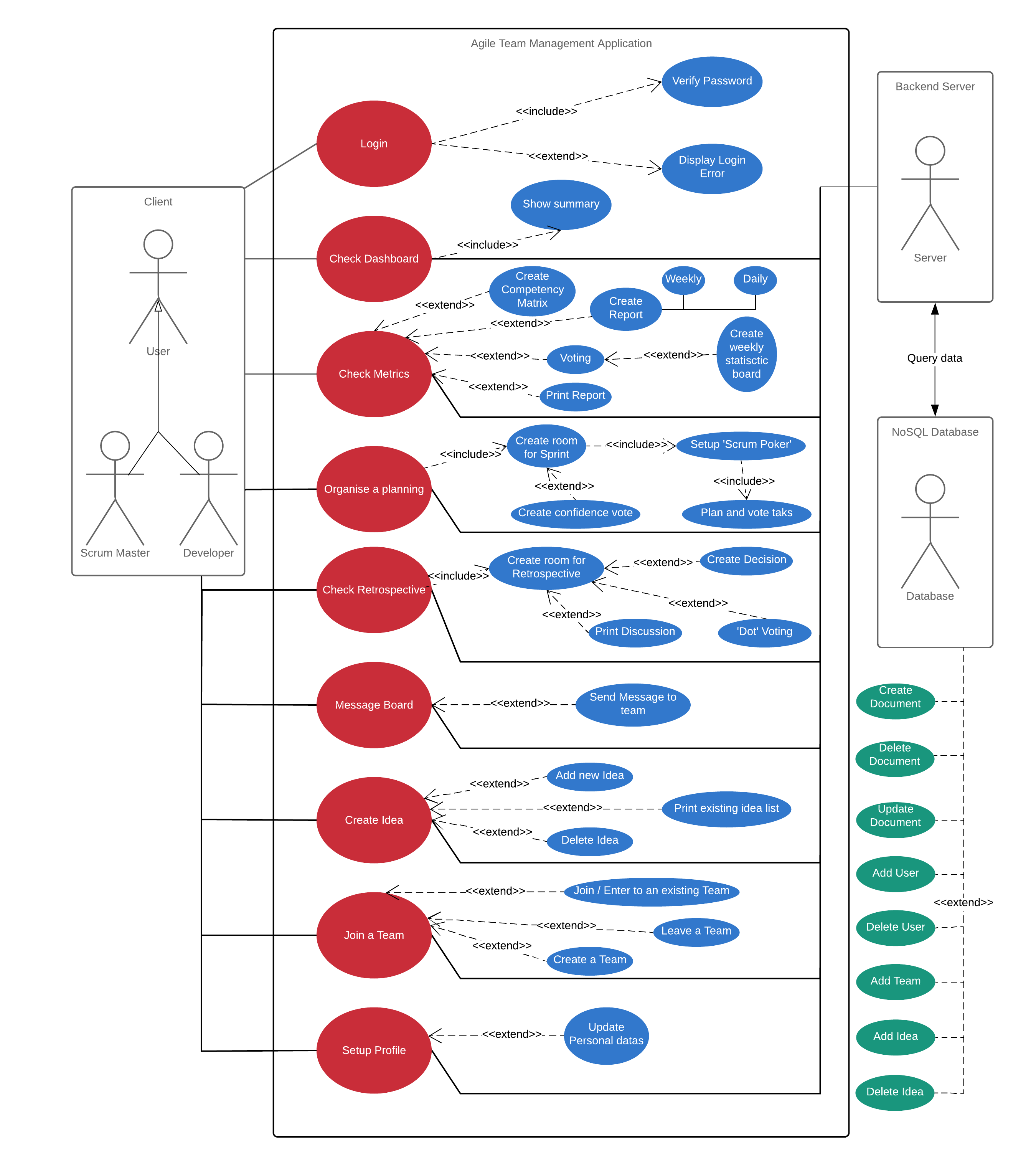
Az elkészült alkalmazás Passport.js-t használ felhasználói autentikációra, valamint verifikációra. Jelen dolgozat tárgyaként szolgáló webes alkalmazás a Passport által elérhető ’stratégiák’ közül a Google közösségi stratégiáját használja. Előkövetelmény, hogy a felhasználóknak legyen Google felhasználói profiljuk, hogy be tudjanak lépni az alkalmazásba.

A környezet beállításához a Google Developer Console felületen regisztrálni kell az elkészített alkalmazást a nevével, és a host címével együtt. Az alkalmazáshoz rendelődik egy egyedi titkosított *ClientId* és egy *ClientSecret*. Erre a regisztrációra azért van szükség, mert így tudja a Google autentikálni az alkalmazást, és a kéréseket kezelni az irányából. Ezután implementálni kell szerver oldalon a *Passport.GoogleStrategy*-t.

Létrehozunk egy új GoogleStrategy objektumot, a clientId, clientsecret és callbackURL kulcsokkal, amiket a Developer Console-n konfiguráltunk be, ezáltal a két platform tud csatlakozni egymáshoz. Sikeres belépés esetén a szerver visszakap egy JSON objektumot, amiben az autentikált felhasználó adatai lesznek benne, amik, a felhasználónév, id, email cím, profilkép url címe… A szerver létrehoz egy új rekordot az ATM adatbázis User táblájában, a GoogleStrategy eredménye alapján. Mivel a *User* táblába *findOrCreate* metódussal lett elmentve a rekord, így ha nem létezik akkor elmentésre kerül, azután a szerveren lévő kezdetben üres *user* objektum feltöltésre kerül az felhasználó adataival, ha létezik a táblában, vagy most lett beregisztrálva. Ez az objektum tartalmazza szerver oldalon a felhasználó adatait, ami elküldésre kerül API-on keresztül a frontend felé.



1. ábra SQL adatbázis szerkezete



Nico-nico:

<https://www.agilealliance.org/glossary/nikoniko/>

Miért érdemes egy Agilis csapatnak Nico-Nico táblát használnia? Ez egy hangulat mutató, ami általában egy sprintet fed le, és mind a csapatra, mind az egyénre lehet hosszú és rövidtávú következtetéseket, metrikákat levonni belőle, valamint lokalizálni az esetleges negatív tényezőket. A Nico-Nico táblázat fejléce a napokat tartalmazza, és ahhoz vannak oszlopban felsorolva a csapat tagjai. Általában három fajta mutatóval dolgoznak, ez három emotikon, vagyis a köznyelvben elterjedt ’Smiley’. Három típusa van:

* Szomorú:



* Neutrális: 
* Boldog:



Ezek által mérhető a csapat motivációja és teljesítménye. A Nico-Nico lényege, hogy ne legyen semmi befolyásoló tényező, ezért az aznapi hangulatot jelző kis emotikont a munka befejeztével kell elvégezni, hogy ez legyen az utolsó teendő a munkahelyről való távozás előtt, így semmilyen új input nem fogja befolyásolni az adott személy hangulatát az aznapi munkanapra vetítve. Scrum esetén a Scrum Master felelőssége ezen táblázat kiértékelése, az esetleges negatív dolgok felkutatása, és feloldása. Jelen táblázat hasznosságát más sem bizonyíthatja jobban, mint a Nico-Nico táblázat összehasonlítása az adott sprint sikerességével, mivel ezen két metrika nagyban korrelál egymáshoz. Egy sikertelen sprint esetén a táblázat több ’szomorú’ fejet fog tartalmazni, és egy sikeresnek mondható sprint esetén több ’boldog’ fej lesz, mivel a tervezés minősége és a munka mibenléte motiválta a csapatot. Ha a táblázatban kezdenek felgyűlni a ’szomorú’ smileyk akkor érdemes lehet a scrum masternek organizálni egy retrospective-t a Product Owner felé, hogy a csapat megvitassa a problémát a jelenlegi sprinttel kapcsolatban. Adott esetben ez a sprint újra tervezését vagy a sprint megszakítását eredményezheti.

Kompetencia mátrix:

<http://infografika.ofi.hu/Tud%C3%A1st%C3%A1r/A-diagramok-t%C3%ADpusai/61-sugardiagram>

Egy agilis csapatban szükség van kompetencia mátrix elkészítésére, ami vizualizálja a csapat tudását, a különböző tudásanyagok, technológiák lefedettségi szintjét, ezzel analizálva, és izolálva a hiányzó kompetenciákat a jövőbeni fejlődést elősegítve, ezáltal a csapat hatékonyságát növelve.

A kompetencia mátrix egy sugárdiagram, ami egy itt jön az idézet: *’ a* [*diagram*](http://infografika.ofi.hu/component/rafcloud/1/diagram?searchphrase=exact&ordering=newest) *középpontjából induló és a külső diagramgyűrűn végződő elkülönült tengely mentén ábrázolja az egyes kategóriák értékeit. Az adatsoroknak egy középponthoz, illetve egymáshoz viszonyított változásait vagy gyakoriságát szemlélteti. Minden kategória saját értéktengellyel rendelkezik, amelyek a középpontból sugárirányban ágaznak ki’.*

Jelen szakdolgozat középpontjában álló webes alkalmazás által használt kompetencia mátrix elemei dinamikusan konfigurálhatók a felhasználók által. Ezek programozási nyelvek, valamint technológiák. Ezeket a rendszer az ATM.Competencies táblában tárolja az adatbázisban. Ezek a rekordok törölhetők, valamint megváltoztathatók.

**Fejlesztői dokumentáció**

**Fejlesztői környezet**

A fejlesztés és a verifikáció Microsoft Windows 10 operációs rendszer alatt történt. A forráskód megírásához Microsoft Visual Studio Code-ot használtam. Az elindított szoftvert kliensoldalon Google Chrome web böngészőn teszteltem, de bármelyik mai modern böngésző tökéletes szoftver futtatására, és kipróbálására. Választásom azért a Chrome-ra esett, mert elérhető hozzá egy hivatalos fejlesztői kiegészítő, ami segít „debuggolni” a React alkalmazásokat frontend oldalon. Az adatbázis létrehozása, és a táblák struktúrájának leírása szerveroldalon történik, viszont teszteléshez, és fejlesztéshez a MySQL WorkBench, jelen szakdolgozat írásakor elérhető legfrissebb 8.0.18-as verzióját használtam.

**Frontend**

A felhasználói felület, vagy más néven UI (User Interface) tervezésénél elsődleges szempontnak számított, hogy a felhasználók számára egyszerűen kezelhető és érhető, és átlátható legyen, minél nagyobb transzparenciát követelve meg. Ezek főleg a saját, valamint barátoktól és kollégáktól összeszedett információk, kinek mi fontos, hogy igazán ergonomikus legyen egy weboldal. A fent említett specifikációk és elvárások alapján a következőket határoztam meg a felhasználói felülettel szemben:

* A kliensoldali alkalmazás legyen képes HTTP (**H**yper**T**ext **T**ransfer **P**rotocol) protokollon keresztül kommunikálni (kéréseket küldeni és fogadni) egy REST API szolgáltatással.
* Felhasználó regisztrációja, valamint a már regisztrált felhasználók beléptetése.
* A felhasználói felületet érintő logika megvalósítása.
* Regisztrált és bejelentkezett felhasználók kiléptetése az alkalmazásból.
* Csapatok létrehozása, törlése. Csapathoz való csatlakozás, vagy csapat elhagyása.
* Ötletdobozba történő ötletek létrehozása, törlése, valamint teljesített státuszúvá változtatása.
* Különböző felelősségi körök kezelése. Két meghatározott feladatkör létezik, *Developer* és *Scrum Master*. Bizonyos interakciók a *Developer* elől rejtve maradnak, ezekkel csak a *Scrum Master* tud interakcióba lépni.
* Retrospecive boardon történő „szoba” létrehozása.
* Az adott szobában különböző „issue”-k, problémákról történő kártyák létrehozása az előre ledefiniált három témához.
* Új érték hozzáadása a NicoNico táblázathoz.
* Kompetencia oldalon, új kompetencia létrehozása, valamint a saját csapatunkhoz történő hozzárendelése.
* Kompetencia mátrix esetén, a csapatunkhoz rendelt már létező kompetenciák közül történő konfigurálás az adott személyre szabva.
* Üzenetküldési lehetőség a csapat minden tagja számára.
* Pozíció beállítása *Developer*-ről *Scrum* *Master*-re.

**Struktúra**

A struktúra szempontjából a megjelenítés teljesen elkülönül a modell és adatbázis rétegektől. A megvalósításhoz használt React.js frontend oldali komponens alapú könyvtár, és itt érdemes pár szót szánni a könyvtár kifejezésre, mivel a React hivatalosan nem egy frontend oldali keretrendszer. A React nem támogatja az MVC (Model – View – Controller) architektúrát, ez csak egy könyvtár, ami segítségével UI elemek jeleníthetők meg és szabhatók személyre. React esetén csak V-ről, tehát csak a View-ról beszélhetünk. Előnye, hogy gyors, könnyen tanulható, könnyen újraírható, javítható vagy refaktorizálható a komponens alapú szerkezet miatt, viszont ez nagyobb projektek esetén a visszájára fordulhat, mivel a túl sok különálló egységként kezelt komponens átláthatatlanná teszi a struktúrát, és nehezen kezelhetővé az állapot menedzsmentet (state management).

**Szerkezet és felépítés**

A szoftver projekt mappájában található két könyvtár, ez a *public* és az *src*. Ezek tartalmazzák a frontendet felépítő kódokat. Ebben a részben ezek tárgyalását végzem, valamint kitérek a mappastruktúra és elrendezés logikájára. A kódbázis megírása során szem előtt tartottam a JavaScript programozási paradigmáit, valamint a konvenciókat, így minden osztály vagy komponens nagy kezdőbetűvel kezdődik, valamint ha szóösszetételről van szó, akkor minden egyes kezdőbetű nagybetű. Ez megfelel az úgynevezett *CamelCase* konvenciónak.

A komponensek szerkezetét tekintve a React két lehetőséget kínál, amit az implementálni kívánt komponens tulajdonságai és funkcionalitásai alapján érdemes megválasztani. Ez a kettő a *Class Component* és a *Functional Component*. Ezeknek a fájloknak jsx kiterjesztésük van. De vajon miért jsx, és miért nem js? Az előző két kérdés összefügg egy másik kérdéssel, miszerint miért jobb, miért több a React, és miért kéne bárkinek ezt használnia? A jsx kiterjesztés jelentése JavaScript XML, ami azt jelenti, hogy ezáltal mind szintaktikailag mind szemantikailag lehetőségünk van a jelenleg elérhető natív ES6 (ECMAScript 6) verziónak megfelelő JavaScript kódunkat kiegészíteni XML (Extensible Markup Language) szintaktikájú kódokkal. Ezt a *React.createElement* függvény a böngésző számára értelmezhető natív szabványosított JavaScript kóddá generálja. Kezdetben két nagy eltérés volt megfigyelhető az Osztályszintű *(Class Component)* és a Functionális komponensek *(Functional Component)* között. A fő különbség, hogy az osztályszintű komponensek rendelkeznek úgynevezett *state*-tel, tehát állapotuk van, ami dinamikusan képes változni, valamint van egy *lifecycle* metódusuk, ami meghatároz egy életciklust a komponensnek. Ezek kezdetben hiányoztak a funcionális komponensekből. Az egyik legfontosabb dolog, ami mind a két típusú komponensnek része, az a *prop* tag, ami arra használható, hogy a szülő komponens render metódusában adatokat, vagy információkat lehessen közvetíteni a gyerek komponensek felé, így a gyerekek a propson keresztül megkapják a számukra szükséges információt. Fontos leszögezni, hogy a props, mint objektum nem dinamikus, mint a state, csak akkor változik meg, ha a szülő újra meghívja a render függvényét, akár más értékekkel feltöltve a gyerekek propsait. Manapság a határ nem olyan éles a funkcionális és osztályszintű komponensek között, mivel 2019. második felének elején (React v16.8) egy új API jelent meg a reacthoz, ami a React Hooks API nevet kapta. Ennek segítségével tudunk definiálni saját, akár több különálló state változót funkcionális komponensekhez, valamint saját Hook-ok létrehozására is van lehetőség. A funkcionális komponens nem rendelkezik render metódussal, csak az osztályszintű komponens.

* public
  + index.html
    - Tartalma egy HTML5 formátumnak megfelelő html fájl, aminek a tartalma egy üres *div* html komponens, aminek egy *id* attribútuma van *root* néven. Ez később nagyon fontos lesz, mivel később a ReactDOM.render függvény a Virtual Dom (Virtual Document Object Model) segítségével ezt a divet fogja feltölteni tartalommal.
* src
  + index.js
    - Egyetlen egy funkcionalitást hajt végre. Az index.html-ben található *root* *id*-val rendelkező *div* komponensre meghívja a *render* függvényt, és átadja neki az *App* nevezetű komponenst, ez hamarosan kifejtésre kerül.
  + setupProxy.js
    - Itt történik a proxy beállítása. E nélkül a belépés és regisztráció nem történne meg.
  + App.jsx
    - Funkcionális komponens
    - Gyakorlatilag ez a fő komponens, ami végül megjelenítésre kerül. Lehet rá úgy tekinteni, mint mindegyik funkcionális egység, vagy komponens szülőjére. Minden komponens az App komponens gyermeke. Itt vannak definiálva a route-ok (útvonalak), amikre kérések jöhetnek, és a hozzájuk tartozó komponensek megadása, hogy a megfelelő útvonalon a megfelelő komponens kerüljön átadásra.

const App = () => {

return (

<*Router* *history*={history}>

<*UserProvider*>

<*Route* *path*="/" *component*={MenuBar} />

<*Route* *path*="/profile" *component*={Profile} />

<*Route* *path*="/messageBoard" *component*={MessageBoard} />

<*Route* *path*="/dashboard" *component*={DashBoard} />

<*Route* *path*="/metrics" *component*={Metrics} />

<*Route* *path*="/retrospective" *component*={RetroSpective} />

<*Route* *path*="/ideabox" *component*={IdeaBox} />

<*Route* *path*="/team" *component*={Team} />

</*UserProvider*>

<*Route* *path*="/" *exact* *component*={Home} />

</*Router*>

);

};

* + - Itt megfigyelhető, hogy a *„/”* útvonalon kívül mindegyik másik útvonal gyereke egy *UserProvider* nevezetű komponensnek. Ez később kerül kifejtésre, viszont ebben a részben elég annyit tudni róla, hogy ha a *UserProvider* komponensnek van tartalma, tehát nem üres, akkor a rendszer elérhetővé teszi a benne foglalt útvonalakat is, különben csak a „/”-t, ami a főoldal.
  + history.js

import {createBrowserHistory } from "history";

export default createBrowserHistory();

* + - Elkészül egy *history* objektum, aminek a segítségével a böngésző cache-ében tudjuk tárolni a már meglátogatott útvonalakat, és ezekhez vissza, és előre tudunk menni a böngésző navigációs sávját használva.
  + components mappa
    - buttons mappa
      * ButtonList.jsx
        + Funkcionális komponens
        + „*Wrapper*”-ként funkcionál.
        + A data mappában található konfigurációs beállítások alapján visszatér egy *IconButton* komponenssel.
        + Ez később egy legördülő menü része lesz, és a Social Login (Google+) funkcionalitást fogja kezelni.
      * IconButton.jsx
        + Funkcionális komponens
        + Tartalma egy *<a>* html tag, ami a szülőtől kapott *props*-ok alapján elkészít és visszatér egy teljesen kitöltött <a> taggal, aminek van *img*, *href*, alt, *color*, *txt*, és *name* attribútuma.
    - cards mappa
      * Card.jsx
        + Funkcionális komponens.
        + *img*, *href,* *name*, és *color* attribútummal rendelkezik.
        + Egy darab formázott <div> komponenssel tér vissza a propsok alapján feltöltve.
      * CardList.jsx
        + Funkcionális komponens.
        + A data mappában található konfigurációs beállítások alapján visszatér egy *<Card>* komponenssel.
      * RetroCard.jsx
        + Funkcionális komponens.
        + Egy darab propssal rendelkezik, ami a *description*, ez az az érték, ami megjelenik a kártyán.
      * RetroRoomCard.jsx
        + Funkcionális komponens.
        + Három darab érték található a props-ban.
        + data.roomName:

Aktuális szoba neve

* + - * + data.id:

Aktuális szoba ID-ja. Ez alapján lesz azonosítható az adatbázisban.

* + - * + onClickEvent:

Szülő komponens *onClickEvent* függvénye.

* + - displays mappa
      * ChartDisplay.jsx
        + Funkcionális komponens
        + Visszatérési értéke egy darab RadarChart komponens, aminek átadja a felhasználó csapatának id-ját, valamint a *generateLables* függvényt, aminek visszatérési értéke egy lista a csapatra vonatkozó kompetenciákból.
      * CompetencyTable.jsx
        + Funkcionális komponens
        + API kérések:

*/api/competencies*

Összes, már létrehozott kompetencia lekérése az adatbázisból API-n keresztül.

Metódus:

GET

*/api/addTeamCompetency*

competencyId és teamId alapján hozzárendel a felhasználó csapatához egy kompetenciát.

Metódus:

POST

* + - * IdeaTable.jsx
        + Funkcionális komponens.
        + Feladata megjeleníteni a szülőtő komponenstől kapott objektum és propsok alapján egy dinamikusan változó táblázatot.
        + API kérések:

*/api/deleteidea*

Létező ’idea’ törlése név alapján az adatbázisból.

Metódus:

POST

*/api/completeidea*

Létező ’idea’ *complete* mezőjének igazra való állítása.

Metódus:

POST

* + - * Messages.jsx
        + Funkcionális komponens
        + Csapatnév alapján állít elő annyi SingleMessage komponenst, ahány rekord található az adatbázisban az adott csapathoz.
      * NicoNicoTable.jsx
        + Funkcionális komponens
        + Vizuálisan megjeleníti a csapatra vonatkozó NiciNico-kat, táblázatos formában. ( Felhasználó neve, Dátum, NicoNico értéke)
        + API kérések:

/api/niconicos

Összes NicoNico adat lekérése szűrve a csapat id-ra.

Metódus:

GET

* + - * ProfileTable.jsx
        + Funkcionális komponens
        + Feladata vizualizálni a bejelentkezett felhasználó adatai alapján kulcs-érték párokként a tulajdonságokat. (Felhasználó neve, id, Profilkép url címe, pozíció (Developer / Scrum Master), csapat (ha létezik).
        + API kérések:

*/api/setProfile*

Felhasználónév és id alapján frissíti az adatbázisban a felhasználóhoz tartozó pozíciót Developerről Scrum Masterre.

Metódus:

POST

* + - * RadarChart.jsx
        + Funkcionális komponens
        + Csapat id alapján dinamikus diagramon vizualizálja csapattagokra lebontva a kompetenciákat, egy hálót alkotva.
        + API kérések:

*/api/usercompetency/<csapatId>*

Csapat id alapján a hozzá tartozó kompetenciák csapattagonként, és a hozzájuk tartozó egyedi értékekkel.

Metódus:

GET

* + - * SingleMessage.jsx
        + Funkcionális komponens
        + Egy darab üzenet vizualizálására szolgáló komponens.
        + Három darab props attribútuma van, a *value* ami az üzenet (message) tényleges tartalma, valamint a *sender* ami a küldő neve, és a *date*, ami a küldés dátuma.
      * SliderConatainer.jsx
        + Funkcionális komponens
        + Visszatérési értéke egy CompetencySlider komponens, ha a teamData változó nem üres, ellenkező esetben egy *Loading*… szöveg jelenik meg.
      * SnackBar.jsx
        + Szerintem ez nincs használva am
      * TeamTable.jsx
        + Funkcionális komponens
        + Összes létező csapat megjelenítésére szolgáló komponens, amikkel interakcióba lehet lépni. (Join / Leave).
        + API kérések:

*/api/deleteteam*

Metódus:

POST

Id alapján kitörlésre kerül a csapat az adatbázisban.

*/api/jointeam*

Metódus:

POST

Felhasználónév és csapat id alapján felhasználó összekötése az adott csapattal. ’Csatlakozás a csapathoz’

* + - * Terminal.jsx
        + Funckionális komponens
        + Üzenetek megjelenítésére szolgáló vizuális elem.
    - inputs mappa
      * CompetencyInputFiled.jsx
        + Funkcionális komponens
        + Visszatérési értéke egy szövegmező, valamint egy funkciógomb. Feladata a szövegmezőbe írt érték adatbázisba való mentése, mint új kompetencia.
        + API kérések:

*/api/createCompetency*

Meódus:

POST

Név alapján létrehoz egy új rekordot a competency táblában.

* + - * CompetencySlider.jsx
        + Funkcionális komponens
        + Visszatérési értéke egy szöveg, a kompetencia nevével, valamint egy Slider „csúszka” komponens, ami dinamikusan menti a beállított adatokat az adatbázisba amint változtatás történt.
        + API kérések:

*/api/saveCompetency*

Metódus:

POST

Paraméterek:

Kompetencia Id-ja

Felhasználó Id-ja

Beállított érték

* + - * IndeaInputField.jsx
        + Funkcionális komponens
        + Feladata a szövegmezőbe beírt érték alapján új idea létrehozása az adatbázisban.
        + API kérések:

*/api/idea*

Metódus:

POST

Paraméterek:

Szövegmező értéke

Csapat neve

’Completed’ érték

* + - * MessageInputField.jsx
        + Funkcionális komponens
        + Feladata a szövegmezőbe írt érték adatbázisba való mentése, mint üzenet.
        + API kérések:

*/api/sendMessage*

Metódus:

POST

Paraméterek:

Szövegmező értéke

Dátum ->ez rosszs!!!

Csapat neve

Felhasználó neve

* + - * RetroCreator.jsx
        + Funkcionális komponens
        + Feladata a szövegmezőbe írt érték alapján új retrospective rekord létrehozása az adatbázisban.
        + API kérések:

*/api/createRetroSpective*

Metódus:

POST

Paraméterek:

Csapat neve

Szövegmező értéke

* + - * RetroInputField.jsx
        + Funkcionális komponens
        + Három komponens alkotja. Szövegmező, *RetroSelect* komponens, valamint egy funkciógomb. Feladata, hogy új rekordot hozzon létre az Issue táblában az adatbázisban, a szövegmező értéke, a RetroSelect-ben kiválasztott érték, és az adott RetroSpective id alapján.
        + API kérések:

/api/createIssue

Metódus:

POST

* + - * RetroSelect.jsx
        + Funkcionális komponens
        + Három választási lehetőség adott benne:

Worked well

To be impoved

Want to do in the next sprint

* + - * + Ezek megfelelnek a retrospective alá tartozó issue-k típusainak.
      * Smiley.jsx
        + Funkcionális komponens
        + Tartalma három darab smiley ikon, a hozzájuk tartozó értékekkel, valamint a szülőtől kapott *postNicoNico* függvénnyel.
    - menus mappa
      * DataTags.jsx
        + SEHOL NINCS HASZNÁLVA
      * MenuBar.jsx
        + Funkcionális komponens
        + Ez a komponens reprezentálja a navigációs menüt az oldal teteján. Megtalálható rajta dinamikusan a különböző oldalakra vezető ikonok, valamint az útvonalak meghatározása. A komponens része még egy legördülő menü, valamint egy kilépés gomb, ami a */auth/logout* útvonalra vezet, amit a szerver az előre definiált viselkedéssel kezel, és kilépteti a bejelentkezett felhasználót.
      * UserDropDown.jsx
        + Funkcionális komponens
        + Legördülő menü, két értékkel, amik hiperhivatkozások.

Profil

Főoldal

* + - stepper mappa
      * DashBoardStepper.jsx
        + Funkcionális komponens
        + Feleadta lépések sorozatán át szöveges segítséggel végig vezetni a felhasználót a szoftver működésén.
        + Minden egyes oldal külön lépésként szerepel.
    - wrappers mappa
      * Col????????
  + contexts mappa
    - CompetencyProvider.jsx
      * Context API használatával létrehozott provider / kontextus.
      * Felhasználó csapatára vonatkozó összes adat egy helyen egy kontextusban történő tárolása. Ehhez az összes gyerek komponens hozzáfér.
      * API kérések:
        + */api/teamCompetencies*

Metódus:

GET

* + - UserProvider.jsx
      * Context API használatával létrehozott provider / kontextus.
      * A kontextus értéke a bejelentkezett felhasználó minden adata. Mivel egy kontextusnak végtelen számú gyereke lehet, ezért hasznos, az alkalmazás elején definiálni az ilyen kontextusokat, így az összes gyerek komponens, és azok gyerekei is elérik a felhasználó összes adatát, de mégse redundáns a kód.
      * API kérések:
        + */user*
        + Metódus:

GET

* + data mappa
    - index.js
      * A UI által használt Google ikon meta adatai találhatóak ebben a fájlban. Ezek az *img*, *name*, *href*, *alt*, *color* és *txt* kulcsok és a hozzá tartozó értékek.
      * Ezt használja a ButtonList.jsx fájl.
  + pages mappa
    - DashBoard.jsx
      * Funkcionális komponens
      * Egyetlen komponenst tartalmaz, ami a DashBoartStepper
    - Home.jsx
      * Funkcionális komponens
      * Ez a komponens látja el a „Főoldal” szerepet.
      * Része a Terminal komponens, ami eltérő időben üzeneteket jelenít meg, valamint egy Google login kártya komponens.
    - IdeaBox.jsx
      * Funkcionális komponens
      * Két fő komponensből áll:
        + IdeaInputField
        + IdeaTable
      * API kérések:
        + /idea
        + Összes létező rekord az adatbázisból, ezek adódnak át a gyerek komponenseknek.
        + Metódus:

GET

* + - MessageBoard.jsx
      * Funkcionális komponens
      * Két fő komponens alkotja:
        + Messages
        + MessageInputField
      * API kérések:
        + */messageBoard*

Üzenetek lekérése az adatbázisból.

Metódus:

GET

Ezek átadódnak a gyerek komponenseknek.

* + - Metrics.jsx
      * Funkcionális komponens
        + Dinamikusan vezérelhető TabPanel komponens fedi el a különböző komponenseket.
        + Itt található a NicoNico-ra, valamint a metrikákra vonatkozó komponensek.
        + API kérések:

/api/addNicoNico

Metódus:

POST

Paraméterek:

Felhasználó id

Niconico értéke (1,3,5)

* + - Profile.jsx
      * Funkcionális komponens
        + It kerül megjelenítésre a ProfileTable komponens a felhasználóra vonatkozó adatok alapján.
    - RetroSpective.jsx
      * Funkcionális komponens
        + Tartalma a RetroCreator komponens, valamint itt kerül vizualizációra a már kész retrospective-k nevei, amikre kattintva, megjelenítésre kerül az *issueView*.
        + API kérések:

*/retrospectives/<teamId>*

Csapat id alapján lekéri az adatbázisból a retrospective rekordokat.

Metódus:

GET

*/api/issues/<retroId>*

Retrospective id alapján lekéri az adatbázisból az Issue táblában található rekordokat.

Metódus:

GET

* + - Team.jsx
      * Funkcionális komponens
      * Két fő komponensből áll:
        + CreateTeamInputField
        + TeamTable
      * API kérések:
        + /team

Lekéri az adatbázisból a Team táblából az összes rekordot.

Metódus:

GET

* + res mappa
    - google.png
  + style mappa
    - index.css
      * Itt található külön az oldal fő szerkezetét érintő css kódok, ami az oldal dizájnját hivatott alkotni.

**Projekt mappaszerkezetének felosztása és logikája**

Ahogy fentebb is megfigyelhető, a különböző programkódok és fájlok külön mappákba, és szerkezeti egységekbe vannak rendezve. Ez nagyban megkönnyíti a projekt átláthatóságát, valamint modularitását. A szempont, ami szerint dolgoztam, hogy a mappa és modulnevek legyenek egyszerűek, és könnyen értelmezhetők.

components

Ide kerül implementálásra az összes különálló egységként kezelendő komponens, amik kisebb részeit alkotják majd nagyobb egységeknek. Minden komponens ez alá a mappa alá kerül, akár almappákba szervezve. A forráskód élettartalmát és fenntarthatóságát nagyban növeli a modularitás, és az egységekbe zárás.

Components/buttons

Components/cards

Components/displays

Components/inputs

Components/menus

Components/stepper

Components/wrappers

**Mi is az Agilis szoftverfejlesztés?**

Ebben a fejezetben az Agilis szoftverfejlesztés fő elemeit, kialakulást, valamint a Scrum agilis fejlesztési módszertan kerül kifejtésre. Különböző aspektusokból vizsgáljuk meg mi a létjogosultsága az agilis módszertan használatának szoftverfejlesztői környezetben, valamint kifejtésre kerül a Scrum mint módszertan, a csapat felépítése és a módszertan adta események.

**[1] Kiáltvány az agilis szoftverfejlesztésért**

A szoftverfejlesztés hatékonyabb módját tárjuk fel saját tevékenységünk és a másoknak nyújtott segítség útján. E munka eredményeképpen megtanultuk értékelni:

1. Az egyéneket és a személyes kommunikációt a módszertanokkal és eszközökkel szemben
2. A működő szoftvert az átfogó dokumentációval szemben
3. A megrendelővel történő együttműködést a szerződéses egyeztetéssel szemben
4. A változás iránti készséget a tervek szolgai követésével szemben

Azaz, annak ellenére, hogy a jobb oldalon szereplő tételek is értékkel bírnak, mi többre tartjuk a bal oldalon feltüntetetteket.

A 2001-ben megjelent fent olvasható kiáltvány szembement az addigi fejlesztési módszerekkel, valamint csapat és vállalati struktúrákkal. Az agilis módszertan nagyobb hangsúlyt fektet az egyénre, és az úgynevezett „soft skillek”-re, mint a csapat orientáltság, kommunikációs készség, szervezőképesség, és a rugalmasság. A szoftver minősége elsőbbséget élvez a dokumentáltsággal szemben, valamint folyamatos kapcsolat tartását a vevő és a fejlesztőcsapat között. A kiáltvány magába foglalja az *Agilis szoftverfejlesztés 12 alapelvét*, ami nagyobb szabadsággal látja el a fejlesztőcsapatot, viszont ez a nagyobb szabadság és rugalmasság nagyobb felelősséget is helyez a csapatra. A döntések meghozatalától kezdődően együtt kell működjenek az üzleti szakértők a fejlesztők és a vevők egyaránt, ezzel elősegítve a könnyebb és gyorsabb információ folyamot. A csapat felelőssége a folyamatos kiválóságra való törekedés, a csapat hatékonyságának és tudásának növelésére való igény fenntartása, és a munkakörülmények minél gördülékenyebbé tétele. Végére pedig érdemes megemlíteni az egyik legfontosabb újítást az Agilis módszertan és az addigi bevált módszerek között, ami a gyakoribb szállítás. Ez azt jelenti, hogy a szoftvert kisebb egységekben, általában pár hetenként, vagy havonta szállítani kell a vevő felé. Ezek kisebb újítások. Amivel ez a gyakori szállítás értéket teremt a régebbi módszerekhez képest, hogy a folyamatos kisebb egységenként történő szállítás esetén azonnal lehetősége van a vevőtől visszajelzést kapnia a fejlesztőcsapatnak, így rugalmasan igazodva az esetleges változtatásokhoz, véleményekhez, minthogy egyszer szállítja a teljes egészében elkészült szoftvert, ami potenciális hibákat, vagy a vevői igényektől eltérő dolgokat tartalmazhat. A agilis esetben a végtermék megfelel minden vevői igénynek, és további változtatásra nincs szükség.

**[2] Agilis szoftverfejlesztés 12 alapelve:**

1. Legfontosabbnak azt tartjuk, hogy az ügyfél elégedettségét a működő szoftver mielőbbi és folyamatos szállításával vívjuk ki.
2. Elfogadjuk, hogy a követelmények változhatnak akár a fejlesztés vége felé is. Az agilis eljárások a változásból versenyelőnyt kovácsolnak az ügyfél számára.
3. Szállíts működő szoftvert gyakran, azaz néhány hetenként vagy havonként, lehetőség szerint a gyakoribb szállítást választva.
4. Az üzleti szakértők és a szoftverfejlesztők dolgozzanak együtt minden nap, a projekt teljes időtartamában.
5. Építsd a projektet sikerorientált egyénekre. Biztosítsd számukra a szükséges környezetet és támogatást, és bízz meg bennük, hogy elvégzik a munkát.
6. A leghatásosabb és leghatékonyabb módszer az információ átadásának a fejlesztési csapaton belül, a személyes beszélgetés.
7. A működő szoftver az elsődleges mércéje az előre haladásnak.
8. Az agilis eljárások a fenntartható fejlesztést pártolják. Fontos, hogy a szponzorok, a fejlesztők és a felhasználók folytonosan képesek legyenek tartani egy állandó ütemet.
9. A műszaki kiválóság és a jó terv folyamatos szem előtt tartása fokozza az agilitást.
10. Elengedhetetlen az egyszerűség, azaz az elvégezetlen munkamennyiség maximalizálásának művészete.
11. A legjobb architektúrák, követelmények és rendszertervek az önszerveződő csapatoktól származnak.
12. A csapat rendszeresen mérlegeli, hogy miképpen lehet emelni a hatékonyságot, és ehhez hangolja és igazítja az működését.

[1] Kiáltvány az agilis szoftverfejlesztésért - <https://agilemanifesto.org/iso/hu/manifesto.html>

|  |  |  |
| --- | --- | --- |
| Kent Beck Mike Beedle Arie van Bennekum Alistair Cockburn Ward Cunningham Martin Fowler | James Grenning Jim Highsmith Andrew Hunt Ron Jeffries Jon Kern Brian Marick | Robert C. Martin Steve Mellor Ken Schwaber Jeff Sutherland Dave Thomas |

[2] Agilis szoftverfejlesztés 12 alapelve - <https://agilemanifesto.org/iso/hu/principles.html>

[3]

[4]

[5]

[6]

[7]

[8]

[9]

[10]

[11]

[12]

[13]

[14]

[15]