Budapesti Műszaki és Gazdaságtudományi Egyetem
Villamosmérnöki és Informatikai Kar
Automatizálási és Alkalmazott Informatikai Tanszék

Boros Bence

TŐZSDEI KERESKEDÉST SEGÍTŐ WEBALKALMAZÁS FEJLESZTÉSE

KÖVESDÁN GÁBOR

BUDAPEST, 2020

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

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

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

Kelt: Budapest, 2020. 12. 03.

...…….
Boros Bence

Összefoglaló

A szakdolgozatom témájaként mindenképpen szerettem volna valami olyat választani, ami nem egy átlagos, könnyen letudható projekt, hanem beleesik az érdeklődési körömbe és még gyakorlati haszna is lehet. Édesapám révén már kezdetleges ismeretekkel rendelkeztem a devizakereskedés terén, és úgy gondoltam, hogy ez egy kellően sok irányból megközelíthető téma ahhoz, hogy ezen dolgozatom alapjául szolgáljon.

A manuális devizakereskedés egy sok tanulást és tapasztalatot igénylő tevékenység, így nekem programozóként magától értetődő volt a gondolat, hogy célszerű lenne azt automatizálni. Ehhez kell egy algoritmus, amit múltbéli adatok alapján, gyakorlatilag mintafelismerésre optimalizálok. A kényelmes használathoz szükség van egy grafikus felületre is. A modern kor követelményeihez igazodva ezt egy reszponzív webalkalmazás formájában valósítom meg.

Az alkalmazás egy MongoDB adatbázisból, egy Express webszerverből, és egy React grafikus felületből áll. Ezen három főkomponens egyéb, a feladathoz illő technológiákkal kiegészülve egy full-stack rendszert alkot, melynek segítségével bárki elkezdhet kereskedni az általam tervezett robottal, valamint saját preferenciái szerint beállíthatja és nyomon követheti annak működését.

A program elkészítése nem várt pontokon is komoly kihívásokat rejtett, melyek megoldásán keresztül sokat fejlődtem, mint programozó. Egy egyszerűbb témával talán egy teljesebb és laikus szemmel nézve szebb alkalmazás születhetett volna, azonban számomra ennek a feladatnak a megoldása nagyobb értékkel bír.

Abstract

As the topic of my dissertation, I wanted to choose something that is not an average, easy-to-do project, but falls within my sphere of interest and can even be of practical use. Through my father, I already had a rudimentary knowledge of forex trading, and I thought it was a topic that could be approached from enough directions to serve as the basis for my dissertation.

Manual forex trading is an activity that requires a lot of learning and experience, so for me as a programmer, it was self-evident that it would be advisable to automate it. This requires an algorithm that I optimize for pattern recognition based on historical data. A graphical interface is also required for convenience. Adapting to the requirements of the modern age, I implement this in the form of a responsive web application.

The application consists of a MongoDB database, an Express web server, and a React graphical interface. These three main components, complemented by other task-appropriate technologies, form a full-stack system that allows anyone to start trading with the robot I designed, set it up according to their own preferences and track its operation.

The making of the program involved serious challenges at unexpected points, through the solution of which I developed a lot as a programmer. A simpler topic might have given rise to a more complete and in a layman's view more eye-catching application, but for me, solving this task is of greater value.

1 Bevezetés

1.1 Motiváció

Felszínes szinten mindig is érdekeltek a gazdasági folyamatok, azon belül is a tőzsdén való kereskedés. Természetesen ebben része van annak is, hogy a gazdaságnak talán ez az az ágazata, ahol a legnagyobb haszonra lehet szert tenni. Ehhez persze jól kell ismerni a folyamatokat, ami felé részemről az első lépés ennek a dolgozatnak a témaválasztása volt.

1.2 Tőzsdei kereskedés

A tőzsde olyan intézményként született, amelynek helyzetét ugyanúgy, mint egy szabályalkotási, közigazgatási és bíráskodási joggal felruházott testületét, nem egy törvény, hanem a fejlődés különböző időpontjaiban alkotott különböző jogszabályok, illetve a szokásjog szabályozta. A tőzsde a piacok piaca. Központosítja a kereskedelmet nem csak egy ország, hanem az egész világpiac számára. A modern gazdaságban ugyanis a kereslet és a kínálat központosítása létszükséglet. [1]

1.2.1 A tőzsde múltja

Maga az elnevezés az 1300-as ével elején Brüggében élő van der Burse patríciuscsalád nevéből ered. A Burse-k fogadót tartottak fen Brügge központjában, ahol az itáliai kereskedők találkoztak északi üzletfeleikkel, és ide telepítették váltóikat. Brügge szerepét 1460-tól Atwerpen vette át, itt a XVI. Század elején már nemzetközi árutőzsde működött. Érdekesség, hogy a tőzsdét a szeszélyessége miatt nőnemű szóval jelölik több idegen nyelvben is (pl. Die Börse).

A tőzsde a vásárból alakult ki. A különbség persze nagy: a vásárokon mindig jelen volt az áru, a tőzsdén soha. A vásárból tőzsde akkor lett, amikor az üzletkötések tömegessé váltak, és mind az áru, mind a fizetési eszköz oldalán a helyettesíthetőség vált uralkodóvá. Az első tőzsdét Antwerpenben hozták létre. Itt történt meg először, hogy különböző nemzetiségű kereskedők jöttek össze rendszeresen üzleteket kötni. Addig ugyanis a különböző fejedelmi privilégiumokra vigyázó és mindenkire féltékeny más–más nemzetiségű kereskedőket mesterségesen elválasztották egymástól.

Az amszterdami tőzsde a Holland Kelet–Indiai és a Nyugat–Indiai Társaság részvényeinek adásvételével vált a legjelentősebbé. Rövid időn belül még az antwerpeni tőzsde forgalmát is túlszárnyalta. A 17. század folyamán itt már lényegében kialakult a modern tőzsdei üzlet technikája. Ekkor vált ketté az áru– és az értéktőzsde.

Az 1602-ben alapított Holland Kelet-Indiai Társaság először 1605-ben fizetett osztalékot, a 15%-os osztalékfizetés azonban nem készpénzben, hanem borsban történt. A társaság részvényeit kezdetben az utcán és kávéházakban árusították, de hamarosan az üzletkötés az 1611-ben alapított amszterdami tőzsdére tevődött át. A részvények kezdetben névre szóltak, tulajdonosuk felelőssége korlátlan volt. 1672-ben a holland-francia háború idején az Egyesült Tartományok dobtak piacra hosszú lejáratú, egységes címletű adóssági leveleket, és a példát még ugyanabban az évben követte az angol, majd hét év múlva a francia kincstár.

A részvényforgalom a múlt század közepén az első vasúttársaságok megjelenésével indult gyors növekedésnek. A párizsi tőzsdén 1800-ban még csak hét, 1869-ben már négyszáznál több értékpapírt, zömmel részvényt jegyeztek.

A részvénytársaságok alapítását is kontinentális joggyakorlat (először Napóleon 1808-as kereskedelmi törvénye, majd az 1843-as, az előbbinél szigorúbb porosz részvénytársasági törvény) állami engedélyhez kötötte, szigorú felügyeletet írt elő, ugyanakkor megengedte a bemutatóra szóló részvények kibocsátását. Angliában – és az Egyesült Államokban – a társasággá alakulás soha nem volt állami engedélyhez kötve, de itt a részvények a múlt század végéig szigorúan névre szóltak.

Minden elővigyázatosság ellenére a gazdasági–pénzügyi válságok és a túlzott spekuláció időnként így is tőzsdekrachokat idézett elő. Az első tőzsdeválság 1557-ben alakult ki, amikor is az Amerikából behozott aranyra túlzott spekulációt folytattak. Az 1873-as ún. Gründerkrachot a porosz–francia háború utáni nagy alapítási láz okozta. Ebből a válságból már Magyarország is kivette a részét, jelezve, hogy bekapcsolódott a világkereskedelembe. A következő hírhedt tőzsdeválság az 1895-ben a dél–afrikai aranybányákkal kapcsolatos részvényspekulációk miatt rázkódtatta meg a nemzetközi pénzpiacokat. Az 1929. október 14-i New York-i pedig a világválság kezdetét jelezte. A 19. század végétől a kormányok és a nagy bankok jelentősen beavatkoztak a tőzsdei üzletekbe, a krachok megakadályozása céljából.

1.2.2 A devizakereskedés

A devizakereskedés nagyban hasonlít a részvényekkel való kereskedésre, a különbség, hogy itt a szóban forgó termék nem egy társaság által kibocsájtott részvény, hanem devizák keresztárfolyama. Sok tekintetben a devizapiacok állnak a legközelebb a tökéletes piac elméletéhez, aminek alapja a hatalmas forgalom, a piac rendkívüli likviditása, a szereplők magas száma, földrajzi szétszórtsága, a kereskedési idő (hétvégék kivételével napi 24 óra), az árfolyamokat befolyásoló tényezők sokfélesége és a magas elérhető haszonkulcs.

A devizapiac legfontosabb szereplői a kereskedelmi bankok, a külkereskedelemben részt vevő vállalatok, a nem banki pénzügyi intézmények (mint a befektetési alapok vagy a biztosító társaságok) és a központi bankok. Magánemberek is beléphetnek a devizapiacra – egy turista például, amikor a szálloda recepciójánál pénzt vált –, ám ezek a készpénzes tranzakciók a teljes devizakereskedelem töredékét teszik csak ki. [2]

A bankok a devizapiac mindennapos szereplői, hiszen ügyfeleik – főként a vállalatok – devizaigényeinek kielégítése szokásos napi munkamenetük részét képezi. Ezen túl a bankok más bankok számára is jegyeznek árfolyamot, amelyen hajlandók valutát venni, illetve eladni. A bankok közötti devizakereskedelem – amelyet bankközi kereskedelemnek nevezünk – teszi ki a devizapiaci forgalom legnagyobb részét.

1.2.3 Magánemberként a devizapiacon

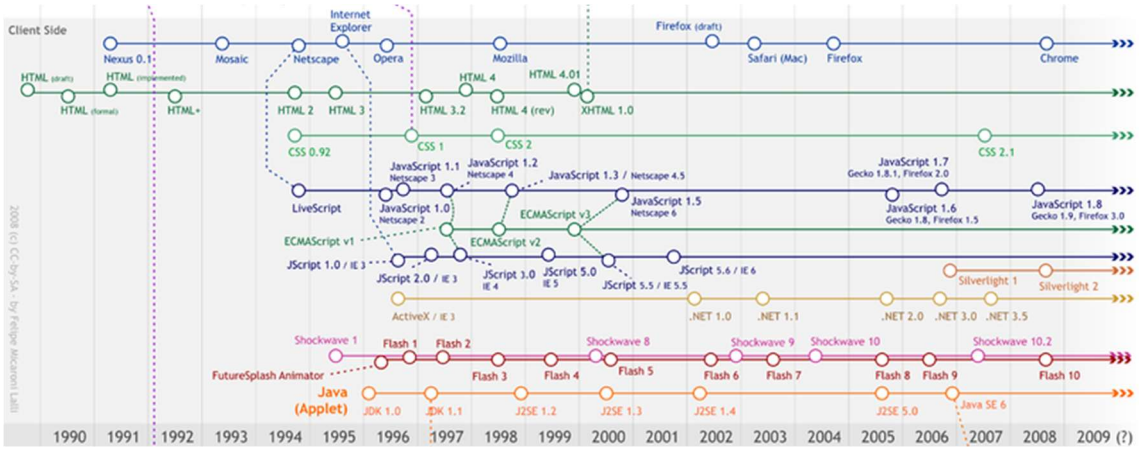
Átlagos állampolgárként is bekapcsolódhatunk a forex piac (foreign exchange market – nemzetközi devizapiac) körforgásába. Ehhez szükségünk van egy bróker cégre. Ezen cégek nemzetközi ellenőrzés alá esnek, így nagyon szigorú szabályok szerint kell működniük.

Rajtuk keresztül intézhetjük az ügyleteinket, valamint a legtöbb esetben tőkeáttételen keresztül extra tőkét nyújtanak. Ez azt jelenti, hogy ha van $1000-om, akkor pl. 1:15-ös tőkeáttétel esetén, $15000 értékben kereskedhetek, vagyis sikeres kötés esetén 15-szörösére nő a profitom. Ugyanakkor egy rosszul sikerült ügylet után a veszteség is 15-ször nagyobb lesz.

1.3 Webes technológiák

Napjainkban a legtöbb hétköznapi alkalmazáshoz tartozik egy webes felület, ugyanis az gyakorlatilag platformfüggetlen, ami azt jelenti, hogy a lehető legszélesebb rétegekhez juttathatjuk így el a szolgáltatásainkat.

1.3.1 Történelem



1. ábra [8]

A webfejlesztés hőskorában egy weboldal nem volt más, mint egy statikus dokumentum. Ezen dokumentumok HTML (HyperText Markup Language) nyelven íródtak, amely a fájl strukturálását hivatott segíteni. Akkoriban a különböző böngészők egyedi módon vizualizálták a szövegesen megadott struktúráltságot.

Ahogy egyre több ember és egyre szélesebb körök kezdték el használni az internetet, úgy már nem volt elég a tartalomra koncentrálni puritán megjelenéssel. Nőtt az igény színvonalas webdizájnra és ezt a változást a nyelvnek is le kellett követnie.

A HTML 1.0 -nál csak az alapvető szerkezet kialakítása volt megvalósítható, olyan elemeké, mint a bekezdések, hivatkozások, fejlécek és felsorolások.

A 2-es verzió sem hozott sok újítást, de lehetőség volt például félkövér vagy dőlt betűk használatára, illetve képek beillesztésére. Ezen verzió kiegészítésében már megjelentek az űrlapok, illetve az ezen belüli többsoros szövegbeviteli és a kiválasztható opciók lehetőségei.

A nagyobb újítást az 1996-ban elfogadott HTML 3.0 és 3.2 jelentette, ahol már lehetőség volt java appletek, valamint scriptek beágyazására, melyek dinamikussá tudták tenni az addig statikus HTML kódot. Ezekben az új verziókban jelent meg először a

„style” elem is, továbbá olyan tartalom- és struktúraformázási megoldások, mint a táblázatok vagy a betűtípusok változtatása.

1998-ban aztán megérkezett a HTML 4.0, melyben már megjelentek a nemzetközi karakterkészletek, támogatták a jobbról balra olvasást, továbbfejlesztették az űrlapok és táblázatok használhatóságát, valamint itt vált hivatalossá a frame-ek (keretek) használata is. [3]

1.3.2 Modern webalkalmazások

2008-ban mutatták be a HTML5-öt, amely aztán 2014-ben szerzett hivatalos ajánlást. Az újítás elsődleges célja a beépülő modulok (Pl. Adobe Flash, Microsoft Silverlight) használatának megszüntetése volt. Ennek érdekében széles körben támogatja a natív multimédiás és grafikus elemeket. Ezen felül az új szabvány lehetőséget ad arra, hogy szemantikailag kifejezőbb strukturális elemeket használjunk. Ezzel a keresőmotorok könnyebben tudják értelmezni az oldalunk felépítését és tartalmát, amiért előrébb sorolják azt a találatok között.

Manapság a weboldalak túlnyomó többsége a HTML5 szabványt követi, azonban a webfejlesztők egyre kevesebb nyers HTML kódot írnak. Előtérbe kerültek ugyanis a reszponzív megjelenést és könnyebb fejlesztést biztosító keretrendszerek (pl. Angular), és osztálykönyvtárak (pl. React).

Ezen eszközökkel könnyedén készíthetünk egy oldalas alkalmazásokat, amelyek felhasználói oldalon tapasztalt teljesítmény szempontjából felülmúlják a hagyományos megoldásokat.

2 Felhasznált technológiák

2.1 Algoritmus

Az alkalmazás alapja egy forex kereskedő algoritmus, amely előre meghatározott mintákat figyelve reagál a piaci mozgásokra. Ennek a fejlesztésére a MetaTrader 4 kereskedő platform az MQL4 nyelvet szolgáltatja.

2.1.1 MetaQuotes Language 4 (MQL4)

Az MQL4 alapvetően egy integrált programozási nyelv, amely segítségével kereskedő robotokat, indikátorokat, szkripteket és függvénykönyvtárakat lehet készíteni.

Az MQL4 egy C++ koncepciókat követve született, magas szintű, objektumorientált nyelv, amely rengeteg, az adatok elemzéséhez szükséges beépített függvényt tartalmaz. Tartalmaz továbbá beépített indikátorokat, így a leggyakrabban használt eszközök rendelkezésre állnak.

A nyelv felhasználásának két iránya van. Az egyik az adatok elemzése és ezzel a manuális kereskedés professzionálisabb szintre emelése, a másik a kereskedés teljes automatizálása. Én az utóbbi irányba indultam el. Ez ugyan leveszi az emberről a manuális kereskedés terhét, azonban a robotokat karban kell tartani, mivel az idő múlásával és a piac változásával azok elavulhatnak.

2.2 Backend

Az alkalmazás lelke a kiszolgáló szerver, amely kezeli az adatbázist, a több-felhasználós működést, figyeli a piaci mozgásokat, végrehajtja a szükséges piaci tevékenységeket, futtatja a historikus tesztert, valamint adatokat szolgáltat a grafikus felületnek. Ezen feladatok ellátása érdekében az alábbi technológiákat használtam fel.

2.2.1 MongoDB / Mongoose

A MongoDB egy jól skálázódó, flexibilis dokumentum adatbázis (NOSQL - Not Only SQL), amely felépítéséből adódóan natívan támogatja a JavaScript fejlesztést.

A technológiát 2007-ben kezdték el fejleszteni, így meglehetősen kiforrottnak számít a dokumentum adatbázisok között és ez is a legnépszerűbb választás. Az adatbázis

kezelése a mongo shellen keresztül történik, ami a JavaScript V8-as motorját használja. [4]

MongoDB-ben az adatokat JSON-szerű dokumentumokban tároljuk, így dokumentumról dokumentumra változhat azok struktúrája. Ez a felépítés lehetővé teszi struktúrák egymásba ágyazását is, amely a hagyományos adatbázisok sokszor költséges *join* műveletét eliminálja. Ez ugyanakkor hibázásra is lehetőséget ad, hiszen innentől a programozó felelőssége, hogy az adatmentések és a lekérdezések helyesek legyenek, de következetesen írt programkóddal könnyen ki lehet aknázni a technológia előnyeit. Az adatbázis minden dokumentumot egyedi _id azonosítóval lát el, ezeken keresztül lehet azokat hivatkozni. A dokumentumokat kollekciókba szervezhetjük, ezzel az azonos jelentésű dokumentumokat egy helyről érhetjük el.

Mindezeken felül a MongoDB teljesen ingyenesen használható, ami a neves adatbáziskezelő szoftverekhez képest egy újabb előny.

A Mongoose egy MongoDB-hez fejlesztett NPM csomag, amelynek segítségével nem kell nyers adatbázishívásokat végrehajtanunk. Ez az eszköz elvégzi helyettünk a validációt, a típusegyeztetést, leegyszerűsíti a lekérdezéseket és a mentéseket is, valamint végrehajtja az objektummodellezést.

2.2.2 Node.js

A Node.js egy szoftverrendszer, melyben JavaScriptben írhatunk szerver oldali alkalmazásokat. Maga a rendszer C/C++-ban íródott, és egy esemény alapú I/O rendszert takar a JavaScript V8 motorja felett.

A JavaScript a világon a jelenleg legnépszerűbb programozási nyelv, többek között ezért esett erre a választás. A Node.js-t úgy írták meg, hogy (szinte) minden esemény aszinkron legyen, ezért a program sosem blokkolódik, azaz nem kell várni, hogy egy művelet befejeződjön, hanem vele „párhuzamosan” futtathatunk további műveleteket. Ez pontosan ugyan úgy működik, mint a böngészőben levő kérések, vagy más események, mint például a *click*. Ez az alkalmazás folyamatosabb futását, valamint az egész rendszer optimálisabb működését teszi lehetővé. [5]

Mindezek mellett a Node.js legnagyobb előnye a Node Package Manager (NPM), amely egy online tárhely nyílt forráskódú Node.js projekteknek, valamint egyben egy parancssoros csomag-, verzió- és függőségkezelő is. Az egyszerű használat és az elérhető

könyvtárak mennyisége mellett a fejlesztés alatt álló projektek hordozhatóságát is elősegíti, ugyanis a szükséges függőségeket a *package.json* fájlban tárolja, aminek segítségével egy parancsra bárhol telepíthetőek a szükséges könyvtárak.

2.2.3 Express.js

Az Express.js egy minimalista, könnyűsúlyú keretrendszer Node.js felett, ami általános megoldást kínál webalkalmazások készítéséhez.

A keretrendszer könnyen kezelhetővé teszi az útvonalválasztást és az adott útvonalon végrehajtandó feladatok strukturálását is. Az útvonalakra middleware-eket lehet feliratkoztatni, amelyek három paraméterrel rendelkező függvények. A három paraméter a *request*, a *response* és a *next*. Az első a beérkező http kérést, a második az arra adott választ takarja. Az egy útvonalra feliratkoztatott middleware-ek egy láncot alkotnak, amelyek közül az Express.js az elsőt hívja meg. A harmadik paraméter (ami egy függvény) szolgál arra, hogy egy middleware-ből továbblépjünk a láncban utána következőre. A next meghívása nélkül a futási lánc megszakad és a keretrendszer elküldi a HTTP választ. Amennyiben előrelátóan fejlesztünk, a middleware-ek adott esetben többször is felhasználhatóak, amely csökkenti a kódduplikációt.

Az egyszerűsége és robosztussága miatt széles körben kedvelt alap API-k készítéséhez.

2.2.4 Passport.js

A Passport.js egy authentikációs middleware Express.js-hez. Egy cél kiszolgálására tervezték, az pedig HTTP kérések authentikációja. Használatával követjük a funkciók szétválasztása elvet, hiszen egymagában old meg egy feladatot, és így később nekünk ezzel máshol már nem kell foglalkoznunk.

Támogat a legegyszerűbb jelszó alapú lokálistól kezdve a token alapú és az OAuth authentikáción keresztül egészen a közösségi fiókokkal való belépésig mindent, amire egy webalkalmazás programozójának szüksége lehet.

2.2.5 Oanda API / Simple FxTrade

Az Oanda egy online brókercég, akiken keresztül bárki bekapcsolódhat a tőzsdepiac működésébe.

A kereskedéshez kínálnak saját webes platformot, más asztali rendszereket kiszolgáló szervereket, valamint API-t is szolgáltatnak. Én az utóbbi lehetőséggel éltem és ennek segítségével írtam meg a kereskedő alkalmazásomat.

Az API-n keresztül elérhető az összes devizapárhoz tartozó grafikon összes adata, amelyeket felhasználva egyedi elemzéseket, vagy akár kereskedő algoritmusokat lehet készíteni. Ugyanitt teljes funkcionalitást kapunk pozíciók kötése terén is.

Az API-hoz a Simple FxTrade NPM csomag kínál wrapper szolgáltatást, ami leegyszerűsíti az API-hívások kezdeményezését.

2.3 Frontend

2.3.1 React

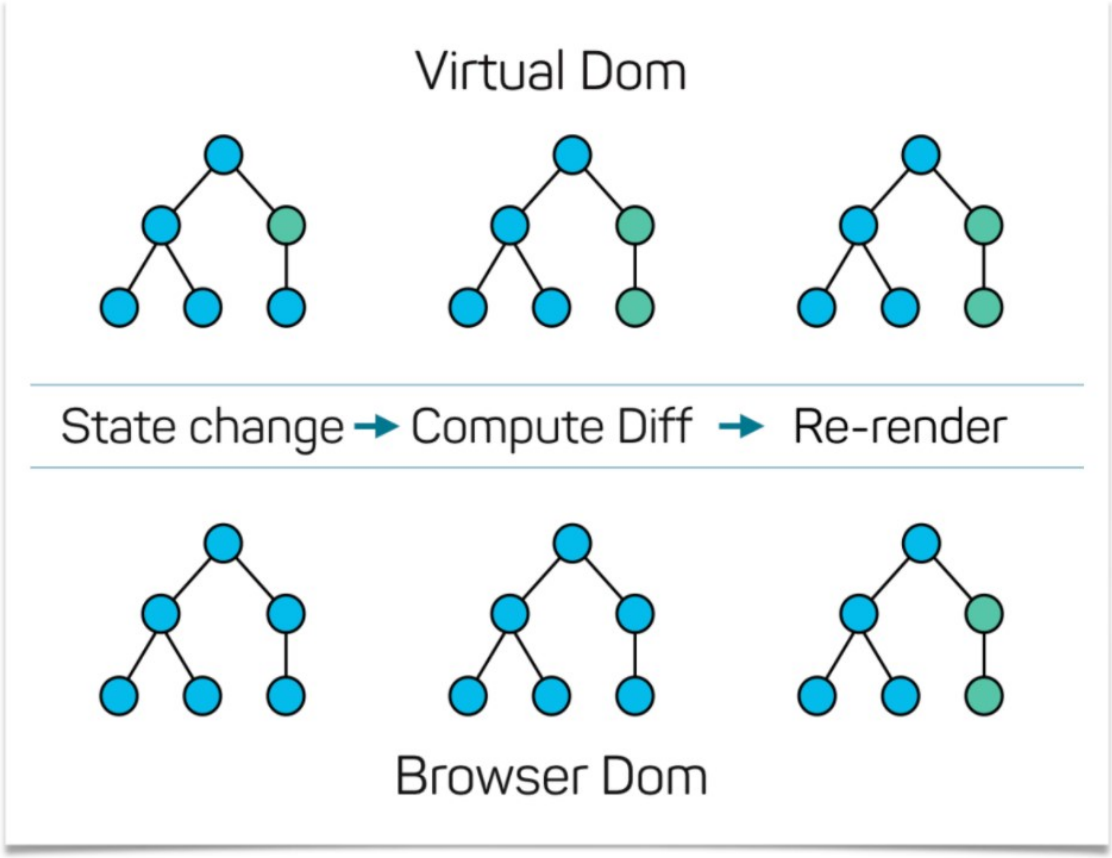
A React egy deklaratív, effektív, és rugalmas JavaScript könyvtár, felhasználói felületek készítéséhez. Lehetővé teszi komplex felhasználói felületek összeállítását izolált kódrészletekből, amiket komponenseknek hívunk. Mivel a komponens logikáját sémák helyett JavaScriptben írjuk, könnyedén lehet adatot mozgatni az alkalmazásban, és így az állapotok a DOM-on kívül maradnak. [6]

A komponenseket a React saját template nyelvében, JSX-ben írhatjuk. Egy komponenst egy JavaScript osztály, vagy függvény valósít meg. A render metódusban HTML-szerű kódot írhatunk, amelyet JavaScript kifejezéseket használva egészíthetünk ki, vagy módosíthatunk.

A komponensek egymásba ágyazhatók, így a fájlrendszerünk átlátható, szépen strukturált maradhat, valamint egy adott komponenst többször is felhasználhatunk. Éppen ezért érdemes minél kisebb logikai egységeket létrehozni.

A React egy úgynevezett virtual DOM (Document Object Model) koncepciót (2. ábra) alkalmaz, amelynek segítségével a tartalom változása esetén újra tudja tölteni kizárólag a módosult részt, nem szükséges az egész oldalt lekérni újra a szerverről. Ezzel teljesítményben képes felülmúlni a vetélytársait, amennyiben ez a funkció kihasználásra kerül.

A virtual DOM-on belüli navigáció a React saját Router komponensével történik. Ezt általában a kiinduló komponensben helyezzük el.



2. ábra [9]

A hátránya egy teljes keretrendszerrel szemben az, hogy itt a kiegészítő szolgáltatásokat nekünk kell összegyűjteni és integrálni az alkalmazásba, ugyanakkor ez egy nagyfokú szabadságot is biztosít a fejlesztőnek.

2.3.2 React Bootstrap

A React Bootstrap az egyik leggyakrabban alkalmazott dizájn keretrendszer, a Bootstrap átültetése React webalkalmazásokhoz.

A keretrendszer egy egységes dizájnt biztosít az alkalmazáson belül. Mindezt úgy teszi, hogy a fejlesztőnek egy egyszerűbb alkalmazásnál nincs is szüksége arra, hogy CSS kódot írjon.

A React Bootstrap React komponenseket biztosít, amelyet úgy paraméterezhetünk fel, mint azok eredeti HTML megfelelőit. Ezek mellett kiegészítésként egyéb paramétereket is tudnak fogadni a komponensek, például az elrendezéshez, vagy a reszponzív megjelenéshez kapcsolódóan.

2.4 Docker

A Docker jelenleg az egyik legelterjedtebb konténer framework. A konténer egy teljesen elkülönített, virtualizált környezetet biztosít az alkalmazásainknak, amelynek a gazdaszámítógép szolgáltat erőforrásokat. A konténerek nagyon hasonlítanak a virtuális gépekhez, azonban itt csak a felhasználói tér virtualizációjáról van szó, a többi erőforrás közös.

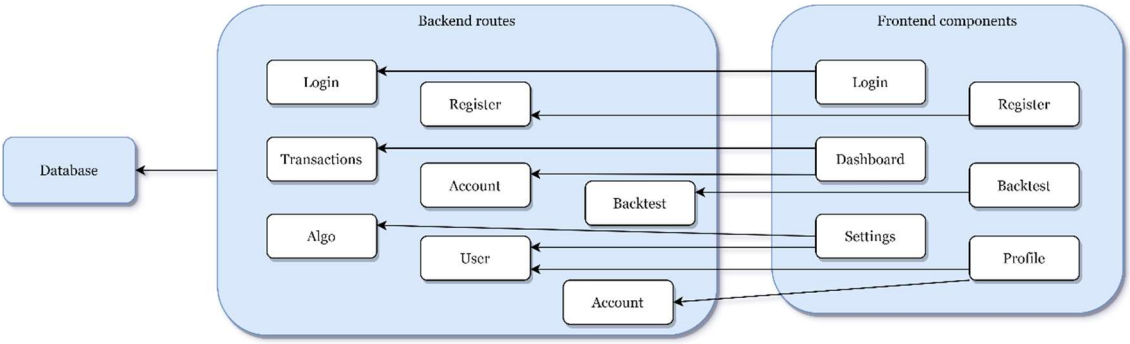
A Docker alapvetően három részből áll: a démon, a kliens, és egy REST API. A démon a host gépen fut és parancsokat fogad a klienstől, az API segítségével pedig közvetlenül kommunikálhatunk a démonnal. Mi felhasználóként persze elsősorban a klienst fogjuk használni, ennek adunk ki különböző parancsokat.

A konténerek definiálására az ún. dockerfile szolgál, ez instrukciókat tartalmaz arra vonatkozóan, hogyan épüljön fel a konténer. Ezekkel a parancsokkal például előírhatjuk az operációs rendszer verzióját, bizonyos Linux csomagok installációját, portok megnyitását, és még rengeteg egyéb dolgot.

A dockerfile elkészülte után jöhet a build lépés, ami egy Docker image-et fog létrehozni. A Docker image-re úgy kell gondolnunk, mint egy kizárólag olvasható template-re, ami a fájlban meghatározottak alapján létrejövő környezetet tartalmazza. Ha ehhez hozzácsatoljuk az alkalmazásunkat és egy írható/olvasható fájlrendszert, létrejön a Docker konténer. Amint ez kész, az alkalmazásunk bármely környezetben változtatás nélkül futtathatóvá válik. [7]

3 Tervezés

3.1 Az adatbázis elérésének útjai a grafikus felületről



3. ábra

3.2 Algoritmus

Az algoritmusom alapötlete onnan jött, hogy a múltbéli adatokat visszanézve azt vettem észre, hogy két jelentősen eltérő intervallumú mozgó átlag metszéspontja sok esetben trendfordulót jelez a hosszabb távú (legalább 1 órás) grafikonokon.

A kötésekhez zárási feltételt nem adok meg, ugyanis az algoritmusom a hosszú egyirányú trendeket próbálja kihasználni, amihez tökéletesen illik a csúszó stop eszköz, ami leköveti a piac mozgását és helyes beállítás esetén csak a fordulópontnál zárja a kötést.

3.2.1 Szükséges fogalmak

A forex kereskedésben a grafikonokon gyertyákkal ábrázolják a piac mozgását. Egy gyertya jelzi a grafikon alap időegysége (pl. 1 óra) alatti változást. A változás mértéke, azaz a kezdő és a befejező időpont árszintje közötti különbség a gyertya nagysága, emellett a kiugró szélsőértékeket kanócokkal illusztrálják.

A mozgó átlag a legutolsó X darab gyertya (általában) záró árszintjének az átlaga, ahol X a mozgó átlag intervalluma. Az idő előrehaladtával ez egy vonalgrafikont rajzol ki, így lehet kettőnek a metszéspontjáról beszélni. Minél rövidebb az intervallum, annál gyorsabban követi az aktuális árszintet a mozgó átlag, így mondhatjuk azt, hogy a rövidebb metszi át a hosszabbat.

A stop egy kötéskor megjelölt árszint, ami garantál egy bizonyos maximális veszteséget az adott kötésből. Amennyiben a piac nekünk kedvezőtlen irányba mozdul és eléri a stop szintet, a kötés lezárul. Ennek egy variánsa a csúszó stop, amely ha a piac nekünk kedvező irányba mozdul, akkor ezt a mozgást egy megadott távolságból leköveti, így ha elég ideig halad egy irányba az árszint, akkor már a stop szintjén történő zárás is profitábilis lesz.

3.3 Backend

A webszerverem egy tisztán JavaScript nyelven íródott Node.js alkalmazás, ami az Express.js middleware-rendszerét használja. Az adatokat MongoDB adatbázisban tárolom.

3.3.1 Felépítés

3.3.1.1 Adatbázis

Az adatbázis működéséhez modellekre van szükség, melyeket Mongoose sémákkal írhatunk le.

A felhasználó sémája a **User**, amely tartalmazza a nevét, az email-címét, a telefonszámát, a kereskedéshez szükséges API-kulcsát, valamint a számlájának az azonosító számát, amivel kereskedni szeretne. Kapcsolódik a sémához még egy algoritmus objektum, ami a felhasználó által alkalmazott beállításokat reprezentálja. Ez a modell, és ezen belül is az email-cím szolgál az azonosítás alapjául. Éppen ezért a jelszó titkosított megfelelőjét is itt tárolom egy *hash* és egy *salt* értékpár segítségével.

Az algoritmusok sémája az **Algo**, amelyben tárolom például a kereskedni kívánt devizapárt, a grafikon időegységét, a stop szint távolságát, a mozgó átlagok intervallumát és egyéb szükséges paramétereket. Az összes paraméter rendelkezik alapértelmezett értékkel, így a regisztráció pillanatától kezdve mindenféle beállítás nélkül használhatóvá válik az alkalmazás.

Az adatbázist össze kell kötni a programmal, ezt a feladatot egy konfigurációs fájl látja el, ami a felkonfigurált *mongoose* objektumot adja vissza. Ezt importálva lehet hozzáférni az adatbázisműveletekhez.

3.3.1.2 Express szerver

Az index.js fájl tartalmazza a szerver konfigurációját. Itt van az alkalmazás belépési pontja, itt hivatkozom be az útvonalakat tartalmazó, valamint az authentikációs stratégia is itt kerül leírásra.

3.3.1.3 Elérési útvonalak

A routes.js fájl tartalmazza, hogy milyen útvonalakon fogad kérést az alkalmazás, valamint itt iratkoztattam fel ezekre az útvonalakra a szükséges middleware-eket is.

3.3.1.4 Middleware-ek

A kérések kiszolgálása ezeken a fájlokon keresztül történik meg. A feladatok több részre bontásával több ilyen egységet is többször fel tudtam használni.

3.3.1.5 Algoritmus

Maga az algoritmus az oanda.js fájlban található. Az itt található kód figyeli a piac mozgását és reagál az adatbázisban található algoritmus-paraméterek függvényében.

3.3.1.6 Historikus teszter

Annak ellenére, hogy az algoritmus megegyezik az élesben használttal, megvalósításban jelentősen eltér, hogy a régi adatokon tesztelés funkcióját is ellássa, ezért ez külön fájlba kerül. Az eltérés abból adódik, hogy a régi adatokon futó algoritmus nem tudja használni a bróker cég szolgáltatásait.

3.3.2 API végpontok

A webszerverem 9 végponton várja a beérkező kéréseket.

A felhasználói authentikációval kapcsolatos végpontok a **/login**, amely a bejelentkezés folyamatát vezérli, valamint a **/register**, amely a felhasználói regisztáció feladatait látja el. Az authentikáció JWT token alapú, így kijelentkezésre nincs szükség, ehelyett egy viszonylag szűk, kétórás lejárati időt adtam meg az authentikációs tokennek.

A lekérdezésekhez a **/account**, a **/algo**, a **/user** és a **/transactions/:count** végpontok használhatók. Ezek sorrendben a felhasználó számlaadatait, algoritmus-paramétereit, profil adatait, valamint a legutóbbi megadott számú végrehajtott tranzakciót szolgáltatják.

Adatok mentéséhez és módosításához a **/algo/save** és a **/user/save** végpontok állnak rendelkezésre.

A historikus teszter futtatásához a **/backtest** útvonalra kell elküldeni a paramétereket, amelyeket felhasználva a bekötött middleware lefuttatja a tesztet és visszaküldi a végeredményt.

3.4 Frontend

A webes grafikus felületem egy egyszerű React alkalmazás, amely a backendem által nyújtott API szolgáltatásainak a kiaknázásához szükséges.

A fő komponens az *index.jsx* fájlban található, ez az alkalmazás belépési pontja. Ez a komponens többek között felelős az authentikáció helyes működéséért, a kliensoldali útvonalválasztásért, itt valósítom meg a komponensek közötti kommunikációt is.

Az útvonalválasztó köti össze az API szolgáltatásait a megfelelő komponensekkel. Ennek a mikéntje a fejezet elején látható. (3. ábra)

4 Implementáció

4.1 Algoritmus

Az algoritmusom alapvető fejlesztése a MetaTrader 4 platformon történt. A lefedett időintervallum, amire optimalizáltam a kódomat 2016.12.01-2019.12.01. Azért választottam ezt az intervallumot, mert ennél több ideig nem nagyon tud működni egy kód, valamint az ezeken belüli intervallumokon is tökéletesen látszik, hogy mikor milyen sikerességgel működött a robotom.

4.1.1 Kiindulási alap

Az első futó kódom nem csinált mást, mint adott profit- és stop szintekkel, valamint adott kötésegységgel pozíciót nyitott minden alkalommal, amikor a 15 gyertyás mozgó átlag keresztezte a 30 gyertyásat. Amelyik irányba éppen tartott a kisebb intervallumú mozgó átlag a másikhoz képest, abba az irányba nyílt a pozíció.

Ennek egy optimalizált változata egy év alatt el tudott érni akár 200%-os profitot is, azonban más időintervallumon futtatva komoly veszteséget is képes volt generálni. Egy kezdetleges kódnál ez nem meglepő, ugyanis semmiféle biztonsági mechanizmus nincs beépítve, és mindössze egy eseményre reagál.

4.1.2 Fejlesztések

4.1.2.1 MakeOrderWaitLimit

Ahogy néztem a vizualizált eredményeket, egyből szemet szúrt, hogy amikor egymáshoz közel mozog a két mozgó átlag, akkor teljesen irrelevánsak a metszéspontok.

4. ábra

Ez a fenti grafikonon is jól látszik, ahol a zöld vonal a nagyobbik és a piros a kisebbik mozgó átlag. Itt egyedül a nyíllal jelölt metszéspont hordozza azt az információt, amire nekem szükségem van. Az azt követő metszéspontnál történő eladás (lefelé nyitás) könnyen veszteséggel zárhatna, hiszen rögtön utána újra következik egy metszéspont, ami szerint már venni kéne. Látható, hogy ez nehézségekhez vezethet.

Erre egy olyan megoldást találtam ki, hogy egy globális flagtől függően tudjon az algoritmus csak pozíciót nyitni. Ezt a flag-et minden sikeres nyitás után hamisra állítom. Akkor billen vissza igazba, hogyha a két mozgó átlag egy adott távolságnál jobban eltávolodott egymástól. Nagyjából 50 és 150 pont közé érdemes ezt a limitet felvenni. Ezt az eltávolodást a *checkMAs()* függvényben ellenőrzöm, és itt is állítom vissza a flag-et igazra, ha teljesült a feltétel.

4.1.2.2 Csúszó stop

5. ábra

A nyíllal jelölt metszéspontnál történő vétel esetén beállít az algoritmus egy csúszó stopot a felhasználó által megadott távolságra. Ezzel a távolsággal is lehet például szabályozni, hogy mekkora kockázatot kíván valaki vállalni.

Ez a csúszó stop a meghatározott távolságra követi az árszintet, amíg egyszer csak el nem éri a piac az aktuális stop szintet. Ekkor a kötés lezárul.

A fenti esetben (5. ábra) az algoritmus jól tudta kihasználni a viszonylag hosszú ideig tartó egy irányú mozgást, aminek az eredménye a zöld nyíllal jelölt profit. Ha előre meghatározott zárási szintet használna az algoritmus, akkor ezeket a trendeket nem használná ki.

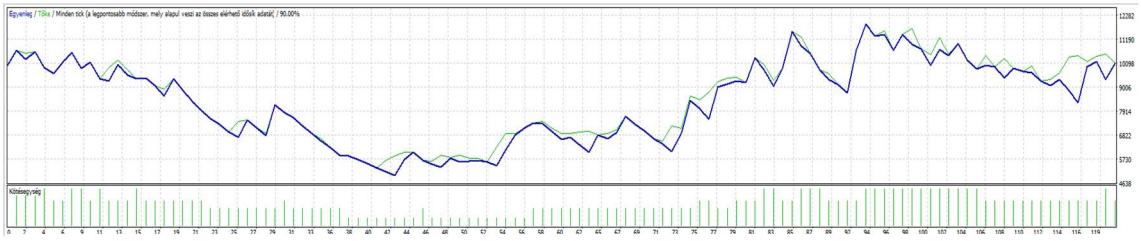
Ennek a módszernek a hátránya az, hogy a rövidebb ideig tartó trendeknél, valamint oldalazó piac esetén kiszámíthatatlanná teszi a működést, viszont hosszú távon

ezeket az esetleges kisebb veszteségeket kompenzálják a nagyobb nyereségek, amit a csúszó stoppal lehet elérni.

4.1.2.3 Dinamikus kötésegység

Minden kötésnél a kötésegység határozza meg, hogy mekkora értékben adunk el, vagy veszünk a piacon. Minél nagyobb ez az érték, annál nagyobbat lehet nyerni és veszíteni is az üzleten.

A kötésegység előre definiált szinteket elérve változik. A szorzókat tekintve meglehetősen konzervatívan álltam a dologhoz, hiszen minden egyéb tekintetben ez egy magasabb kockázatú algoritmus. Míg amikor a kezdeti tőke alatt van az aktuális egyenleg, akkor gyorsan csökken a kötésegység értéke, addig a kezdeti tőke fölött rendre 50-30-20-20-20%-kal növelem azt. Ezzel sikerült elérnem, hogy az alábbi időintervallumban ne menjen el az összes pénz, mint egy korábbi változatban, hanem szépen lassan szedje össze magát és végeredményben ne termeljen veszteséget. Az ábra alsó sorában lévő zöld vonalak magassága reprezentálja a kötésegység nagyságát



6. ábra

4.1.3 Végeredmény

Az alábbi grafikonon tökéletesen látszik, hogy ez egy kifejezetten bátor algoritmus, ugyanakkor az is látszik, hogy ez hosszútávon megtérül.

7. ábra

4.1.4 Kiegészítés

Az előző bekezdést pontosítva ez az algoritmus a készítésének idejében volt ennyire sikeres, mára a már korábban említett avulás jeleit mutatja.

4.1.5 Az elkészült algoritmus MQL4 nyelven

Az algoritmusom néhány fontosabb szerepet játszó függvényének megvalósítása:

```
void OnTick()
{
   if(!isActiveTime()) return;
   setLots();
   int ticket = 0;
   updateMAs();
   int cross = checkCross();
   double lastAwesome = awesome;
   awesome = iAO(Symbol(), Period(), 0);
   bool b = false;
   if(!OrderSelect(lastTicket, SELECT_BY_TICKET)) b = true;
   if (TimeCurrent() - OrderOpenTime() > PeriodSeconds() || b)
   {
      if (makeOrder)
      {
         if (cross > 0)
         {
            ticket = OrderSend(Symbol(), OP_BUY, Lots, Ask, 5,
                        Bid-StopLoss*Point, NULL,
                        "", MAGICMA, 0, Green);
         }
         else if (cross < 0)
         {
            ticket = OrderSend(Symbol(), OP_SELL, Lots, Bid, 5,
                        Ask+StopLoss*Point, NULL,
                        "", MAGICMA, 0, Red);
         }
         if (ticket < 0) Alert("Error Sending Order!");
         else if (ticket > 0)
         {
            lastTicket = ticket;
            makeOrder = false;
         }
      }
   }
   for(int i = 0; i < OrdersTotal(); i++)
   {
      if (!OrderSelect(i, SELECT_BY_POS, MODE_TRADES)) break;
      if (OrderSymbol() == Symbol() && OrderMagicNumber() == MAGICMA)
      {
         if (OrderTicket() > 0)
         {
            TrailingStop(OrderTicket());
         }
      }
   }
}
```

```
void updateMAs()
{
   MAs[0] = iMA(Symbol(), Period(), MA1, 0, MODE_SMA, PRICE_CLOSE, 1);
   MAs[1] = iMA(Symbol(), Period(), MA2, 0, MODE_SMA, PRICE_CLOSE, 1);

   lastMAs[0] = iMA(Symbol(), Period(), MA1, 0, MODE_SMA, PRICE_CLOSE,
2);
   lastMAs[1] = iMA(Symbol(), Period(), MA2, 0, MODE_SMA, PRICE_CLOSE,
2);
}

int checkCross()
{
   double a = abs(MAs[1] - MAs[0]);

   if (!makeOrder && a > MakeOrderWaitLimit ) makeOrder = true;

   int ret = 0;
   if ((lastMAs[0] < lastMAs[1]) && (MAs[0] > MAs[1])) ret = 1;
   else if ((lastMAs[0] > lastMAs[1]) && (MAs[0] < MAs[1])) ret = -1;
   return ret;}
}

double balMuls[] = { 0.6, 0.8, 1, 2, 3, 5, 7, 10 };
double lotMuls[] = { 0.25, 0.5, 0.7, 1, 1.5, 1.95, 2.34, 2.808, 3.3696
};

void setLots()
{
   for(int i = 0; i < 8; i++)
   {
      if (AccountBalance() < startBalance * balMuls[i] &&
                        Lots > startLots * lotMuls[i])
         Lots = startLots * lotMuls[i];
   }

   for(int i = 0; i < 8; i++)
   {
      if (AccountBalance() >= startBalance * balMuls[i] &&
Lots < startLots * lotMuls[i + 1])
         Lots = startLots * lotMuls[i + 1];
   }
}
```

Ezen függvények írják le a fejezetben korábban kifejtett funkciók működését.

4.2 Backend

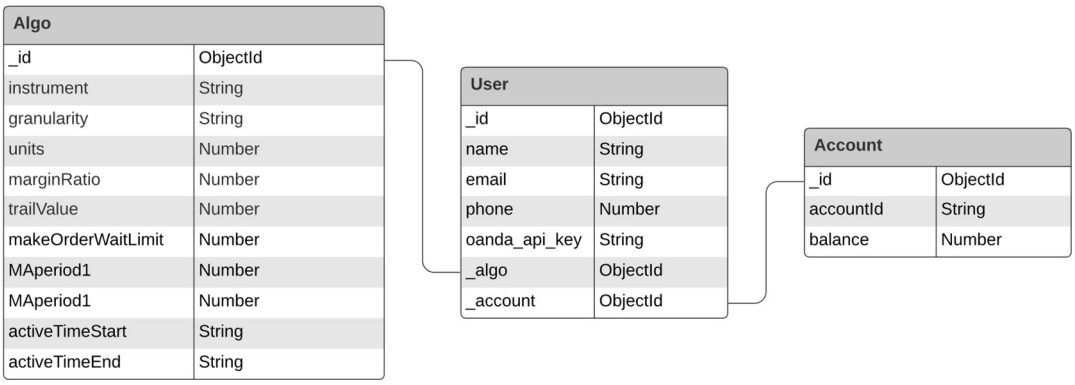
A projektem szerveroldali része egy Node.js-ben készített Express webszerver, amely egy flexibilisebb formában megvalósítja a kötött, platformspecifikus MQL4 nyelven íródott algoritmust, valamint ennek a használatát REST API-n keresztül elérhetővé teszi a felhasználóknak.

4.2.1 Adatbázis

Adatbázisként a Node.js platformhoz legkézenfekvőbb MongoDB-t használtam. A használat megkönnyítése érdekében felhasználtam a Mongoose nevű NPM csomagot.

4.2.1.1 Konfiguráció

Létrehoztam akorábbi Tervezés fejezet Adatbázis szekciójában specifikált alapvető modell fájlokat, melyekben alapértelmezett értékekkel is elláttam minden lehetséges paramétert, hogy a felhasználó akár már a regisztráció után közvetlenül is használni tudja az alkalmazást. A végleges modellek a következők lettek:



8. ábra

A Mongoose ezek alapján a modellek alapján tudja létrehozni a kollekciókat. Továbbá ezek a modellek adnak egy struktúrát az adatbázisban egyébként megkötések nélkül létrehozható dokumentumoknak. Ezzel a lekérdezések kiszámíthatóbbak és egyszerűbbek lesznek.

Az authentikáció alapjait is itt fektettem le. Ehhez a Passport-Local-Mongoose nevű NPM csomagot használtam. Ez a csomag a Mongoose-zal együttműködve, beépülő

```
User.plugin(passportLocalMongoose);

User.virtual("password")
  .set(function(password) {
    this._password = password;
    this.salt = this.makeSalt();
    this.hashed_password = this.encryptPassword(password);
  })
  .get(function() {
    return this._password;
  });
```

modulként tudja megvalósítani a felhasználói authentikációt. A bekötését az alábbi módon oldottam meg a *User.js* fájlban, ahol a *User* változó maga a modell:

A továbbiakban ha egy felhasználót hitelesíteni szeretnék, elég meghívni a *passport.authenticate()* függvényt a megfelelő paraméterekkel. Továbbá itt rögtön megvalósítottam a biztonságos jelszótárolást is.

4.2.1.2 Lekérdezések

A lekérdezéseket legegyszerűbben kollekciónként tehetjük meg, amihez a kívánt adatokat reprezentáló modellből létre kell hozni egy példányt, majd azon tudjuk meghívni a Mongoose lekérdező függvényeit. Használatát tekintve hasonlít egy relációs adatbázishoz.

Álljon itt példként az alábbi kód, amely egy felhasználó algoritmusának az új adatokkal való frissítését végzi el a *request* objektumon kapott adatok alapján.

```
const Algo = new Algo();

Algo.findOneAndUpdate(
      { _id: req.user.username.algo },
      req.query,
      {
        useFindAndModify: false,
        new: true,
      },
      (err, result) => {
        if (err)
          return res
            .status(500)
            .json({ message: "Algo update failed", error: err });
        if (result) return res.json({
            message: "Algo update successful"
        });
        return res.status(500).json({ message: "Algo update failed" });
      }
    );
```

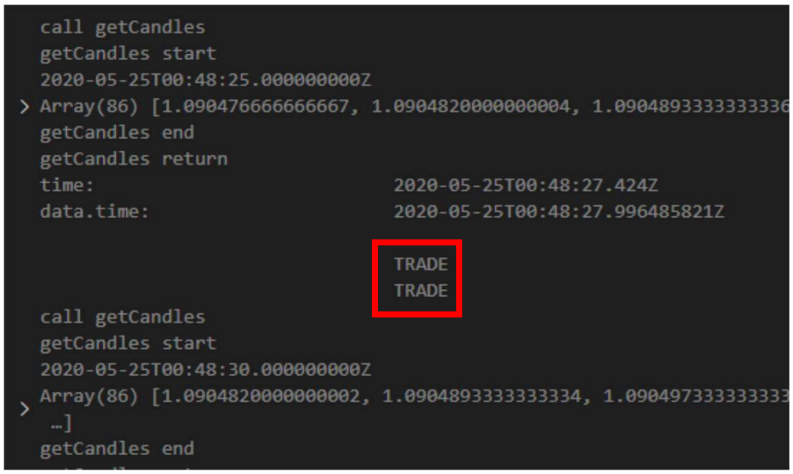
4.2.2 Az algoritmus átültetése

Ahhoz, hogy egy saját szerveren, egy saját API-t kiszolgáló kereskedő algoritmust tudjak létrehozni, a már meglévő algoritmust portolni kellett JavaScriptre.

Ez a feladat bizonyult a projekt legnagyobb kihívásának, ugyanis ahhoz, hogy ki tudjam aknázni a JavaScript és azon belül is a Node.js teljesítménybeli előnyeit, egy korábban teljesen szinkron módon futó algoritmust kellett implementálni egyszálas, aszinkron környezetben.

A probléma nehézségét az adta, hogy az adatok, amelyeket felhasználva az algoritmusom működni tud, mind egy távoli API-ról érkeznek aszinkron módon.

Az eredeti elképzelés szerint egy adatfolyamra regisztrálva futott volna az algoritmus, hogy teljes egészében át tudjam venni az MQL4-es működést. Ezzel kifejezetten meggyűlt a bajom, ugyanis az aszinkron működés, és a stream pufferes működése miatt állandóan egymásra csúsztak a függvényhívások, így egyszerre többet kötött (9. ábra), mint kellett volna, vagy éppen kihagyott gyertyákat.



9. ábra

Megpróbáltam flagekkel és *setTimeout()*-okkal kezelni a problémát, de végleges megoldást nem jelentett. A kihagyásokat sikerült megszüntetni, de a többszöri kötés megmaradt. Így jött az ötlet, hogy amint egy kötéshez érünk, vágjuk is el a streamet. Ezzel megint sokat javult a helyzet, de alkalom adtán még mielőtt lekapcsolódtam volna az adatfolyamról, még belehívott egy pufferből érkező adat hatására a kötésbe. Néhány óra ezzel való próbálkozás után kellett megvilágosodnom, hogy semmi szükségem a streamre, bőven elég időzített hívásokat intézni a szerver felé.

A felismerés után körülbelül 10 sorból sikerült is megoldani az alkalmazás legnagyobb problémáját, amivel a kezdetek óta küzdöttem. Egyedül az időzítéssel volt egy kis probléma, ugyanis egy rosszul megválasztott intervallummal kimaradhattak gyertyák, amik egy 1 órás grafikonon már nagy problémát jelentenek. A megoldás az lett, hogy megtartottam a stream miatt bevezetett flageket, és egy gyertya lefutása alatt 5-ször hívok rá az *onData()* függvényre, ami a korábbi MQL4-es *onTick()* megfelelője. Ez azért jó, mert elég időt hagy az API-hívások lefutásához, de nem késik sokat az adatok elemzésével akkor se, ha pont gyertyaváltás előtt 1 másodperccel próbálkozik a hívással.

4.2.3 Az Oanda API elérése

Az API-hoz találtam egy Simple FxTrade nevű NPM csomagot, amely egy burkoló osztálykönyvtár az API felett. Ez ugyan nem különösebben jól dokumentált, viszont egy kis próbálkozás és *console.log()*-olás után átlátható, és az alap funkciókat jól ellátja.

Az alapvető működési elve egy állapotgéphez hasonlítható, ugyanis használat előtt fel kell konfigurálni, amikor is a felhasználó API kulcsát, és a használni kívánt, brókernél nyitott számlának az azonosítóját kell megadni. Ezek után a legtöbb lekérdezést egy-egy egyszerű függvényhívással is végre lehet hajtani.

A probléma ott kezdődött, amikor az API-kéréseket specifikálni kellett. Az én esetemben a gyertyák lekérdezése nem volt kivitelezhető, ugyanis nem lehetett megadni, hogy pontosan hány gyertyára van szükségem, pedig az API támogatja ezt a funkciót. Így itt kénytelen voltam axios (NPM csomag nyers HTTP kérések lebonyolításához) segítségével egy nyers API-hívást kidolgozni. Összességében így is sok fölösleges kódolást spóroltam meg, de tekintve, hogy mennyi idő ment el az osztálykönyvtár működésének a megértésére, nem biztos, hogy ajánlanám bárkinek is a használatát.

4.2.4 Többfelhasználós működés

Ahhoz, hogy ez az alkalmazás rendes szolgáltatásként működhessen, egyszerre több felhasználót is ki kell tudnia szolgálni.

Az első elképzelésem ennek megvalósításával kapcsolatban egy ahhoz hasonló rendszer volt, mint amit a relációs adatbázisok tranzakciókezelésénél használnak. Ehhez „lock”-ok gyanánt újabb globális változókat vezettem be. Ezeknek elméletben működniük kellene, azonban azzal, hogy aszinkron kérések hatására állítom be azokat, meglehetősen kiszámíthatatlanná válik a működésük. Ennek kiküszöbölésére több kísérletet is tettem, azonban végül kénytelen voltam egy ehhez csak apró részletekben hasonlító megoldást kitalálni.

A végső jó ötletet a Simple FxTrade csomagoló osztály adta. Egészen pontosan annak az állapotgép-szerű működése. Erre alapozva kidolgoztam egy olyan működést, amely az aszinkron környezetben – annak minden előnyét megtartva – képes állapotgépként működni. Mindezt úgy, hogy a felhasználók saját algoritmusainak a működése között garantáltan nincs átfedés.

A megvalósítás alapja egy közönséges tömb, amelybe induláskor, valamint a felhasználói adatbázis frissülése esetén beolvasom a felhasználók minden szükséges adatát, így innentől sokkal gyorsabban, memóriából tud dolgozni a program. A beolvasás során a három összekapcsolt kollekciót rögtön össze is fésülöm, így az adatok elérése abszolút intuitív, objektumorientált szemléletet követ.

A felhasználókat az email-címük alapján azonosítom. Ez az a paraméter, amely behálózza az egész alkalmazást, hiszen minden függvény megkapja, amely az algoritmus futásához közvetlenül kapcsolódik. A függvények ezzel a paraméterrel indexelik a memóriába beolvasott tömböt, és így érik el az adott felhasználóhoz tartozó specifikus adatokat.

A Simple FxTrade felépítését kihasználva írtam egy saját konfiguráló függvényt, amely szintén az email-cím alapján azonosítja be a felhasználót, majd szerzi be a tömbből a konfigurációhoz szükséges adatokat. Ezt a függvényt minden egyes alkalommal meg kell hívni, mielőtt az algoritmus API-hívást kezdeményez. A konfigurációs függvény lefutása után minden esetben közvetlenül következik a HTTP kérés konfigurálása, így garantálható, hogy az a megfelelő adatokkal fog történni.

4.2.5 Felhasználói authentikáció

A felhasználói belépést jelszó alapján kell szabályozni, azonban ahhoz, hogy ne kelljen minden egyes kérés esetén újból bejelentkezni, szükség van egy rendszerre, amely a kliens oldalon bejelentkeztetve tartja a felhasználót.

4.2.5.1 Lokális authentikáció

Ahhoz, hogy a bejelentkezési oldalon megadott email-cím és jelszó alapján ellenőrizni tudjam a felhasználó hitelességét, össze kell vetnem a jelszót az adatbázisban található enkriptált jelszóval. Erre nyújt egyszerű megoldást a Passport lokális stratégiája, amely mindezt egyetlen függvényhívással elvégzi.

4.2.5.2 Token alapú authentikáció

Miután a felhasználót a lokális stratégia hitelesnek találta, a hitelesítő függvény callback függvényében a Passport JWT stratégiájának segítségével beállítok egy tokent, amivel később azonosíthatja magát a felhasználó.

A tokent a felhasználó adatbázisbeli azonosító kódjából és email-címéből, valamint egy titkos kulcsból generálom. Ezt a tokent a belépés után a szerver visszaküldi a kliens oldalnak Később ez alapján könnyen beazonosítható lesz a felhasználó.

Onnantól, hogy a kliens oldal megkapta az azonosító tokent, minden kérés fejlécében meg kell azt adni, mint authentikációs mezőt, máshogy nem lehet használni a szerver szolgáltatásait.

Ennek a fajta authentikációnak és az Express middleware-rendszerének a nagy előnye, hogy a stratégia alább bemutatott módon való konfigurálása esetén amint beazonosít a szerver egy felhasználót, annak azonnal minden adatát ki tudom olvasni az adatbázisból, azonnal strukturálni tudom és hozzá tudom adni a *request* objektumhoz. Ezzel nem kell a middleware-ekben extra adatbázislekéréseket végrehajtanom és az eredményt a *response* objektumon hordozni.

```
passport.use(
  new JWTStrategy(
    {
      jwtFromRequest: ExtractJWT.fromAuthHeaderAsBearerToken(),
      secretOrKey: "awrt2",
    },
    function (jwtPayload, done) {
      return User.findById(jwtPayload.id)
        .populate("_account")
        .populate("_algo")
        .then((user) => {
          return done(null, user);
        })
        .catch((err) => {
          return done(err);
        });
    }
  )
);
```

4.2.6 Útvonalválasztás és middleware-ek

A szerverre beérkező kéréseket a kérésben megadott útvonal szerint az Express.js middleware-eken keresztül dolgozza fel. Ezek konfigurálásához szükség van a szerver példányára, amire fel lehet iratkoztatni a szükséges middleware-eket.

Az egy útvonalra feliratkoztatott middleware-ek lánc-szerűen kapcsolódnak egymás után. A middleware-ek egy-egy függvényt exportálnak. Alapértelmezetten ezen függvények három objektumot kapnak paraméterként. A *request* objektum tartalmazza a beérkezett kérés paramétereit. A *response* objektum szolgál arra, hogy a middleware-ek

lánca kialakítsa a HTTP választ. Konvenció szerint ezen objektum *locals* paraméterén keresztül tudnak a lánc elemei kommunikálni az utánuk következőkkel. A *next* paraméter egy függvény, amelynek meghívására a az említett három paraméter átadásával meghívódik a lánc következő eleme.

A fent leírt rendszerbe épül be tökéletesen a Passport authentikációs szolgáltatás, ugyanis azon útvonalak esetén, ahol felhasználói azonosításra van szükség, elég csak a csomag által szolgáltatott middleware-t bekötni a lánc elejére és ez biztosítja is, hogy csak hitelesített felhasználók férhetnek hozzá az utána következő tagok funkcióihoz.

4.2.7 Historikus teszter

A múltbéli adatokkal dolgozó algoritmusnak pontosan olyan működést kell mutatnia, mint az élő adatokkal dolgozónak. Ez alapján egyértelműnek tűnne az eredeti algoritmus kódjának újrahasznosítása, azonban az aszinkron környezet, az állapotgép-szerű működés, valamint a szükséges HTTP kérések miatt rendkívül komplikálttá válna a kód, ha azt úgy írnám át, hogy a teszterrel is képes legyen működni. Éppen ezért, valamint később látható egyéb okok miatt egy külön fájlt szenteltem neki.

Ebben először is inicializálom a felhasználót, aminek az adatait a tesztert kezelő middleware-en keresztül kapom meg. A következő lépés a szükséges adatok beszerzése az Oanda szerveréről. Ehhez meg kell vizsgálni a megkapott időtartamot, amin a felhasználó futtatni szeretné a tesztet, ugyanis az Oanda API-ja nem támogatja azokat a kéréseket, melyeknél a gyertyák száma meghaladja az 5000-et. Ha ilyenre lenne szükségem, akkor a kérést fel kell darabolnom, majd a válaszokat újra összeilleszteni.

Miután megvannak a gyertyáim, a későbbi használat megkönnyítése érdekében mindegyikhez hozzárendelem a két mozgó átlag adott időben felvett értékét. Ez egy újabb eltérés az élő adatokon futó algoritmushoz képest, hiszen ezzel a megoldással itt elég csak egyszer mozgó átlagokat számolni, míg élesben minden új gyertya beérkezésénél újra kell számolni, ami meglehetősen költséges. Többek között ennek a megoldásnak is köszönhető, hogy míg a MetaTrader platformon futtatott ehhez hasonló tesztek hosszú perceket vesznek igénybe, addig az én algoritmusom a legtöbb lekérést másodpercek alatt képes feldolgozni és eredménnyel szolgálni.

Maga az algoritmus a megkapott gyertyákon iterálva, azaz szinkron módon fut végig. Éppen ezért, valamint a több felhasználó hiánya miatt is kellett újraírni a kód egyes

részleteit. Az eredetihez képest más az adatok elérése, valamint a működéshez szükséges biztonsági flagekre sincs már szükség.

Mivel itt nem áll rendelkezésre az Oanda szervere, hogy kezelje a kötéseket, így minden ehhez kapcsolódó szolgáltatást szimulálni kell. Amire szükségem van, az a csúszó stop kezelése, ez alapján a kötések zárása, a margin kezelése, továbbá a lezárt kötések eredményeinek feldolgozása.

A csúszó stop kezelését az eredeti MQL4-es algoritmusban található függvény mintájára készítettem el a szükséges módosításokkal, hogy működjön a teszter modelljével.

A kötések zárásának szükségességét minden iterációban ellenőrzöm. Amennyiben bármelyik kötés elérte a stop szintet, azt törlöm az aktív kötések közül, kiszámolom, hogy milyen eredményt produkált, majd frissítem a margin és a számlaegyenleg értékeket.

A teszter futásának végén HTTP válaszként visszaadom, hogy milyen kötéseket hajtott végre az algoritmus, valamint, hogy mekkora profitot, vagy adott esetben veszteséget generált volna a megadott időintervallumban.

4.3 Frontend

4.3.1 Adatbevitel

Az adatbevitel Formokon keresztül történik. Ezeknek a Formoknak az elemei többek között a beviteli mezők, amelyek közvetlen összeköttetésben állnak az őket tartalmazó komponensek állapotaival. Alább a név mezőn keresztül szemléltetem a fent leírtak megvalósításához szükséges kódot.

```
constructor() {
    this.state = {
      name: String,
    };
    this.handleNameChange = this.handleNameChange.bind(this);
}

handleNameChange(event) {
    event.preventDefault();
    this.setState({name: event.target.value});
}

render() {
    <Form>
          <Form.Group controlId="name">
            <Form.Label>Your name</Form.Label>
            <Form.Control type="text" value={this.state.name}
                          onChange={this.handleNameChange}/>
          </Form.Group>
    </Form>
}
```

A fenti kódban a komponens létrejöttekor inicializálom annak állapotát, majd a névváltozást kezelő függvényt is „bekötöm”. Erre azért van szükség, hogy a *handleNameChange()* függvényből el tudjam érni a komponens állapotát a *this* kulcsszón keresztül. A *render()* függvény állítja elő a HTML kódot, amit a böngésző meg tud jeleníteni.

Az előbb leírtakra azért van szükség, hogy amikor az adatokat már bevitte a felhasználó és szeretné azokot elküldeni, akkor a HTTP kérésünket a komponens állapotát felhasználva rögtön össze tudjuk rakni és mehetnek is a szervernek a szükséges adatok. Az adatok elküldése és a válasz megérkezése után általában szeretnénk egy másik útvonalra átirányítani a felhasználót.

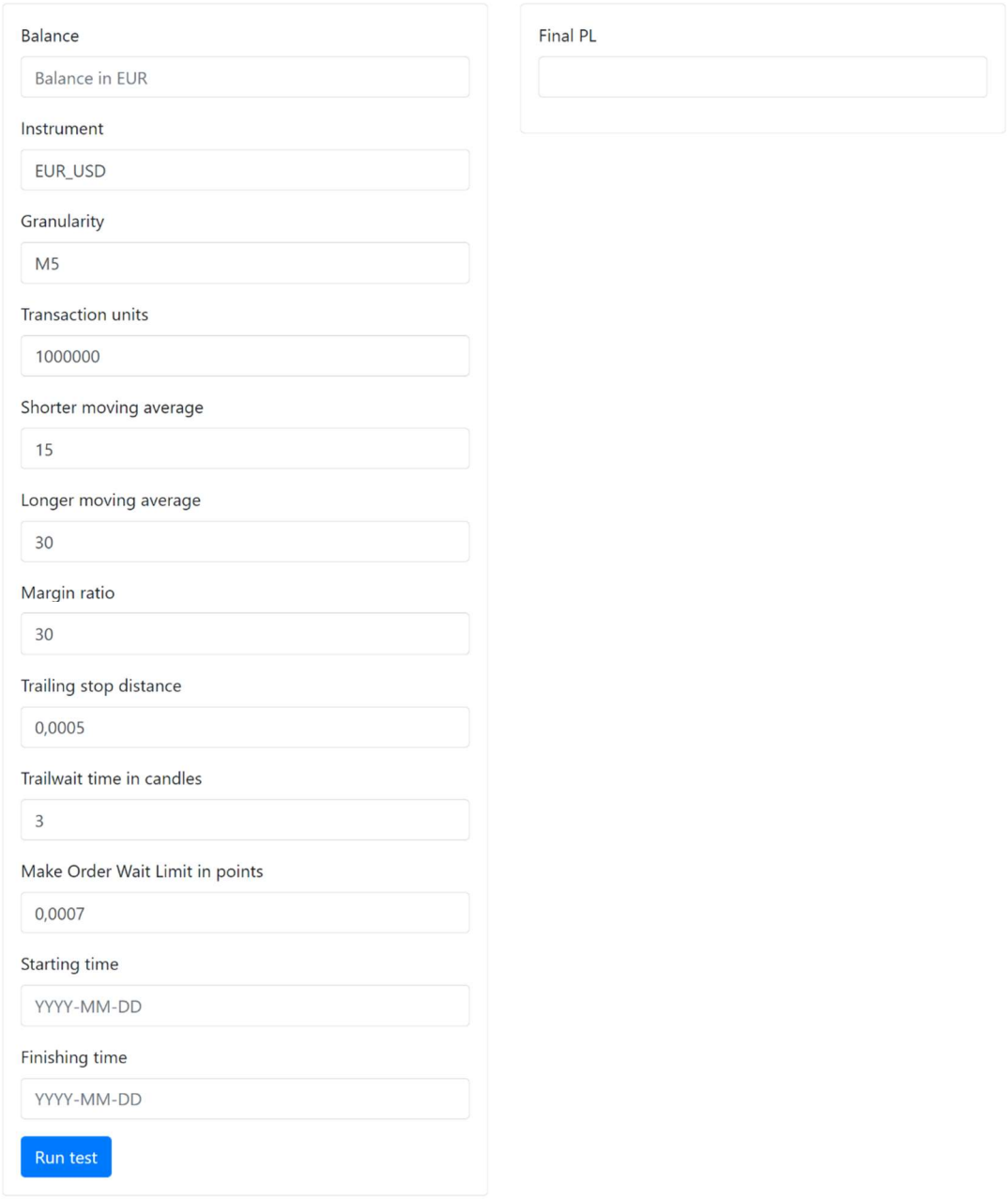
A szerverrel való kommunikáció és az átirányítás az előzőhöz hasonlóan a név paraméteren keresztül bemutatva a következőképpen működik:

```
constructor() {
    this.state = {
      name: String,
    };
    this.handleSubmit = this.handleSubmit.bind(this);
}

handleSubmit(event) {
    event.preventDefault();
    fetch("http://localhost:3001/user/save",
      { method: 'POST', body:
            new URLSearchParams({
              name: this.state.name,
            }),
        headers: new Headers({Authorization: "Bearer " +
                                                this.props.token})
})
      .then(() => this.props.history.push('/dashboard'))
      .catch(console.log);
}

render() {
    <Form onSubmit={this.handleSubmit}>
          ...
    </Form>
}
```

Ugyanúgy szükség van a konstruktor-beli inicializálásra, valamint itt egy szabványos POST kérést is intéztem a szerver felé, amivel elküldtem a bevitt adatokat feldolgozásra. Amennyiben a kérés helyesnek bizonyult és szerveroldalon se történt hiba, a felhasználó átirányításra kerül a *Dashboard* oldalra.

4.3.2 A komponensek

A React alkalmazásom alapvető építőkövei a komponensek. Ezek renderelik ki egy-egy útvonal dizájnját, valamint az adatbevitelért is felelősek. A komponensek az 3. ábra szerint használják a szerverem API-ját.



10. ábra

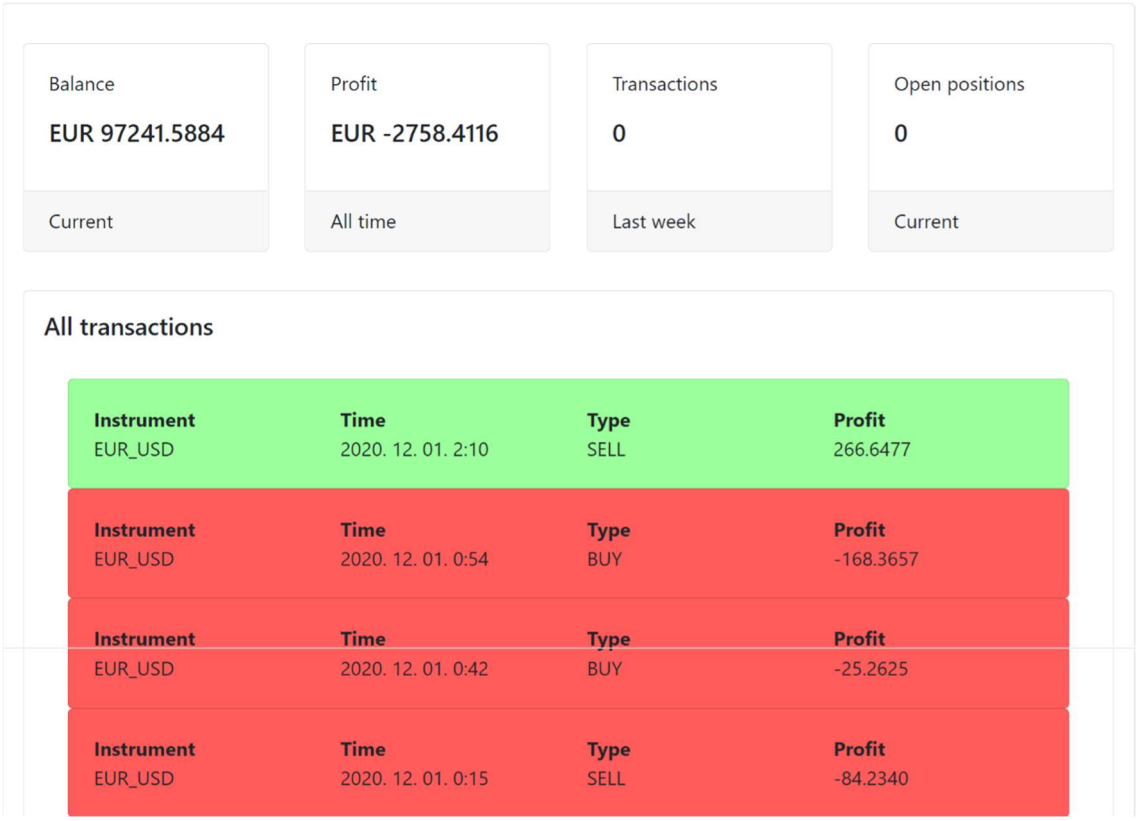
4.3.2.1 Backtest / Settings

A két komponens rendkívül hasonlít egymásra, hiszen ugyanazokat az adatokat kezelik. A Backtest komponens felelős a historikus teszter futtatásához szükséges adatok bekéréséért, míg a *Settings* komponenssel az éles algoritmus paramétereit lehet

megváltoztatni. Megjelenésben a különbség a *Backtest*nél megjelenő eredmény mező, valamint a két dátum mező a form alján, amik a *Settings*nél nem találhatók meg, így a *Backtest*tel szemléltetem a megvalósítást. (10. ábra)

A Formot a könnyebb használhatóság érdekében megnyitáskor feltöltöm a bejelentkezett felhasználó algoritmusának adataival, így neki már csak meg kell változtatnia azt a néhányat, amivel szeretné kipróbálni az algoritmust. Természetesen a kezdő egyenleg, valamint az időintervallum megadása is szükséges. Az eredmény a jobb oldali mezőben jelenik meg.

4.3.2.2 Dashboard



11. ábra

Ez a komponens felelős az algoritmus működésével kapcsolatos legfőbb adatok megjelenítéséért. Továbbá ez szolgál a bejelentkezett felhasználó kezdőoldalaként is.

Mutatja az aktuális egyenleget, az elért profitot, az elmúlt héten lebonyolított tranzakciók számát, valamint a jelenleg nyitott pozíciók számát. Ezeken kívül visszanézhető az algoritmus által végrehajtott összes tranzakció. Ezeknek a megjelenítéséért a *TransactionCard* komponens felel.

Az elmúlt héten történt tranzakciók száma származatott adat, azt a komponens számolja ki a szervertől lekért adatok segítségével.

4.3.2.3 TransactionCard

12. ábra

Ez a komponens szolgál egy lezajlott tranzakció adatainak vizualizálására. A szülő komponenstől paraméterként kapja meg a tranzakció objektumát, ami tartalmaz minden szükséges adatot a bemutatott kártya (12. ábra) elkészítéséhez.

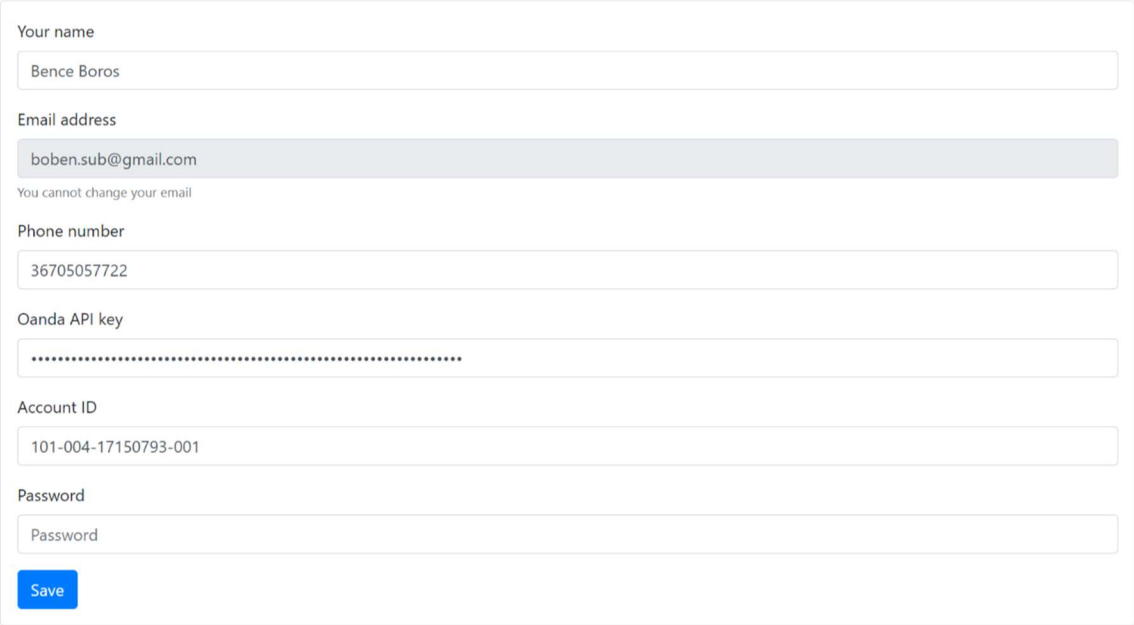
Amennyiben az adott tranzakció profitot termelt, a komponens zöld háttérrel jelenik meg, míg ellentétes esetben piros hátteret kap (11. ábra).

4.3.2.4 Login

Ez a komponens az alkalmazás kezdőképernyője, ha nincs bejelentkezett felhasználó a rendszerben. Továbbá a többi komponens azonnal ide irányítja át a felhasználót, ha bejelentkezés nélkül próbál meg elérni egy útvonalat, ami nem a regisztráció.

Belépni email-cím és jelszó megadásával lehetséges. Amennyiben helyes adatokkal történt a belépés, a szerver válaszként visszaküldi az authentikációs tokent. Ennek a komponensnek a felelőssége, hogy meghívja a szülő komponens beállító függvényét és átadja a tokent paraméterként. Az alkalmazás többi komponense így válik elérhetővé. Sikeres belépés esetén átirányítja a felhasználót a *Dashboard* felületre.

4.3.2.5 Profile / Register



13. ábra

A *Register* komponens felelős azért, hogy továbbítsa a szerver felé azon felhasználók adatait, akik regisztrálni szeretnének a szolgáltatásomra. A szerver válasza után a bejelentkező felületre irányítja a felhasználót.

A *Profile* komponens annyiban tér el ettől, hogy itt az adatmezőket előre feltöltöm a meglévő adatokkal, így azok megváltoztatása lényegesen felhasználóbarátabbá válik. A szerver válasza után a *Dashboard* felületre irányítja a felhasználót.

A két komponens a képernyőképen (13. ábra) látható adatok fogadására képes. A beküldéshez a *Register* komponensnél az összes mező kitöltése szükséges, míg a *Profile* komponensnél a jelszó mező kihagyható, ha azt a felhasználó nem kívánja megváltoztatni.

4.3.2.6 TraderNavbar

14. ábra

Ez a komponens az alkalmazás navigációs fejléce, így az összes hozzá tartozó oldal tetején megjelenik. Ezen keresztül lehet a fentebb bemutatott komponensek között navigálni, valamint itt található az adott helyzetben szükséges authentikációhoz kapcsolódó gomb is.

Amennyiben nincs bejelentkezett felhasználó, akkor a regisztrációhoz vezető gomb jelenik meg a komponens jobb oldalán (14. ábra), ellenkező esetben pedig a kijelentkezéshez szükséges vezérlő (15. ábra).

15. ábra

4.3.3 Útvonalválasztás

A React talán legnagyobb előnye, hogy gyakorlatilag egyetlen oldalunk van összesen, azonban azt, hogy éppen mely komponensek jelennek meg azon az oldalon, mindenképpen szabályozni kell. Erre találták ki a React beépített *Router* elemét, amely a hagyományos többoldalas webalkalmazások mintájára az URL-ben megadott útvonal szerint rendereli ki a megfelelő komponenst.

Ez a Router elem az alkalmazásomban a következőképp néz ki:

```
<Router>
  <div className="container">
    <TraderNavbar token={this.state.token} logout={this.logout}/>
      <Switch>
        <Route exact path="/" render={ (...props) =>
                    <Login {...props} setToken = {this.setToken} /> }
        />
        <Route exact path="/register" render={ (props) =>
                    <Register {...props} token={this.state.token} /> }
        />
        <Route exact path="/dashboard" render={ (props) =>
                    <Dashboard {...props} token={this.state.token} /> }
        />
        <Route exact path="/settings" render={ (props) =>
                    <Settings {...props} token={this.state.token} /> }
        />
        <Route exact path="/backtest" render={ (props) =>
                    <Backtest {...props} token={this.state.token} /> }
        />
        <Route exact path="/profile" render={ (props) =>
                    <Profile {...props} token={this.state.token} /> }
        />
      </Switch>
  </div>
</Router>
```

Az útvonalakat a *Switch* elemen belül lehet megadni. Azon belül minden *Route* elemhez tartozik maga az útvonal, amin elérhető lesz, valamint a *render* paraméterfüggvény. Ebben a függvényben adom meg, hogy mely komponenseket és azokat milyen paraméterekkel használja fel a *Router*.

4.3.4 Authentikáció

Az alkalmazásnak a szerver által kínált token alapú felhasználói authentikáció segítségével kell kezelnie a kliens oldali authentikációt.

Ehhez az alkalmazás fő komponensének állapotában tárolom el a tokent, ha bejelentkezett egy felhasználó. Ez azonban nem elég, ugyanis ez az állapot nem perzisztens, így minden frissítéskor elvesznek az ott tárolt adatok és a felhasználónak újra be kéne jelentkezni.

A token perzisztálásához a JavaScript localStorage szolgáltatását használom, amely a felhasználó által használt böngésző adatmappájába menti el a tokent a kliens gépen.

A dokumentációból nekem ugyan ez nem volt egyértelmű, de a használati tapasztalatom alapján a komponens betöltése másképp történik egy frissítéskor, mint például egy másik komponensből történt átirányítás után. Ennek kiküszöbölésére a konstruktorban és paraméter inicializálása közben is ellenőrzöm, hogy található-e token a localStorage-ban. Ezzel a megoldással, ha már van ilyen, akkor az minden esetben betöltődik a fő komponens állapotába.

Ahhoz, hogy a bejelentkezési oldalról vissza tudjam a fő komponensbe küldeni a szervertől kapott tokent, paraméterként át kell adnom egy függvényt a login komponensnek. A beállításhoz ezt a függvényt kell meghívni és paraméterként átadni a tokent. Ezzel be is állítottuk a fő komponens állapotát, valamint a perzisztálás is megtörtént.

A kijelentkezés is hasonlóan működik, csak az előzővel ellentétben itt mindkét helyről eltávolítom a tokent. Ennek eredményeképp még ha érvényes is maradt a token, újabb azonosítás nélkül nem lehet belépni az adott kliens gépről az alkalmazásba.

Ahhoz, hogy a gyerek komponensek el tudják érni a szervert, mindegyiknek át kell adni paraméterként a tokent.

A fent leírtak megvalósításához az alábbi kód szükséges:

```
constructor() {
  super();
  const token = JSON.parse(localStorage.getItem('token'));
  if (token === null) this.setState({token: ""});
  else if (typeof token.token === "undefined") {
    this.setState({token: ""});
  }
  else {
    this.setState(token);
  }
}

state = JSON.parse(localStorage.getItem('token')) !== null ?
  JSON.parse(localStorage.getItem('token')) :
  {token: ""};

setToken = (token) => {
  localStorage.setItem('token', JSON.stringify({token: token}));
  this.setState({token: token});
};

logout = () => {
  this.setState({token: ""});
  localStorage.clear();
}
```

4.3.5 Dizájn

Az alkalmazás megjelenése a népszerű Bootstrap osztálykönyvtár Reactos manifesztációjára, a React Bootstrapre épül.

Ennek használata nem igényli CSS kód írását, így a fejlesztést lényegesen felgyorsítja. Ezzel együtt egy egységes, letisztult dizájnt kapunk rengeteg beépített elemmel.

Ilyen elemek például a *Form*, a *Button*, a *Card*, amik szerves részét képezik az alkalmazásomnak. Továbbá a Bootstrap grid rendszerét is hasonló elemekkel oldják meg (*Row*, *Col*). Ezen elemek nagyszerűsége abban rejlik, hogy szintaktikailag pontosan úgy használhatók, mint egy közönséges HTML elem, azonban kibővített funkcionalitással rendelkeznek. Ezen funkciók használatához extra paramétereket biztosít, amit szintén a jól megszokott módszerek szerint használhatunk.

4.4 Docker

Az alkalmazásom három különálló eleme (3. ábra) konténerizáció után kiált, így ezt a lépést végre is hajtottam. Az adatbázis, a szerver és a React applikáció külön-külön konténerekben futnak.

A szerverhez és a React alkalmazáshoz is az *alpine-node* image-et választottam, ami a Node.js egy könnyűsúlyú változata. Az adatbázishoz a MongoDB hivatalos image-ét használtam fel. A szerver a 3001-es porton, míg a webapp a 3000-es porton várja a beérkező kéréseket.

Ezek összekötésére *docker-compose*-t használtam, amivel egyetlen paranccsal bármilyen Dockert futtató eszközön telepíthető az egész alkalmazás. A konfigurációs fájl lényegi része alább látható.

```
services:
    react:
        tty: true
        stdin_open: true
        build: react
        restart: always
        ports:
            - "3000:3000"
        volumes:
            - ./react:/react
            - /react/node_modules
        links:
            - server
        networks:
            - webappnetwork
    server:
        build: server
        restart: always
        ports:
            - "3001:3001"
        volumes:
            - ./server:/server
            - /server/node_modules
        depends_on:
            - mongodb
        networks:
            - webappnetwork
    mongodb:
        image: mongo
        restart: always
        container_name: mongodb
        ports:
            - 27017:27017
        networks:
            - webappnetwork
```

5 Tesztelés

5.1 Backend

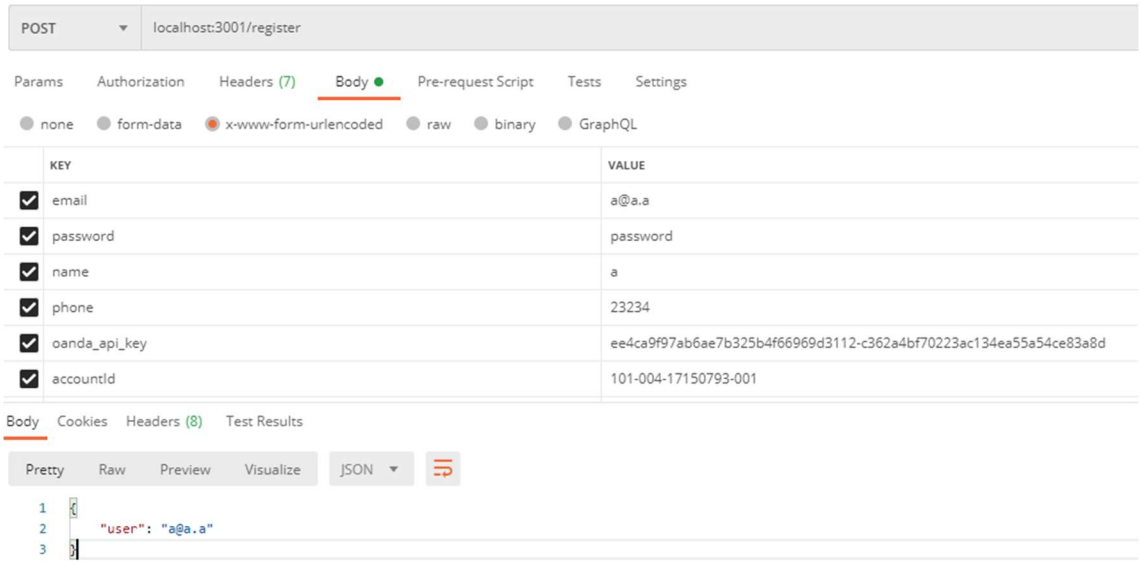
A szerver tesztelését részben manuálisan, részben unit tesztekkel végeztem el. A manuális tesztelésre kétszer is sor került. Amit ebben a részben mutatok be, az a fejlesztés közbeni tesztelés menete, amikor még nem volt működőképes grafikus felületem. A másik a komplett alkalmazás kipróbálásakor történt meg.

5.1.1 Manuális teszt

A manuális teszteléshez a Postman nevű szoftvert használtam, mellyel könnyedén lehet HTTP kéréseket összerakni és elküldeni. A teszt folyamán végig mentem egy átlag felhasználó által legtöbb esetben használt lépéseken.

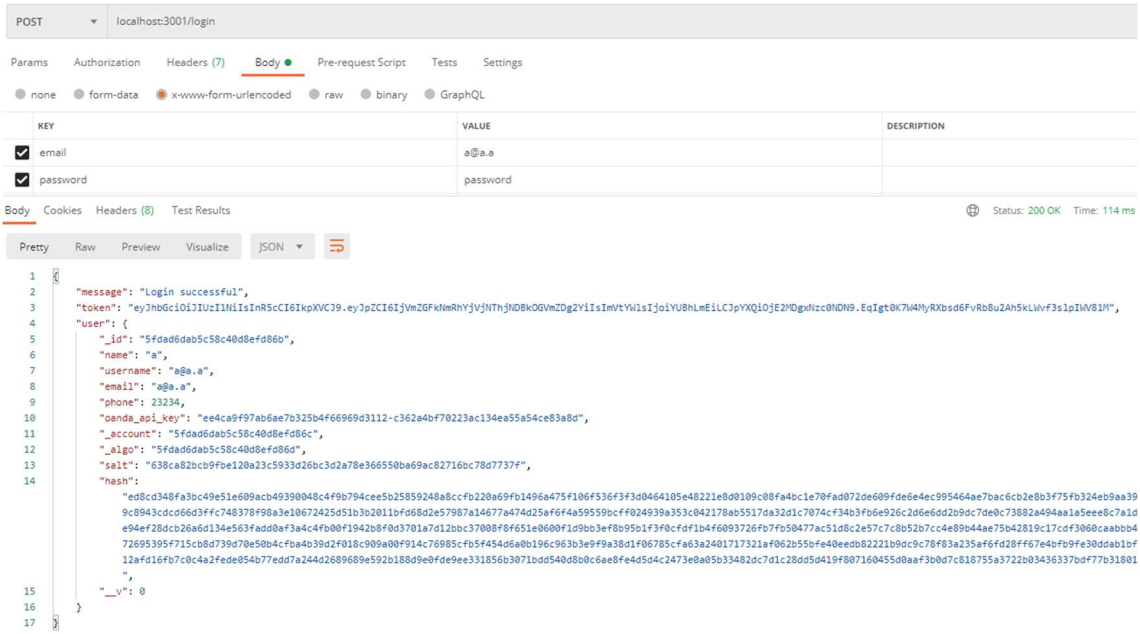
Alább a teljesség igénye nélkül bemutatom néhány fontosabb elemét a tesztnek.

5.1.1.1 Regisztráció



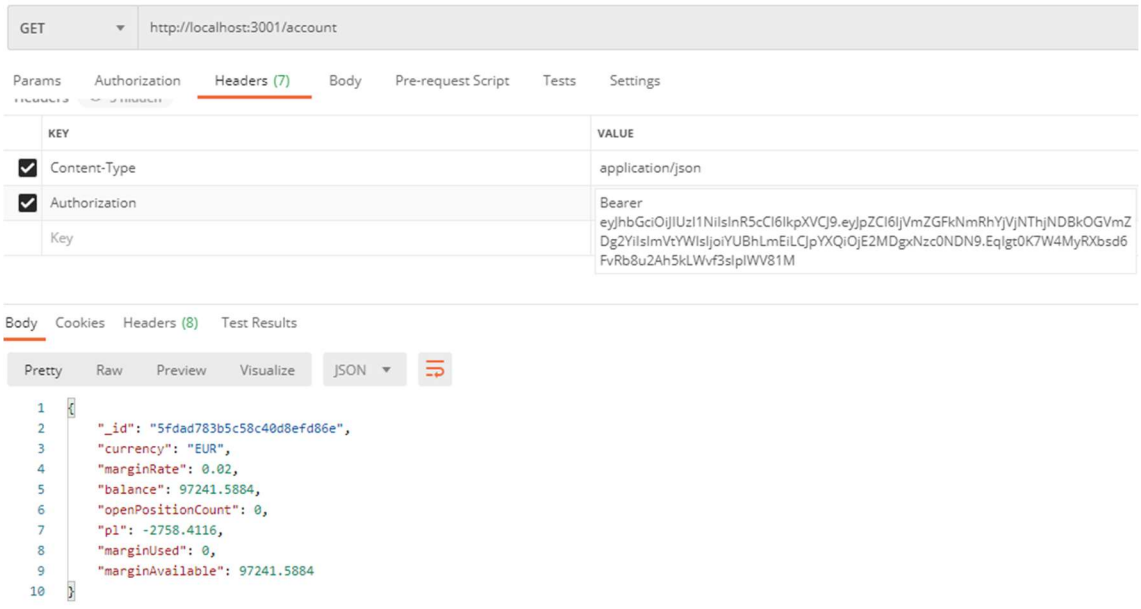
16. ábra

5.1.1.2 Bejelentkezés



17. ábra

5.1.1.3 Számlaadatok lekérése



18. ábra

5.1.2 Unit tesztek

A unit tesztek elkészítéséhez a Mocha, valamint a Chai NPM csomagokat használtam fel. A Mocha egy tesztkörnyezetet biztosít, míg a Chai az eredmények kiértékelését egyszerűsíti le.

A tesztek helyes futásához minden esetben „ki kell mockolni” a működési környezetet, azaz mesterségesen létre kell hoznunk változókat úgy, hogy a tesztelés alatt álló kódrészlet szemszögéből úgy tűnjön, mintha a valódi alkalmazás futása közben hívódott volna meg.

Ahol aszinkron hívásokat is tartalmaz a tesztelt kódrészlet, ott a biztonság kedvéért 10 másodperces időlimitet állítottam be a teszt futásának. Ez alapértelmezés szerint mindössze 2 másodperc lenne.

5.1.2.1 HTTP válasz fejlécének beállítása

Ezt a funkciót egy külön middleware látja el, amelyet minden kérés elejére bekötöttem, így fontos, hogy megfelelően működjön. A fejlécben a tartalom típusát állítja be, hogy a kérést küldő kliens tudja, hogy JSON formátumban fogja megkapni a választ.

```
describe("Content-type", function () {
  it("should be application/json", function (done) {
    const req = {};

    let res = {
      header: null,
      setHeader: (type, value) => {
        res.header = {
          [type]: value,
        };
      },
    };

    setHeaderMW({})(req, res, (err) => {
      expect(err).to.eql(undefined);
    });
    expect(res.header["Content-Type"]).to.eql("application/json");
    done();
  });
});
```

5.1.2.2 Lekérdezések

A lekérdező middleware-ek tesztjeit egy fájlba foglaltam, hiszen nagyon hasonlítanak egymásra. Itt tesztelem a tranzakciók, a számlainformáció és az algoritmus paramétereinek lekérdezését.

Az alábbiakban csak a tranzakciók lekérdezését mutatom be, hiszen a másik kettőben ehhez képest csak apró eltérések találhatók.

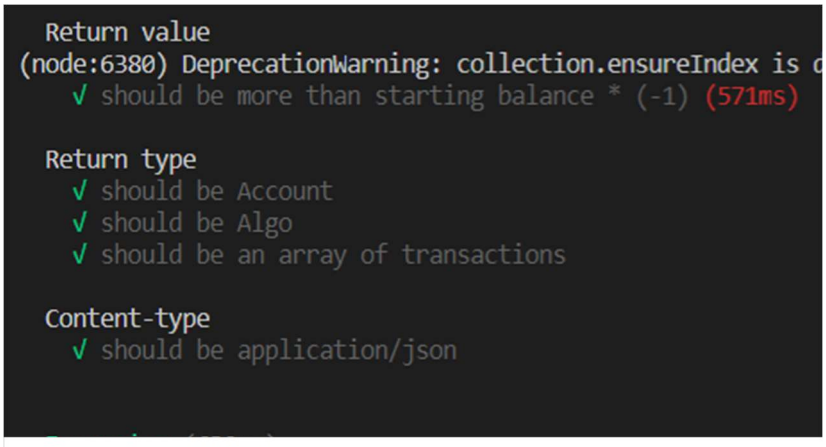
```
describe("Return type", function () {
  this.timeout(10000);
  it("should be an array of transactions", async function () {
    let email = null;
    await User.findOne((err, user) => {
      email = user.email;
    });

    const req = {
      user: {
        username: email,
      },
      params: {
        count: 15,
      },
    };

    let res = {
      json: (a) => {
        res.transactions = a;
      },
    };

    oanda.run();

    setTimeout(() => {
      getTransactionsMW()(req, res, (err) => {
        expect(err).to.eql(undefined);
      });
    }, 1000);

    setTimeout(() => {
      expect(res.transactions).to.have.lengthOf(15);
      expect(res.transactions[0]).to.have.all.keys([
        "id",
        "instrument",
        "time",
        "units",
        "pl",
      ]);
    }, 2000);
  });
});
```

5.1.2.3 Historikus teszter

Ennek a funkciónak a részletes tesztelése a szakdolgozat terjedelmén kívül esne, de egy alapvető ellenőrző tesztet készítettem, ami annyit néz, hogy az eredményként visszakapott profit értéke ne legyen irreális érték, azaz veszteség esetén ne lehessen nagyobb a veszteség, mint a kezdőösszeg. Ezzel továbbá a visszatérési érték típushelyességét is ellenőrzőm.

```
describe("Return value", function () {
  this.timeout(10000);
  it("should be more than starting balance * (-1)", function (done) {
    const algo = {
      instrument: "EUR_USD",
      granularity: "H1",
      units: 1000000,
      MAperiod1: 15,
      MAperiod2: 30,
      marginRatio: 30,
      trailValue: 0.0005,
      trailWait: 0,
      makeOrderWaitLimit: 0.0007,
    };

    const timeframe = {
      from: new Date("2018-07-01"),
      to: new Date("2019-08-01"),
    };

    const balance = 100000;

    backtest.run(balance, algo, timeframe).then((ret) => {
      expect(ret.profit).to.be.above(balance * -1);
      done();
    });
  });
});
```

5.1.2.4 Eredmény



19. ábra

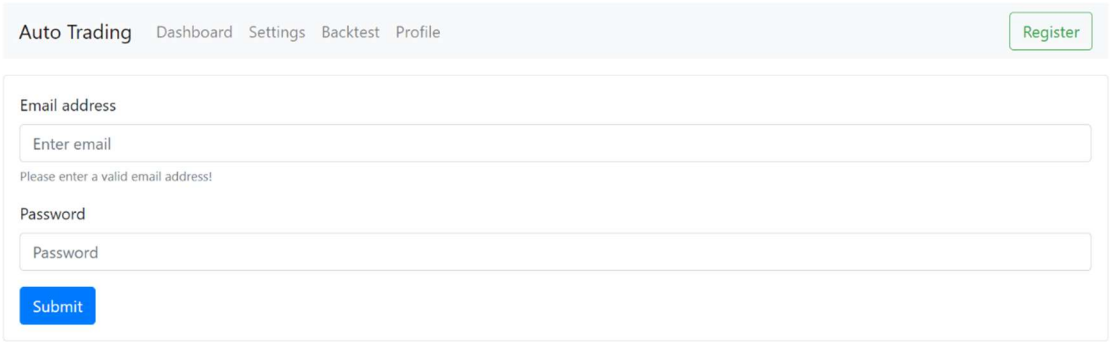
5.2 Frontend

A grafikus felület teszteléseként végig jártam az alkalmazást és annak minden funkcióját kipróbáltam, hogy az az elvárások szerint működik-e. A következőkben ennek a folyamatnak mutatom be néhány sarkalatos pontját.

5.2.1 Regisztráció és bejelentkezés

Az alkalmazást megnyitva a bejelentkező képernyő fogad. A navigációs sávon bárhova kattintok a regisztrációs gombon, nem történik semmi, ami a helyes működés, ugyanis bejelentkezés nélkül a bejelentkező oldalra kell irányítani a felhasználót.

A regisztráció gombra kattintás után kitöltöm a megjelenő formot, majd elküldöm. Ezek után újra a bejelentkező képernyőn találom magam. Itt az előbb regisztrált adatokkal megpróbálok belépni.



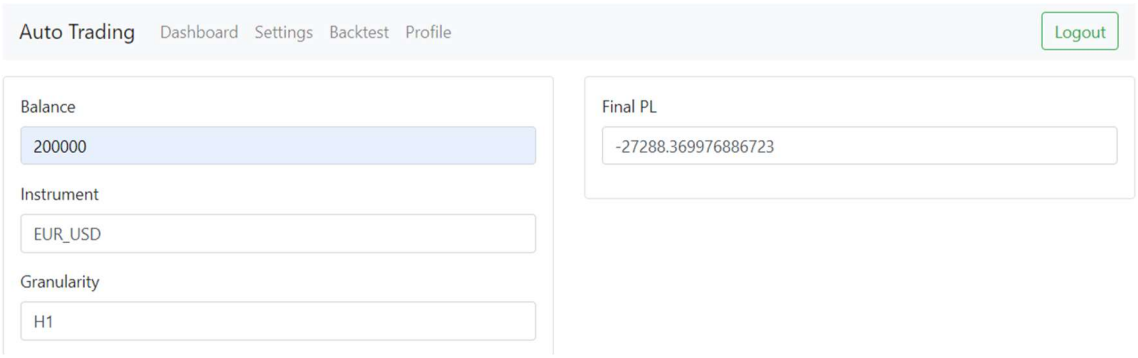
20. ábra

5.2.2 Az alkalmazás használata

A sikeres belépés után a *dashboard* képernyőn találom magam, ahol látom minden adat rendezetten megjelenik, ahogy azt a 11. ábra is mutatja. Innen már lehetőségem van továbblépni a navigációs sáv valamelyik elemére kattintva, ugyanis bejelentkezett állapotban vagyok.

A *settings* felületen megjelenik az algoritmusomhoz tartozó összes adat mezője, amiket előre ki is töltött az alkalmazás. A számlaegyenlegen kívül mindent meg is tudok változtatni, ami megfelel a specifikációnak. Egy adat megváltoztatása után rákattintok a mentésre. Ezzel visszakerülök a *dashboard*ra, ahonnan visszanavigálva a beállításokhoz már az általam megváltoztatott értéket látom.

A backtest oldalon is helyesen jelennek meg a mezők a beállításokhoz hasonlóan. A szükséges extra adatok megadása és a *Run test* gomb megnyomása után az eredmény mezőben megjelenik a profit mértéke. (21. ábra)



21. ábra

A profil oldalra navigálva a 13. ábra szerinti megjelenés fogad. Az email értéken kívül minden módosítható. Itt is, ha változtatok egy értéken, majd később visszalépek erre az oldalra, akkor már az új érték fogad. A jelszó megváltoztatása is tökéletesen működik, hiszen új jelszó beírása, majd kijelentkezés után már az új jelszóval tudtam csak újra bejelentkezni.

6 Összegzés

6.1 Saját munka értékelése

A szakdolgozatom eddigi életem messze legnagyobb projektje volt, így már önmagában arra is büszke vagyok, hogy egy kész munkát sikerült kiadnom a kezeim közül.

A bevezetőben leírt gondolatomhoz a dolgozat elkészülte után is tartom magamat, miszerint a több munka és nagyobb kihívások ellenére is megérte egy érdekes témát választani. A jövőben se döntenék másképp. Az informatikán részben kívül eső témaválasztással olyan irányba tudtam bővíteni az ismereteimet, ami érdekel, valamint szerencsés esetben még hasznos is lehet.

Ezen túl szakmai szemmel nézve is nagyon sokat fejlődtem és tanultam. Megismerkedtem rengeteg technológiával, amelyek egy része kifejezetten hasznosnak bizonyult, egy másik részéről pedig megtanultam, hogy legközelebb érdemes más megoldást keresni.

6.2 Továbbfejlesztési lehetőségek

Annak érdekében, hogy típusosságot, és ezzel még nagyobb kiszámíthatóságot vigyek a projektbe, érdemes lenne TypeScriptet használni. Ezzel a nyelvvel sajnos később ismerkedtem meg, mint ahogy ebbe a projektbe belefogtam, így az energiámat inkább magára az alkalmazásra fordítottam, mint hogy szintaktikailag teljesen újraírjam a meglévő kódokat.

A jelenlegi React alkalmazás ugyan ellát minden feladatot, amire a felhasználónak szüksége lehet, azonban a dizájn nem különösebben egyedi, vagy látványos. Ezzel az oldalával a projektnek még mindenképpen érdemes lehet foglalkozni a későbbiekben.

7 Irodalomjegyzék

[1] „econom.hu,” 18. január 2010.. [Online]. Available: http://www.econom.hu/a-tozsde-tortenete/.

[2] M. Sághi, „A devizapiac,” in *Nemzetközi gazdaságtan - Elmélet és gazdaságpolitika*, Panem Könyvkiadó.

[3] „livestudio.eu/,” [Online]. Available: https://livestudio.eu/hu/blog/55-a-html-tortenete-vagyis-a-weboldal-keszites-hoskora.

[4] P. Zsombor, „Szerver oldali JavaScript,” [Online]. Available: http://malna.tmit.bme.hu/vitmav42.

[5] „weblabor.hu,” [Online]. Available: http://weblabor.hu/cikkek/nodejs-alapok.

[6] „hu.reactjs.org,” [Online]. Available: https://hu.reactjs.org/tutorial/tutorial.html.

[7] „ithub.hu,” [Online]. Available: https://ithub.hu/blog/post/Docker_bevezetes_a_containerek_vilagaba.

[8] „en.wikipedia.org,” [Online]. Available: https://en.wikipedia.org/wiki/Web_development.

[9] „mfrachet.github.io,” [Online]. Available: https://mfrachet.github.io/create-frontend-framework/vdom/intro.html.