XXII. reál- és humántudományi Erdélyi Tudományos Diákköri Konferencia (ETDK)
Kolozsvár, 2019. május 23–26.

Woody

Nyílászárók tervezésére és gyártására alkalmas szoftverrendszer

Szerzők:

Máthé Orsolya
Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar, Informatika szak, III. év
Miklós Botond
Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar, Informatika szak, III. év

Témavezetők:

Bege István, szoftverfejlesztő,
Codespring
Farkas Attila, szoftverfejlesztő,
Codespring
Sulyok Csaba, doktorandusz
Babeş–Bolyai Tudományegyetem, Matematika és Informatika Kar,
Codespring

CODESPRING
OUR CODE YOUR SUCCESS

Kivonat

A nyílászárók (ablakok és ajtók) gyártása során olyan problémákat figyelhetünk meg, mint a papíron elvégzett bonyolult számítások időigényessége, néha pontatlansága, ez pedig váratlan kiadást, időt és munkát jelenthet a cég számára.

A Woody projekt célja megkönnyíteni és hatékonyabbá tenni a fafeldolgozó műhelyben a mindennapi munkát. A megrendelések nyilvántartása mellett lehetőséget nyújt egy szabásjegyzék generálására is. Ez magába foglalja a műhelyrajzot méretarányosan kirajzolt ábrákkal és a termék elkészítéséhez szükséges faelemek listáját. A szoftver egy optimális vágási stratégiát is javasol ezen faelemek kivágására, ezzel csökkentve az anyagveszteséget. A termékek leírása egy erre a célra kifejlesztett szakterület-specifikus nyelvvel (DSL) történik, így az alkalmazás könnyen bővíthető, skálázható a felhasználó igényei szerint.

A reszponzív webes felület lehetőséget ad műhelyen kívüli használatra, így akár helyszíni képeket is fel lehet tölteni mobil eszközről, amelyek elősegítik a gyártási és behelyezési folyamatot.

A dolgozat részletes bemutatást nyújt az alkalmazás architektúrájáról, a felhasznált technológiákról és a fent említett funkcionalitásokról.

Tartalomjegyzék

Bevezető **1**

1. Funkcionalitások **3**

2. A Woody alkalmazás felépítése **5**

2.1. Az alkalmazásszerver 5

2.1.1. Architektúra 5

2.1.2. Formula Parser 7

2.1.3. Domain adatmodell 7

2.1.4. Perzisztens adatmodell 9

2.1.5. Optimális vágás 10

2.2. A webkliens 10

2.2.1. Architektúra 10

2.2.2. Egyoldalas webalkalmazás és útvonalválasztás 11

2.2.3. Komponensek 11

3. Felhasznált technológiák és eszközök **13**

3.1. Szerveroldali technológiák 13

3.2. Web technológiák 14

3.3. Eszközök 16

4. Munkamódszerek **19**

4.1. Scrum és Kanban 19

4.2. GitFlow 19

4.3. Kódellenőrzés 20

4.4. Folyamatos integráció és kitelepítés 20

5. Az alkalmazás működése **22**

6. Következtetések és továbbfejlesztési lehetőségek **26**

Bevezető

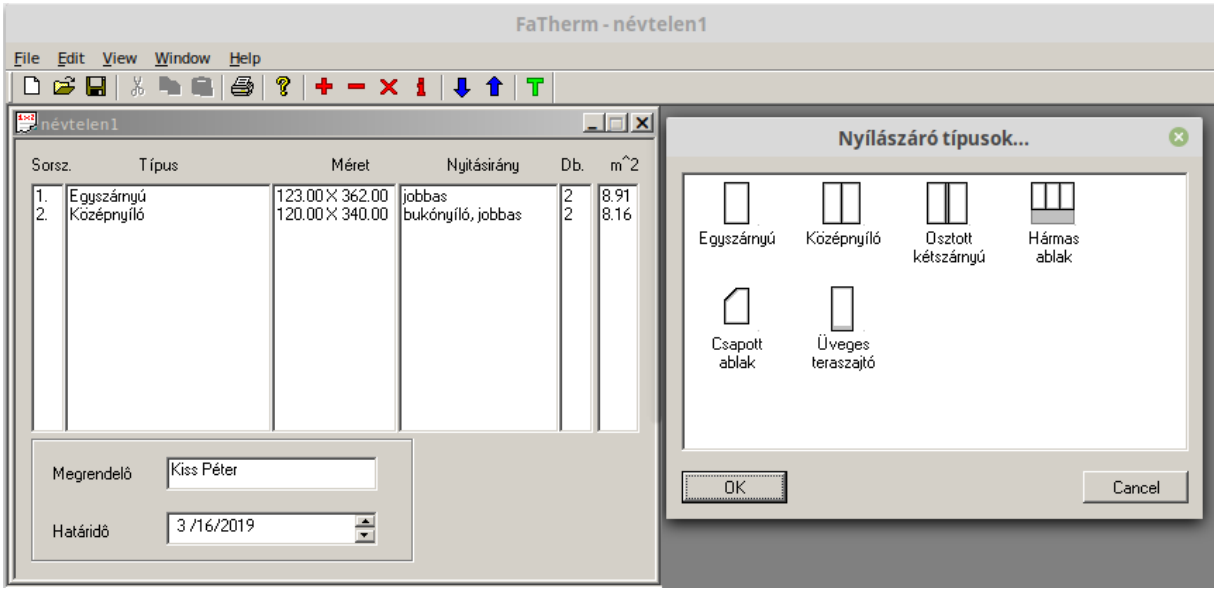
A nyílászárók (ablakok és ajtók) gyártása során a tervezés ma is részben papíron történik. Ennek az oka a sokféle ajtó- és ablaktípus, amit a kliens igényelhet. A sokféle követelmény miatt nehéz olyan szabályrendszert megalkotni, ami minden helyzetre alkalmazható. A papíron végzett számítások hátránya, hogy időigényesek, néha hibásak, ez pedig váratlan kiadást, időt és munkát jelenthet a cég számára. Ugyanakkor a használt képletek változhatnak, ha a kliens egy új, még nem létező ajtó- vagy ablaktípust kér, új gépek kerülnek a gyártási folyamatba vagy ha más új tényezők jönnek be. Az asztalos ugyanakkor egy összetett feladattal találja szembe magát akkor, amikor a megvásárolt nyersanyagot optimálisan igyekszik felhasználni, a lehető legkevesebb veszteséggel.

A Woody alkalmazás célja megkönnyíteni és hatékonyabbá tenni a fent említett folyamatot a gyártó cég számára. Egy webes felületen keresztül lehetőséget nyújt a megrendelések számontartására, valamint az ezekhez tartozó termékek, adatok kezelésére. Rendelésfelvételkor mobil eszközről elérve a weboldalt helyszíni fényképeket lehet feltölteni, melyek a beszerelés során felbukkanó, váratlan problémákat segítenek elkerülni. A rendeléshez tartozó ablakok vagy ajtók megfelelő paraméterezése után kigenerálható a műhelyrajz, mely a paraméterek szerint méretarányosan kirajzolt ábrákat foglalja magába, és a szabásjegyzék, amely tartalmazza a termék elkészítéséhez szükséges faelemek listáját. Ezen faelemek hossz szerinti darabolására egy optimális vágási stratégiát is meghatároz a szoftver, figyelembe véve a raktáron levő nyersanyagok méreteit, ezzel csökkentve az anyagveszteséget. A műhelyrajz, a szabásjegyzék és a vágási stratégia nemcsak a felületen tekinthető meg, hanem ezeket nyomtatni is lehet, hogy a műhelyben könnyen elérhetőek, hordozhatóak legyenek.

Ez a szoftverrendszer egy újragondolása és egyben továbbfejlesztése az 1. ábrán látható FaTherm nevű alkalmazásnak, melyet a dolgozat egyik mentora, Farkas Attila, írt 2004-ben. A FaThermet azóta is aktívan használják egy csomafalvi műhelyben[1]. Mivel ez mára már elavult, nem támogatott technológiákra épült, nagyon nehézkessé vált a karbantartása és a továbbfejlesztése. Ezt hivatott lecserélni és kibővíteni a Woody, olyan módon, hogy akár más műhelyek is felhasználhassák. A desktopról webre térés lehetővé teszi az alkalmazás egyidejű használatát több eszközről. Az új adattárolási szerkezet módot ad újabb terméktípusok hozzáadására, programozói beavatkozás nélkül. Az optimális vágási stratégia megadása pedig a cég költségeit csökkenti.

A dolgozat hátralevő része a következőképpen van struktúrálva: az 1. fejezetében a szoftverben megvalósított funkcionalitások vannak bemutatva. A 2. fejezetben szó esik a rendszer általános architektúrájáról, valamint részletes betekintést nyerhetünk a szerver és kliens oldal

[1] A műhely weboldala: https://www.farkas.ro/



1. ábra. A FaTherm alkalmazás, melyet továbbfejleszt a Woody projekt.

felépítésébe, megvalósításaiknak fontosabb részleteibe. A 3. fejezetben a felhasznált technológiákról és eszközökről található egy-egy rövid tájékoztató. A 4. fejezetben a projekt megvalósítása során alkalmazott munkamódszerek és folyamatok kerülnek bemutatásra. Az 5. és 6. fejezetben az alkalmazás működése, illetve a továbbfejlesztési lehetőségei vannak részletezve.

A projekt fejlesztése 2018 októberében kezdődött a *Csoportos Projekt* tantárgy keretén belül, a Codespring Mentorprogram részeként. Szeretnénk megköszönni Dr. Simon Károlynak és Sulyok Csabának a koordinálást, Bege Istvánnak és Farkas Attilának a mentorálást és szakmai segítséget, valamint a tantárgy kereteiben csatlakozott diáktársainknak, Heim Lászlónak, Kádár Norbertnek, Kovács Patriknak és Sütő Ágostonnak a hozzájárulást. Külön köszönet Sütő Ágostonnak a *Formula Parser* (2.1.2. fejezet) megírásáért [2].

[2]Nyilvános repository: `https://gitlab.com/suto.agoston/nyzlny`, utolsó megtekintés dátuma: 2018-04-05

1. Funkcionalitások

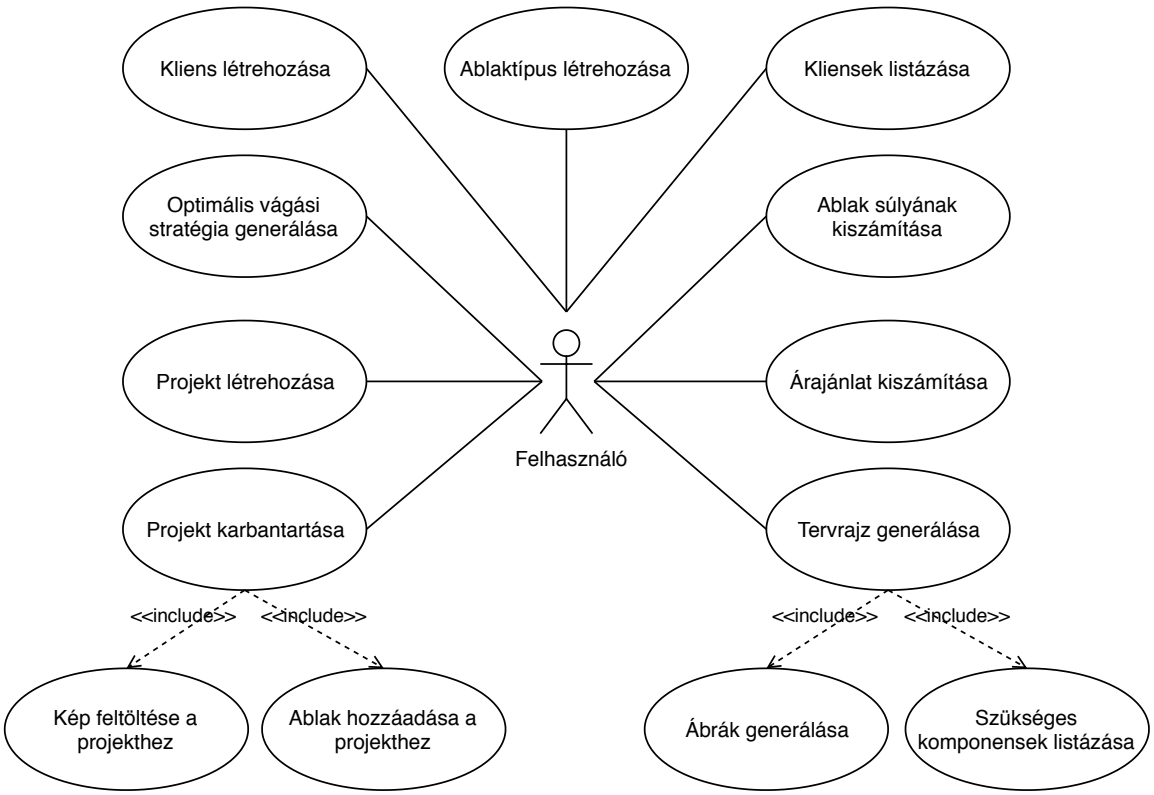
Mivel a weboldalt a műhelyben mindenki egyenjogúan használhatja, egyetlen típusú felhasználóra van szükség, így minden funkcionalitás elérhető bárki számára (ld. 2. ábra).

Minden felhasználónak lehetősége van új klienst és projektet bevinni a rendszerbe. A kliens létrehozásához csak néhány személyes adat szükséges. A projekt létrehozásához kell egy kliens és egy határidő, de opcionálisan lehetőség van ablakok és képek csatolására is (5. fejezet). Az adatbázisban levő projekteket és klienseket ki is lehet listázni.

A Woody legfontosabb funkcionalitásai a projektekhez tartozó adatok értelmezésében rejlenek. Minden projekt esetén le lehet kérni a műhelyrajzot és a szabásjegyzéket, egy optimális vágási stratégiát, illetve a csatolt képeket. A műhelyrajz a projekthez tartozó ablakok dinamikusan kigenerált, méretarányos ábráit tartalmazza, míg a szabásjegyzék az ablakokhoz tartozó komponensek listáját. Ezeket ki is lehet nyomtatni. A nyomtatott változatban a műhelyrajz és a szabásjegyzék alatt egy összefoglaló táblázat is megjelenik a gyártási folyamat megkönnyítése érdekében.

Az optimális vágásnál a felhasználó megtekintheti, hogy a készleten levő deszkatípusokból az adott rendeléshez tartozó faelemeket hogyan lehet kivágni a legkisebb veszteséggel. A vágási stratégiát dinamikusan generált, vektorgrafika-alapú képek szemléltetik.

A projekthez tartozó helyszíni képek listája az ablakok beszerelési folyamatát segítenek



2. ábra. Funkcionalitások

előkészíteni.

A felhasználó módosíthatja is a rendszerben szereplő kliensek adatait, megrendeléseket, hogy azok mindig a valós helyzetet tükrözzék. Egy projekt karbantartása alatt lehetőség van képek törlésére, bevitelére, valamint az ablakok szerkesztésére, törlésére vagy új példány létrehozására. Minden ablak esetében a típusából származó előre definiált változókat lehet csak megadni/átírni, az ablakot meghatározó képleteket csak az ablaktípuson keresztül lehet elérni.

Továbbá, minden projektről lehet kérni árajánlatot/súlybecslést, ami magában foglalja a projekthez tartozó ablakok darabonkénti árát/súlyát, s ezeknek összesítését.

2. A Woody alkalmazás felépítése

A Woody alkalmazásnak két fő alkotóeleme van: a webes felület és az alkalmazásszerver (ld. 3. ábra). Az alkalmazásszerver egy *Application Programming Interface*-et (API) biztosít az adatok elérésére *RESTful* módon. A REST (*REpresentational State Transfer*) egy architekturális minta webes kommunikáció tervezésére, melyet Roy Fielding 2000-ben ismertetett doktori disszertációjában [19]. A fontosabb jellemzői egy *RESTful* szolgáltatásnak az egységes felület, melyen keresztül egyértelműen elérhetőek az adatok, az állapotmentesség, a kliens–szerver alapú kommunikáció, a rétegelés lehetősége a háttérben, anélkül, hogy a kliens ezt észlelné, valamint a *cache*-elés lehetősége.

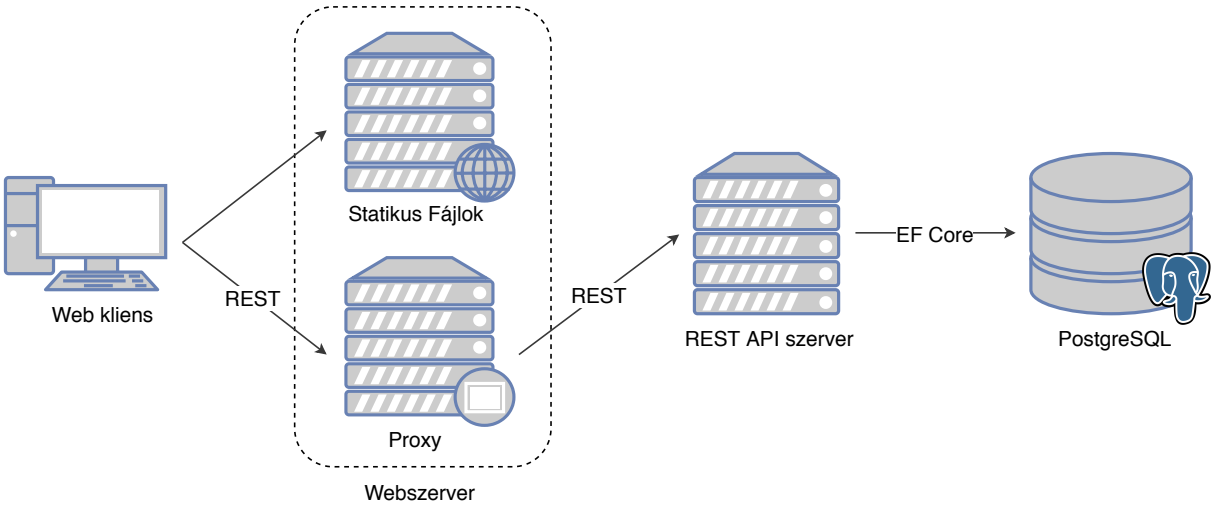
Az alkalmazás felépítéséből adódóan a külvilág csak a webszervert érheti el, ez szolgálja ki a felülethez szükséges statikus állományokat, valamint továbbítja a `/api/`-val kezdődő útvonalakra érkező REST kéréseket az alkalmazásszerver felé egy belső hálózaton. Az alkalmazásszerver feldolgozza a kérést és menti vagy lekéri az adatokat a PostgreSQL relációs adatbázisból, majd visszaküldi az eredményt vagy az esetleges hibát JSON (*JavaScript Object Notation*) formátumban. A megvalósítás további részletei a 4.4. fejezetben találhatóak.

2.1. Az alkalmazásszerver

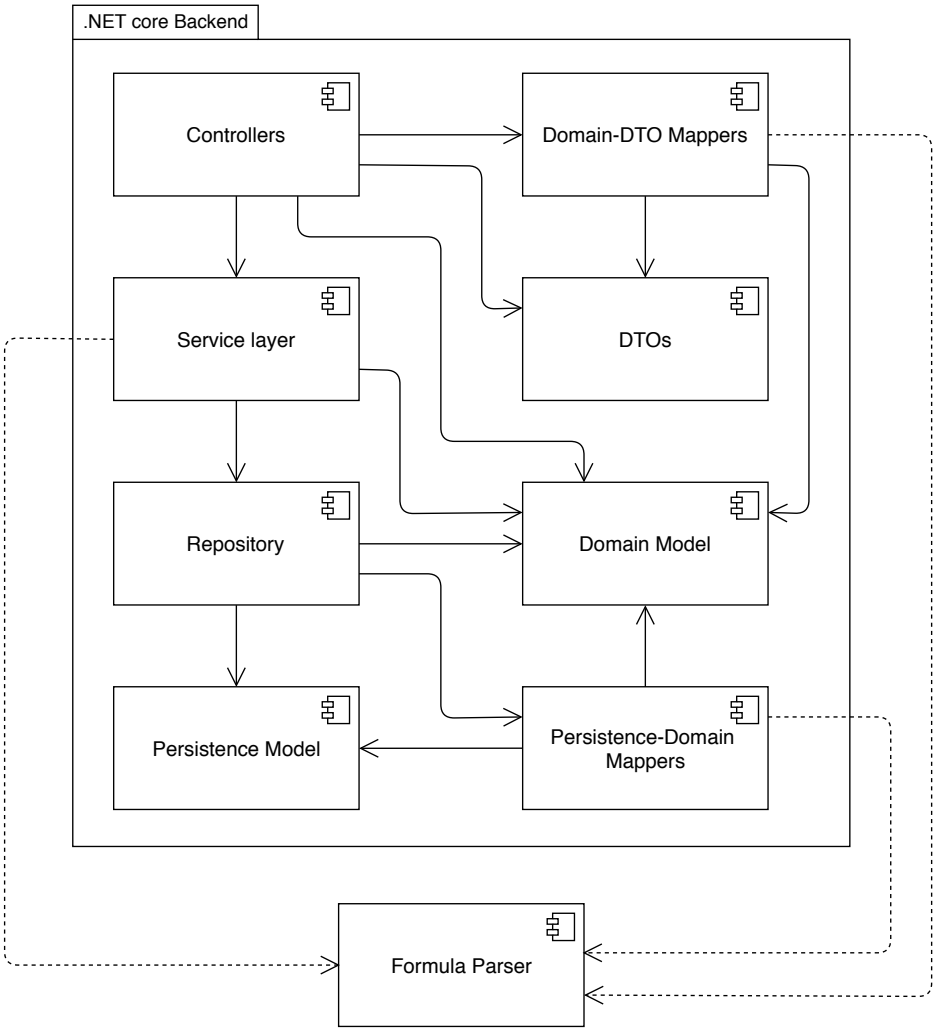
2.1.1. Architektúra

A szerver jól elkülöníthető komponensekből épül fel (ld. a 4. ábra).

A *Persistence Model* tartalmazza azokat az adatstruktúrákat, amelyek meghatározzák az adatbázis szerkezetét. A *Domain Model* és *DTOs (Data Transfer Objects)* ezeknek a struktúráknak egy-egy sajátos formáját képviselik: míg az előbbi az üzleti logika megvalósítását hivatott



3. ábra. A kommunikáció szemléltetése a kliens, a webszerver, az alkalmazásszerver és az adatbázis között.



4. ábra. A szerver komponensei.

megkönnyíteni, az utóbbi a kliens és a szerver közötti adatcserét formalizálja.

A *Controllers* komponens feladata a kliens kéréseinek fogadása és megválaszolása. A beérkező DTO-kat domain modellé, a kimenő adatokat DTO-kká alakítja a *Domain–DTO Mappers* segítségével.

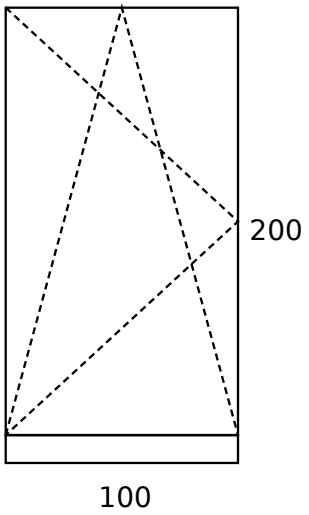
A *Service layer* tartalmazza az üzleti logikát. Ide érkeznek a kérések a *Controllers*-től, itt találhatóak azok a műveletsorok amelyek a megfelelő válaszok előállításáért felelnek.

A *Repository*, az adathozzáférési réteg, az adatbázissal történő kommunikációt valósítja meg. A *Service layer*től kapott domain modellt a *Persistence–Domain Mapper* segítségével persistence modellé alakítja és elvégzi a megfelelő CRUD műveletet (létrehozás, olvasás, módosítás, törlés).

A *Formula Parser* egy olyan modul, amely lehetővé teszi a képletek adatstruktúrákban való tárolását és ezeknek a későbbi kiértékelését.

```
SVG[
    width + 30, height + 20,
    WindowPane[opening, 0, 0,
        width, height * 0.92],
    Fill["none",
        Rect[0, height * 0.92,
            width, height * 0.06]
    ],
    Text[width + 5, height / 2, height],
    Text[width * 0.4, height + 15, width]
]
```

(a) Kódrészlet a *Nyílászáró leíró nyelvben*



(b) A generált műhelyrajz

5. ábra. Példa a *Formula Parser* által definiált DSL működésére.

2.1.2. Formula Parser

A *Formula Parser* két fő feladata a szabályok értelmezése, kiértékelése és a dinamikus rajzok generálása.

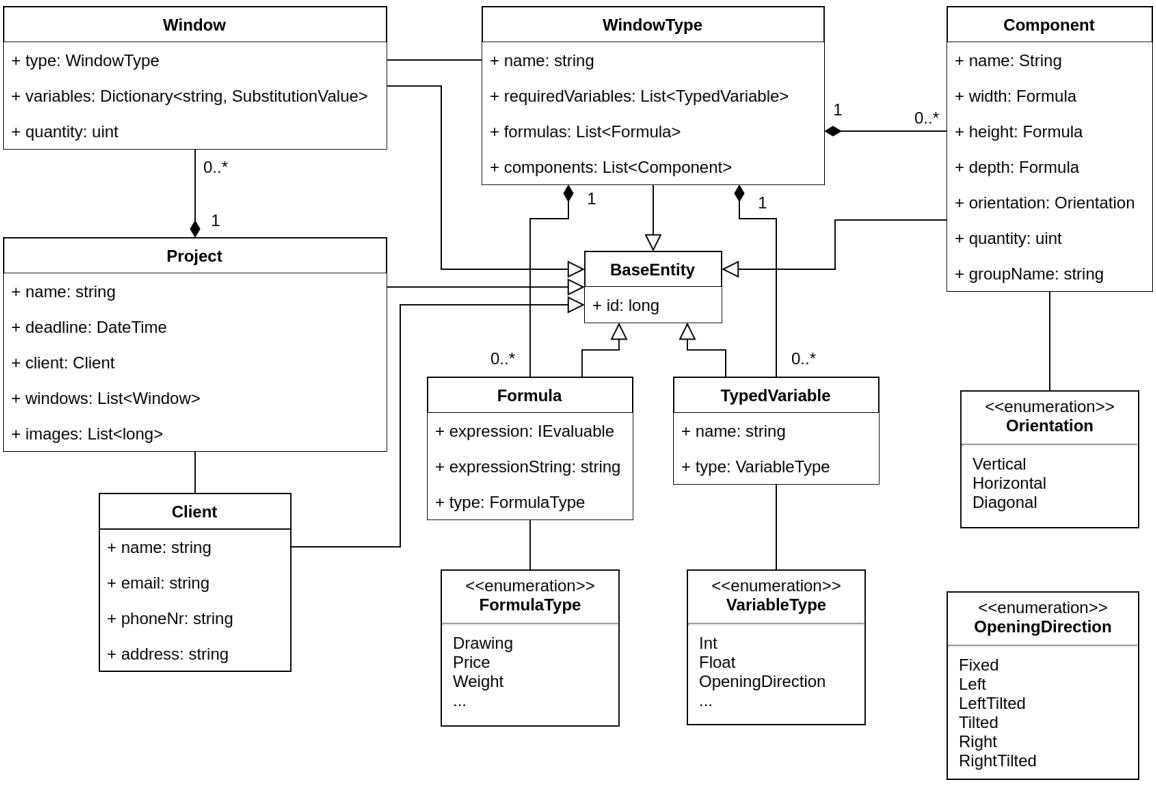
A szabályok lehetnek egyszerű képletek, amelyek a komponensek valamilyen tulajdonságát írják le. Például a `(width + height) / 2` formula egy méretet határoz meg. Ezeket a képleteket az adatbázisban való tárolás mellett később ki is lehet értékelni, ha a megfelelő változóknak van értéke. A példát folytatva, ha `width` értéke `3` és `height` értéke `5`, akkor a *Parser* meg tudja határozni a végeredményt, ami ebben az esetben `4`.

Ugyanakkor ábrákat leíró parancsokat is lehet használni. A `Line[0, 0, 100, 100]` például egy vonalat rajzol ki a `(0, 0)` koordinátából indulva a `(100, 100)` koordinátáig.

Mivel a projektben dinamikusan generált ábrákra van szükség, az előző két funkciót kombinálva létrejött egy szakterület-specifikus nyelv (DSL, Domain-Specific Language), ez a *Nyílászáró leíró nyelv (NyZLNy)*. Ebben olyan parancsok vannak definiálva, amelyek kimondottan az ablakokat ábrázoló ábrák leírását, generálását könnyítik meg. Ilyen az 5a. ábrán látható kódrészletben használt `WindowPane` parancs is. Ennek öt paramétere van: az első megadja, hogy az ablak milyen irányban nyílik, a következő kettő az ablakot ábrázoló téglalap kezdőpontjának a koordinátái, az utolsó kettő pedig ennek a téglalapnak a szélessége és hosszúsága. Ez a parancs az 5b. ábrán megjelenő nagy téglalapot generálja ki, a benne levő szaggatott vonalakkal együtt (ezek jelzik a nyílásirányt).

2.1.3. Domain adatmodell

A Woody projekt esetében három különböző adatmodell reprezentáció is megjelenik a szerverben, ahogy az a 4. ábrán is látható. A *Persistence Model* tükrözi a adatbázis tábláit, erről rész-



6. ábra. Domain Model: Az üzleti logikában használt adatstruktúrák

letesebben a 2.1.4. fejezetben van szó. A *Domain Model* ezeket az adatmodelleket tartalmazza egy olyan formátumban, amivel az üzleti logika a legjobban tud dolgozni, ez kerül nagyító alá ebben a fejezetben. A *DTOs* pedig azokat az adatmodelleket tartalmazza, amelyeket a klienssel való kommunikáció során használ fel a *Controllers* modul a szerverből.

A *Domain Model* pontos szerkezetét a 6. ábra tükrözi. Minden entitást leíró osztály ősosztálya a `BaseEntity`, amely egyetlen `long` típusú `id` adattagot tartalmaz, mely az objektumok beazonosításához szükséges.

A `Client` osztály a kliensek adatait (név, e-mail cím, telefonszám, lakcím) képviseli. A `Project` a megrendelésekhez tartozó információkat foglalja magába, például egy `Client` típusú klienst. Megfigyelhető, hogy míg az ablakokat tartalmazó lista `Window` típusú objektumokat tartalmaz, addig a képeknek csupán az egyedi azonosítójukat tárolja el a `Project`. Mivel a képekkel semmilyen műveletet nem végez az üzleti logika, erőforrás pazarlás lenne a róluk szóló információt belezsúfolni a projekt entitásba. Ezzel ellentétben az ablakokon különböző műveletek hajtódhatnak végre az üzleti rétegben, ezért ott már indokolt a teljes objektumlista.

Az alkalmazás könnyed bővíthetőségének érdekében semmilyen ábrázolással illetve számítással kapcsolatos képlet sincs *hardcode*-olva. Azok az ablaktípusok, amelyeket a kliens lát a webes felületen, a szerveroldalon a `WindowType` entitásban jelennek meg, ahol egy névre és három fontos listára bomlanak le. A `formulas` formulákat, a `requiredVariables` a képletek kiértékeléséhez szükséges típusos változókat, a `components` komponenseket tartalmazó lista.

A `Formula` osztály lehetővé teszi, hogy bármilyen ablaktípust leírjon a `WindowType`. Ennek három adattagja van. Az első, a `type`, meghatározza, hogy mit ír le az éppen szóban forgó képlet egy `FormulaType` enumeráció segítségével, ami lehet akár rajz, ár, súly vagy méret. A második, az `expressionString`, szövegként tárolja az adott képletet. A harmadik, az `expression`, a formula kiértékelhető alakját tartalmazza, ezért ennek a típusa `IEvaluable`. A hagyományos szöveg átalakítását erre a formátumra a *Formula Parser* biztosítja. A `requiredVariables` tárolja a képletekhez szükséges változókat, amelyeket a típusuk (a `VariableType` enumeráció a lehetséges típusokat adja meg) és a nevük határoz meg.

A `Component` entitás az egy-egy ablaktípust felépítő komponenseket, például az ablakkeretet alkotó faelemeket definiálja. A `width, height, depth` (szélesség, magasság, mélység) adattagok képletek, ezzel biztosítva, hogy az ablakok alkotóelemei is kellőképpen kötetlenek legyenek. Emellett minden komponensnek van neve, fekvése, mennyisége és csoportneve (a faelemek csoportba rendezése a gyártás folyamatát és nyomonkövetését segíti).

A `Window` modell képviseli az ablakok példányait. Egy példánynak van típusa (`WindowType`), darabszáma és változó-értékei (az adott ablakpéldányhoz tartozó változók behelyettesítési értékei). A `variables` a változókat `(string, ISubstitutionValue)` párosokként tárolja egy `Dictionary`-ben, amelyek a megnevezést és az értéket képviselik (pl.: `(width, 3)`). Ezek a változónevek meg kell feleljenek az ablaktípus `requiredVariables` mezőjében tárolt változók neveinek. Ezek a konkrét értékek lesznek behelyettesítve ugyancsak az ablaktípusban megadott képletekbe, a *Formula Parser (2.1.2)* segítségével. Így jönnek létre minden ablak instanciához a megfelelő méretű komponensek, a méretarányos ábrák és a megfelelő árbecslések.

2.1.4. Perzisztens adatmodell

A szerver az adatbázissal egy ORM keretrendszer (Object Relational Mapper) segítségével kommunikál, amely lehetőséget ad arra, hogy objektumokon keresztül történjen az adatkezelés. Ezek az objektumok a *Persistence Model*ben vannak definiálva, ami csak néhány dologban tér el a 2.1.3. fejezetben részletesen bemutatott *Domain Model*-től.

A *Repository* szintjén létezik egy `Images` nevű entitás is, ami a képeket alkotó byte sorozatokat is eltárolja, valamint rendelkezik egy `projectId` mezővel, ami egy projektre mutató külső kulcs (*foreign key*). A *Persistence Model*-ben szereplő `Project` nem csak a képekre mutató `id`-kat, hanem a teljes kép objektumokat tartalmazza.

A `Window`-ban a változókat egy `Variable` típusú lista reprezentálja. Ez az új típus rendelkezik egy `string` adattaggal a változó értékének a tárolására, ezenkívül pedig két külső kulccsal, az első egy ablak instanciára mutat, a második egy `TypedVariable` változóra. Ez a típus a domain modellben egy `Dictionary`-ra van *map*elve.

Ezekre a különbségekre azért van szükség, mert a projekt *code first* adatbázist használ. Ez

azt jelenti, hogy az EF core a megadott modellek alapján kigenerálja az adatbázist, ha az még nem létezik. Ahhoz, hogy a kigenerált struktúra pontosan kövesse a Woody megvalósításához szükséges elképzelést, be kell tartani a keretrendszer konvencióit. Azért van külön *Domain Model* a projektben, hogy ezeket a konvenciókat ne kelljen továbbvinni az üzleti logikába.

2.1.5. Optimális vágás

Az ablakkeretet alkotó faelemek adott magasságú és mélységű, de változó hosszúságú deszkákból való kivágása során felmerül a kérdés, hogy létezik-e erre egy optimális vágási stratégia. Habár ennek meghatározása csak egydimenziós probléma, mégis egy NP-nehéz feladat [24].

A Woody-ban egy randomizált, heurisztikus algoritmus próbálja megadni a deszkák optimális darabolását egy-egy projekt számára. Az algoritmusról nincs bizonyítva, hogy meghatározza a leghatékonyabb megoldást, de lényegesen csökkenti a fafeldolgozó műhely anyagveszteségét.

A megrendelés ablakaihoz tartozó faelemek magasságuk és mélységük szerint vannak csoportosítva, ezekre a csoportokra külön-külön fut le az algoritmus.

Első lépésben a faelemeket véletlenszerű sorrendbe rendezi a program, majd erre a sorrendre számol veszteséget. Minden raktáron levő deszkahosszúságra megvizsgálja, hogy a jelenlegi faelemek közül mennyit lehetne kivágni belőle (a prioritást a véletlenszerű rendezés határozza meg) és hogy mennyi lenne a veszteség. A legjobb esetet megtartja. Ezt addig ismételi, amíg minden faelemet fel nem használt. Ezeket a lépéseket az algoritmus többször elvégzi (ennek a számát egy konstans határozza meg, amelynek ebben az esetben 1000 az értéke) és azt a stratégiát tartja meg, amelyik esetében a legkisebb a veszteség.

A kapott eredmény szemléltetése céljából a *NyZLNy* segítségével egy SVG is készül. Ennek kinézetét az 5. fejezetben meg lehet tekinteni.

2.2. A webkliens

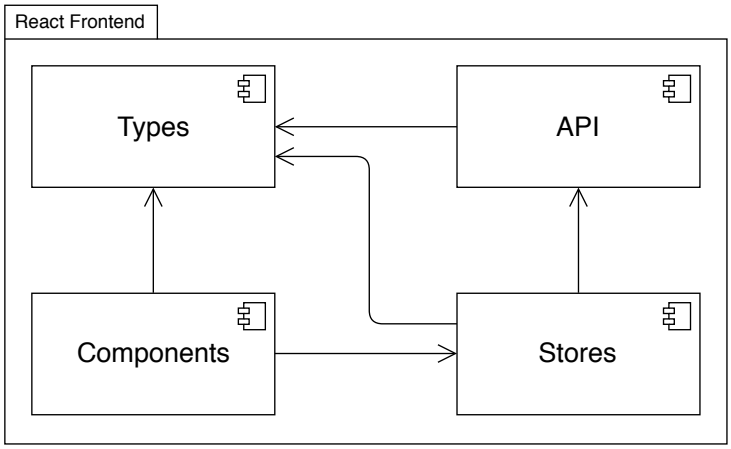
2.2.1. Architektúra

A webes kliens szerkezete négy fő komponensből áll (ld. 7. ábra):

A *Types* tartalmazza az entitások kliens oldali levetítését.

Az *API* modul felelős a kommunikációért a szerver és a kliens között. A hívások aszinkron jellegűek, a szervertől kapott választ pedig visszahívó függvények kezelik le, így az adatlekérés dinamikus. Ugyanebben a modulban történik a beérkező adatok entitássá alakítása, illetve a szerver fele kimenő adatok DTO-vá alakítása.

A *Stores* modul kezeli a kliensalkalmazás állapotát. Minden entitáshoz tartozik egy tároló (*store*), mely az adott entitással kapcsolatos adatokat és az ezekhez tartozó műveleteket gyűjti egybe. Az adatok lekérésére vagy módosítására az *API* modult használja.



7. ábra. A webes kliens négy fő komponense.

A *Components* tartalmazza a felhasználói felülethez szükséges összes webes komponenst. A felületről érkező bemenetek lekezelésére a *Stores* modult hívja meg, a tárolókban változott adatok alapján pedig frissíti ezek ábrázolását a felületen.

2.2.2. Egyoldalas webalkalmazás és útvonalválasztás

Az egyoldalas webalkalmazás (*Single-Page Application*) előnye az, hogy az első betöltés után nem szükséges újratölteni a teljes oldalt, hanem csak azt a részét kell frissíteni, amely módosult időközben [27]. A szükséges HTML, CSS és JavaScript forráskód betöltődik az oldal első megnyitásakor, az alkalmazásszerverrel pedig csak adatcsere érdekében kell kommunikálni. Így az alkalmazás gyorsabbá és egyben zökkenőmentesebbé válik a felhasználók számára.

A 3.2. fejezetben ismertetett React könyvtár elősegíti az alkalmazás ilyen módon való felépítését.

Mivel a webes kliens egy egyoldalas webalkalmazás, az elérési útvonal dinamikus változásával lehet követni, hogy a felhasználó éppen melyik részét használja az alkalmazásnak. Ennek kezelése érdekében szükség van egy útvonalválasztó (*routing*) mechanizmusra.

Az útvonalak változása által a különböző állapotokat a böngésző elmenti az előzményekbe, így könnyebbé téve a visszalépést és a visszakeresést. Az útvonalválasztás a megjeleníteni kívánt komponensek kiválasztásában is segít, például egy esetleges újratöltés után a felhasználót ugyanaz a nézet fogadja, mint az újratöltés előtt.

2.2.3. Komponensek

A felhasználói felületet komponenseknek nevezett kisebb egységek alkotják. Mivel a végső nézet több komponens egymásba ágyazásával alakul ki, ez megengedi az általános célú komponensek újrahasználását a különböző nézetek felépítése során.

A komponensek kétfélék lehetnek: *stateless* (állapot nélküli) és *stateful* (állapottal rendelke-

ző). A *stateful* komponensek automatikusan újragenerálják a nézetüket, amint változás történik az állapotukat alkotó adatok szintjén.

A komponensek dinamikus változása az adatok szerint megkönnyíti a bonyolult felületek kialakítását. A komponensek frissítése történhet kizárólag kliensoldalon, anélkül, hogy a kezdeti adatlekérés után szükség legyen újabb hálózati hívásokra. Egy példa erre az újabb nyílászáró hozzáadása egy rendeléshez. Ekkor a termék típusa meghatározza, hogy milyen változók szükségesek a termék paraméterezéséhez. A változók listája alapján ki lehet generálni egy felületet, amelyen minden változóhoz tartozik egy beviteli mező. Mindegyik mező típusát a hozzá tartozó változó típusa határozza meg, ettől függően a mező lehet szöveges- vagy számbevitel, illetve legördülő lista.

3. Felhasznált technológiák és eszközök

Az alkalmazás szervere C#-ban íródott, az ASP.NET Core keretrendszer felhasználásval, a webes kliens pedig TypeScriptben, a React könyvtár segítségével. Ebben a fejezetben a felhasznált technológiák és eszközök kerülnek bemutatásra.

3.1. Szerveroldali technológiák

PostgreSQL

A PostgreSQL egy nyílt forráskódú, relációs adatbázis-kezelő rendszer, amely népszerűségét architektúrájának, megbízhatóságának, funkcióinak és bővíthetőségének köszönheti [11]. Lehetőséget nyújt saját adat- és indextípusok definiálására, sajátos bővítmények írására.

Az adatokat táblákban tárolja, ahol az oszlopok az entitás attribútumait képviselik (pl.: név, kor), a sorok pedig az egy entitáshoz tartozó adatokat tárolják. Ez a szerkezet a szigorúan meghatározott adatok reprezentálására alkalmas.

ASP.NET Core

Az ASP.NET Core egy nyílt forráskódú, cross-platform keretrendszer webalkalmazások fejlesztésére, amelynek alapja a .NET Core (Windows esetén lehet a .NET Framework is). A technológia hivatalos tájékoztatója [25] szerint „a teljesítmény kulcsfontosságú az ASP.NET Coreban. Gyorsabb, mint más népszerű web keretrendszerek a független TechEmpower felmérése szerint.” A 8. ábrán[3] látható a .NET összehasonlítása a Java Servletekkel, illetve a Node.js-el teljesítmény alapján.

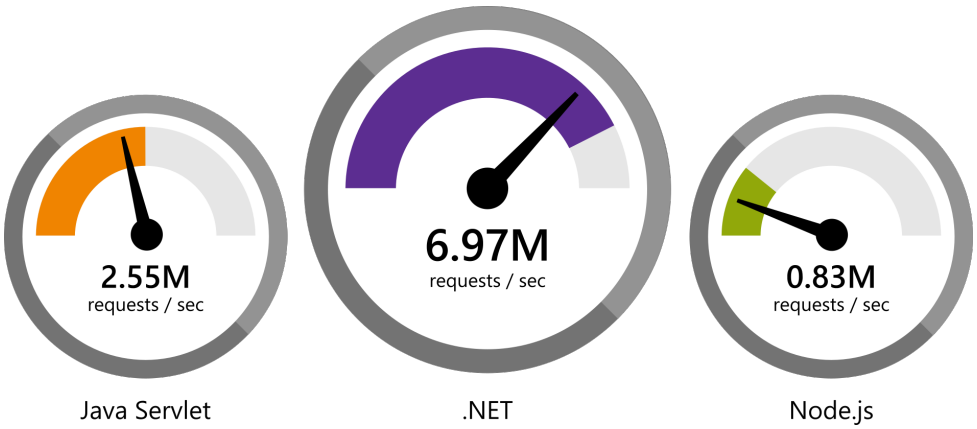
A keretrendszer beépített naplózással és beállításokkal (a kódon belül bárhol el lehet érni a a *Configuration*-t az *IConfiguration* interfészen keresztül) rendelkezik. Támogatja az aszinkron programozási elveket, amely nagymértékben hozzájárul a gyorsaságához. Ugyanakkor lehetőséget ad különböző környezetek (*development, production, stb.*) esetében más-más viselkedés meghatározására.

Habár az ASP.NET Core ingyenes a Microsoft mégis hivatalos, teljes támogatást biztosít hozzá.

Entity Framework Core

Az Entity Framework Core egy kibővíthető, nyílt forráskódú, cross-platform ORM (*object-relational mapper*) keretrendszer, amely lehetővé teszi a fejlesztők számára, hogy objektumo-

[3]Forrás: `https://dotnet.microsoft.com/images/redesign/shared/tech-empower-results.svg`, utolsó megtekintés dátuma: 2018-04-05



8. ábra. ASP.NET Core összehasonlítása a Java Servletekkel és a Node.js-szel teljesítmény alapján

kon keresztül kommunikáljanak az adatbázissal, alacsony szintű SQL kód írása nélkül [4].

Az adatelérés egy modellen keresztül történik, amely entitásokat képviselő osztályokból, valamint egy kontextus objektumból áll. Code first adatbázis esetén a modellek alapján generálódik ki az adatbázis, de lehetőség van az adatbázis alapján a modell kigenerálása vagy mindkét fél megírására. Hogy megőrizze a modell egyszerűségét, az EF Core támogatja a táblázatok tördelését is: különböző entitásokra lehet bontani a táblákat oszlopok szerint, csak az elsődleges kulcs kell mindegyik objektumban szerepeljen.

Az adatokat a *Language Integrated Query (LINQ)* segítségével lehet elérni. Ennek a szintaxisa nem függ az adatbázis-kezelő rendszertől.

3.2. Web technológiák

React

A React egy felhasználói felületek készítésére alkalmas JavaScript könyvtár [13]. Az alapvető építőelemei a komponensek: különálló kis kódrészletek, melyek egymásba ágyazásával komplex felületeket lehet felépíteni. Ezek a komponensek egy leírást biztosítanak a felhasználói felület szerkezetéről, ezen leírások összessége alkotja a *Virtual Document Object Model*t (VDOM). A VDOM teszi lehetővé a tényleges DOM gyors és hatékony frissítését az alkalmazás belső állapotában történt változások alapján.

A felhasználói felület leírása történhet kizárólag JavaScriptben is, viszont javasolt a JSX (*JavaScript eXtension*) használata az átláthatóság és a könnyebb hibakeresés érdekében. A JSX a JavaScript szintaxis kiterjesztése, mely megengedi az XML-hez hasonló kód írását a React komponensekben, ezt majd egy előfeldolgozó átalakítja a megfelelő JavaScript utasításokká. Egy másik előnye a JSX-nek az, hogy az attribútumokon keresztül bármilyen típus átadható, nem csak a megszokott karakterlánc, így robusztusabb a kommunikáció a komponensek közt.

A weboldalon történő navigációt segíti elő a React Router [12], mely egy útvonalválasztó mechanizmus React alkalmazások számára. A könyvtár lehetőséget nyújt különböző tartalom betöltésére az elérési útvonaltól függően, illetve a React Router Dom csomag biztosít navigációra alkalmas komponenseket, melyek által dinamikusan változtatható az elérési útvonal az oldal újratöltése nélkül.

TypeScript

A TypeScript [2] egy kiterjesztése a JavaScript nyelvnek, amely lehetőséget ad típusok, osztályok és interfészek használatára. Ezek fontos előnyt jelentenek fejlesztés során, mert a fejlesztői környezetek a típusdefiníciók alapján képesek jelezni az esetleges hibákat, vagy jobb javaslatokat tudnak nyújtani kódírás közben. Ezek a megkötések nem jelentenek hátrányt futtatáskor, ugyanis a TypeScript *transpiler*e standard JavaScriptre fordítja vissza a forráskódot.

Mivel a TypeScript kiterjeszti a JavaScriptet, ezért egy helyes JavaScript kód is helyes TypeScript kódnak számít. A JavaScript alapú könyvtárak használata esetén szükség van viszont egy típusdefiníciókat tartalmazó állományra, mely leírja a könyvtár szerkezetét és típusait a *transpiler* számára.

MobX

A webes kliens állapota több olyan adatot is magába foglal, melyek az alkalmazás több különböző részét is befolyásolják, mint például a projektek vagy a kliensek listája. Ha mindezt a komponensek kezelnék, az adatmegosztás és a karbantartás nehézkessé válhat.

Az alkalmazás állapotának kezelésére való a MobX könyvtár, ami funkcionális és reaktív programozási elvek alapján teszi egyszerűvé az állapotkezelést [9]. Alapgondolata így szól: „Mindent, amit az alkalmazás állapotából származtatni lehet, származtatni is kell. Automatikusan.”. A könyvtár az adatokat megfigyelhetővé (*observable*) teszi, ezek változására tudnak figyelni és reagálni a figyelők (*observer*). Ezáltal megszűnnek a tárolt és a megjelenített adatok közti eltérésekből fakadó problémák.

A könyvtár készítői azt a mintát ajánlják, hogy az alkalmazás állapotát alkotó adatok központi tárolókban (*store*) legyenek kezelve [8], egy tárolóba a szorosan összefüggő adatok és az ezekhez tartozó műveletek kerülnek. Így kiemelhető az állapot a komponensek szintjéről egy könnyebben átlátható és tesztelhető egységbe, valamint egyszerűbb megosztani ugyanazt az állapotot több független komponenssel.

InversifyJS

Az ECMAScript 2015 óta, valamint a TypeScript által lehetőség van objektumorientált programozásra JavaScriptben. Az osztályok közti szoros kapcsolás elkerülése érdekében szükség van a *dependency injection* (DI) tervezési mintára [21], amely által egy különálló objektum példányosítja és kezeli más objektumok függőségeit. Erre ad egy megoldást az InversifyJS [7], ami egy *inversion of control* (IoC) konténert biztosít JavaScript és TypeScript alkalmazások számára. Elősegíti a MobX és a React közötti együttműködést, általa lehetőség van beágyazni a MobX tárolóit a figyelő komponensekbe. A *store*-ok létrehozásában is segít a függőségek beszúrása által.

Bootstrap

A Bootstrap a legnépszerűbb CSS keretrendszer *responsive* felhasználói felületek tervezésére [3]. A keretrendszernek az egyik erőssége a rácsalapú rendszere, amiben alapértelmezetten 12 oszlopra oszt egy sort, az egyes komponensek pedig a képernyőméret alapján több vagy kevesebb oszlopot foglalhatnak el, vagy akár el is tűnhetnek. Így egyidejűleg lehet több képernyőméretre tervezni a felhasználói felületet. Emellett a Bootstrap magával hoz több előre meghatározott stílusú komponenst is. Ezeket egy külső könyvtár lévén lehet összekötni a Reacttal.

styled-components

Mivel a React egy komponensalapú könyvtár, ezért előnyös lehet a stílusokat a komponensek szintjén meghatározni. Ezt teszi lehetővé a styled-components [16] könyvtár, ami által a JavaScript kódban lehet a komponensekhez kötni a hozzájuk tartozó CSS szabályokat. Ezek a szabályok függhetnek a komponensek *property*-jeitől vagy más változóktól is, így könnyen lehet például színtémát meghatározni az alkalmazás szintjén. A styled-components az így keletkezett CSS szabályokat a komponensek *hash*-je alapján generált egyedi osztálynevekhez köti, így biztosan ütközésmentes lesz a szabálykészlet.

3.3. Eszközök

Git

A Git egy osztott verziókövető rendszer, mely elősegíti a fejlesztési folyamatot, lehetőséget adva a változtatások számontartására és az egyidejű fejlesztésre. A Git más verziókövető rendszerekkel ellentétben (CVS, Subversion, stb.) nem a különbségéket tárolja el a verziók között, hanem mindig a projekt jelenlegi állapotának a képét rögzíti. A változatlan állományok esetén

egy mutatót tárol az előző azonos állományra [1]. A Git egy másik előnye az, hogy a teljes *repository* másolata lokálisan is megtalálható, így a legtöbb művelet internetkapcsolat nélkül is elvégezhető.

GitLab

A Gitlab egy egységes projektmenedzsment felület alkalmazások fejlesztésére, eszközöket biztosítva a *DevOps* életciklus minden stádiumához [6]. Lehetőséget ad a Git alapú tárolók kezelésére, wiki oldalak készítésére, folyamatos integráció és folyamatos kitelepítés megvalósítására, taszkok követésére és forráskód felülvizsgálatára.

Docker

A Docker [26] egy alkalmazások futtatására és kitelepítésére használt virtualizációs eszköz, ami biztosítja az azonos futási környezetet a céleszköztől függetlenül. Ezt az ún. *container*ek segítségével teszi lehetővé, melyek egy virtuális futási környezetet hoznak létre, de a virtuális gépekkel ellentétben kihasználják a gazda Linux gép kernelét, így jelentősen gyorsabbak ezeknél. A *container*ek tartalmazzák az alkalmazást és ennek összes függőségét egy egységként, így kitelepítéskor biztosított az azonos futási környezet.

Webpack

A webes alkalmazás függőségeinek összesítésére alkalmas a Webpack [15], ami egy statikus csomagkészítő JavaScript alkalmazások számára. A számos forrásállományt, illetve az npm csomagkezelő által begyűjtött függőségeket és ezek tranzitív függőségeit egy vagy akár több csomagba összesíti. Így biztosítja azt, hogy az alkalmazás működéséhez szükséges összes alkotóelem jelen van, miközben lecsökkenti az alkalmazás betöltéséhez szükséges hálózati kérések számát. Lehetőség van külön csomagokba választani a forráskódot és a külső függőségeket, így javítva a *cache*-elési lehetőségeket. A futtatáshoz elegendő ezeket a csomagokat átküldeni a kliens böngészőknek.

NUnit

Az NUnit egy tesztelési keretrendszer .NET alapú nyelvekhez, melyet eredetileg a Javaban írt alkalmazások tesztelésére szolgáló JUnit alapján írtak meg [10]. Az NUnit egy széles körben használt, nyílt forrású keretrendszer [20], mely lehetőséget nyújt automatizált tesztesetek futtatására, a tesztesetek paraméterezésére, illetve a hatékonyság érdekében jelezni lehet számára, hogy mely teszteseteket vagy tesztosztályokat futtathatja párhuzamosan.

Statikus kódellenőrzők

A statikus kódellenőrző eszközök a kód minőségét ellenőrzik az adott nyelvre specifikus szabályok szerint, amelyek vonatkozhatnak az olvashatóságra, a karbantarthatóságra vagy akár funkcionalitási hibák észlelésére is. A kódellenőrzők segítségével el lehet kerülni a gyakori kódolási hibákat, illetve biztosítani lehet a projekt szintjén megegyezett kódolási konvenciók betartását.

A TSLint [14] egy ilyen eszköz TypeScript forrásállományok számára, míg a StyleCop egy ingyenes és könnyen használható ellenőrző C# alkalmazásokhoz [22]. Az utóbbi a `StyleCopAnalyzers` csomag révén építhető be a *build* folyamatba. Mindkét eszköznél a bekapcsolt szabályok esetén beállítható, hogy figyelmeztetést vagy hibát jelentsen az adott szabály megszegése.

4. Munkamódszerek

A fejlesztés elősegítése érdekében szükség van bevált módszerekre és pontos folyamatok kialakítására. Ez a fejezet részletezi a fejlesztési munkamenet különböző mozzanatait.

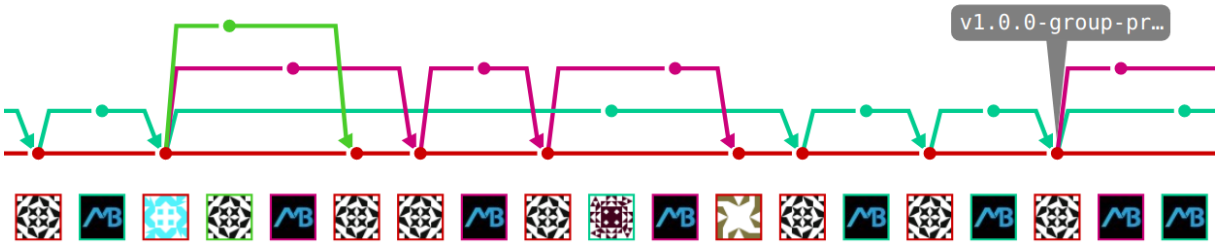
4.1. Scrum és Kanban

A csapat munkamenetének felépítésében jelentős szerepet játszott a Scrum [18]. Ez egy elterjedt agilis módszer, mely inkrementális és iteratív fejlesztést javasol. A fejlesztés *sprint*nek nevezett szakaszokban történik, melyek általában két- vagy háromhetesek. A szakaszok kezdetekor a *sprint planning* során felmérésre kerülnek a teendők, majd relevancia és bonyolultság szerint rangsorolva a *sprint backlog*ba kerül át ezekből annyi, amennyit a *sprint* végéig beláthatóan el lehet végezni. A fejlesztés ideje alatt heti megbeszélések alkalmával lehet kérdéseket tisztázni vagy komolyabb döntéseket meghozni a mentorok segítségével. A *sprint* végi gyűlésen megtörténik az elkészült funkcionalitások bemutatása (*demo*), illetve a visszatekintés (*retrospective*) is, amelynek keretein belül minden csapattag szóvá teszi a pozitív és negatív élményeit az elmúlt sprinttel kapcsolatban, illetve javasolhat változtatásokat a kellemetlenségek elkerülése érdekében a jövőre nézve.

A *sprint* ideje alatt a feladatok állapotának nyilvántartására hasznosnak bizonyul a *kanban board* [23]. Ez a Kanban agilis módszer része, viszont a Scrummal is jól párosítható. A *kanban board* egy vizuális felületet biztosít a munka haladásának nyomon követésére. Ez a felület több, állapotot jelző oszlopból áll, ezekben az oszlopokban egy kártya egy feladatot jelképez. A *sprint planning* alatt kiválasztott feladatok kerülnek a *sprint backlog* oszlopba. Minden csapattag kiválaszthatja és magához rendelheti a számára kézenfekvő feladatot, majd ezt a *doing* oszlopba helyezve jelezi, hogy elkezdett dolgozni rajta. A *review* oszlop a kész és ellenőrzésre váró feladatokat tárolja, a *merged* az ellenőrzött és elfogadott feladatokat. A bemutató során elfogadott teendők átkerülnek innen a *closed* oszlopba.

4.2. GitFlow

A Git által biztosított lehetőségek kihasználása és egy átlátható munkafolyamat érdekében a változások beépítése a kódbázisba a GitFlow alapján történt. A GitFlow [17] egy Gites elágazási startégia (*branching model*), melyet először Vincent Driessen ismertetett 2010-ben. A stratégia szerint két állandó *branch* létezik a projekt alatt, ezek a `master`, és `dev` vagy `develop`. A `master` ág mindig stabil, kitelepítésre kész kódot tárol, míg a `dev` az a központi *branch* a fejlesztés során. Minden újabb beállítás, funkcionalitás implementálása vagy hibajavítás külön *branch*-en történik (ld. 9. ábra), melyek nevében a kezdeti kulcsszó jelzi az adott *branch* típu-



9. ábra. A Woody alkalmazás Gites gráfjának egy részlete. A különböző funkcionalitások elágaznak a `dev`-ből és visszatérnek ide. Megfigyelhető a csoportos projekt végi állapot címkéje a `master` ágon.

sát (`init`, `feature`, `bugfix`, stb.). Ezek a `dev`-ből ágaznak el, a változások végeztével ide is épülnek vissza, ezután az ág megszűnik létezni.

4.3. Kódellenőrzés

A forráskód minőségének biztosításáért egyaránt szükséges az automatikus és kézi kódellenőrzés.

Automatikus ellenőrzést a 3.3. fejezetben ismertetett TSLint és StyleCop biztosítanak front- illetve backenden és jelzik amint egy kódrészlet megszegi a közösen meghatározott minőségi szabályokat.

A kézi kódellenőrzésre a GitLab nyújt egy megfelelő felületet. A különböző *branch*-ek visszacsatolását a `dev`-be ún. *merge request*eken keresztül lehet elvégezni. Ezek végrehajtásával a Git *merge*-eli a forrás ágat a cél ágba. A *merge request* felülete lehetővé teszi a változtatások átnézését és megjegyzések hozzáfűzését az egyes sorokhoz, így könnyen lehet jelezni egy esetleges hibát vagy javítási lehetőséget. A csapat munkamenetében minden *merge request*et átnézi a megfelelő mentor, illetve legtöbb esetben legalább még egy csapattag. A kód frissítése után az egyes megjegyzések szerzői újra leellenőrzik a változott részletet és jóváhagyják vagy tovább egészítik a megjegyzéseket. Amint az összes megjegyzés tisztázódott, a résztvevők jóváhagyásával a változtatások bekerülnek a cél *branch*-be, ami legtöbb esetben a `dev`.

4.4. Folyamatos integráció és kitelepítés

A folyamatos integrációnak (*Continuous Integration*) fontos szerepe van az alkalmazás fejlesztése során, mivel lehetővé teszi az automatikus ellenőrző lépések futtatását minden újabb változtatás esetén, így azonnal jelezve az esetleges hibákat. A GitLab CI/CD [5] segítségével ún. *pipeline*-ok határozhatóak meg, melyek különböző tevékenységek sorozatai. A *pipeline*-ok különböző események bekövetkeztekor futnak le, például a szerver esetében minden újabb GitLab-ra küldött változtatás után meghívódik a *build* folyamat, mely ellenőrzi, hogy kompilálható-e

```
services:
  woody-frontend:
    container_name: woody-frontend
    depends_on:
      - woody-backend
    ports:
      - 8082:80
    # ...

  woody-backend:
    container_name: woody-backend
    depends_on:
      - woody-db
    # ...

  woody-db:
    image: postgres:10.6-alpine
    container_name: woody-db
    # ...
```

10. ábra. Részlet a docker-compose konfigurációs állományából

az alkalmazás. Ha ez sikeres volt, akkor utána a StyleCop ellenőrzés és az automatikus tesztek következnek. Egy *merge request* csak akkor fogadható el, ha a teljes *pipeline* sikeres volt.

A CI-t követi a CD (*Continuous Delivery*), ami a kitelepítés automatizálását foglalja magába. Ez egy olyan folyamat, mely akkor fut le, amikor egy *merge request* révén bekerülnek új változtatások a `dev` ágba, ekkor egy újabb *docker image* jön létre a legfrissebb változtatásokkal. Ez az *image* pedig bekerül a GitLab Container Registry-be, amely egy privát tároló *docker image*-ek számára. A *docker image* állományrendszerek rétegeiből épül fel és ez alapján jön létre egy vagy akár több *docker container*.

A dockerizációs lépés a szerver és webes tárolók esetén fut le. Mivel a Formula Parser egy külön tárolóban található, ezért Git *submodule* által kell beépíteni a szerver tárolóba. Így külön lehet fejleszteni ezt az egységet, viszont teszteléskor és kitelepítéskor úgy működik, mintha a szerver lokális fájlrendszerének a része lenne.

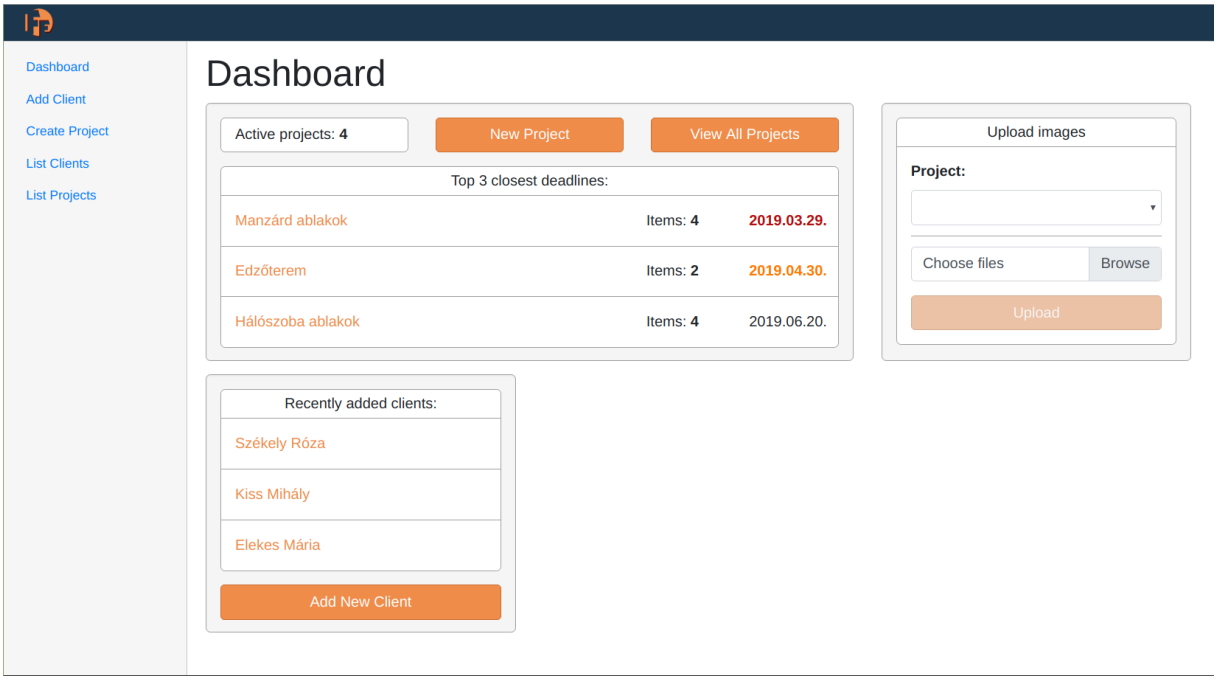
A szerver és web között összekapcsoló elemként működik a `woody-compose` tároló, ahonnan ki lehet telepíteni az alkalmazást a megfelelő tesztszerverre. Ezáltal a csapat összes tagja ugyanazon a példányon dolgozhat, illetve könnyebb a mobil nézetek tesztelése is valódi eszközökről. A kitelepítés a docker-compose segítségével lehetséges, ami egy eszköz több, együttműködő *docker container*ből álló alkalmazások futtatására. Az eszköz egy belső hálózatot hoz létre a alkalmazásszerver, webszerver és adatbázis számára (ld. 10. kódrészlet), a külvilág fele pedig csak egyetlen *port*ot nyit meg (a szemléltetett példában a `8082`-t), ami a webalkalmazásra mutat.

5. Az alkalmazás működése

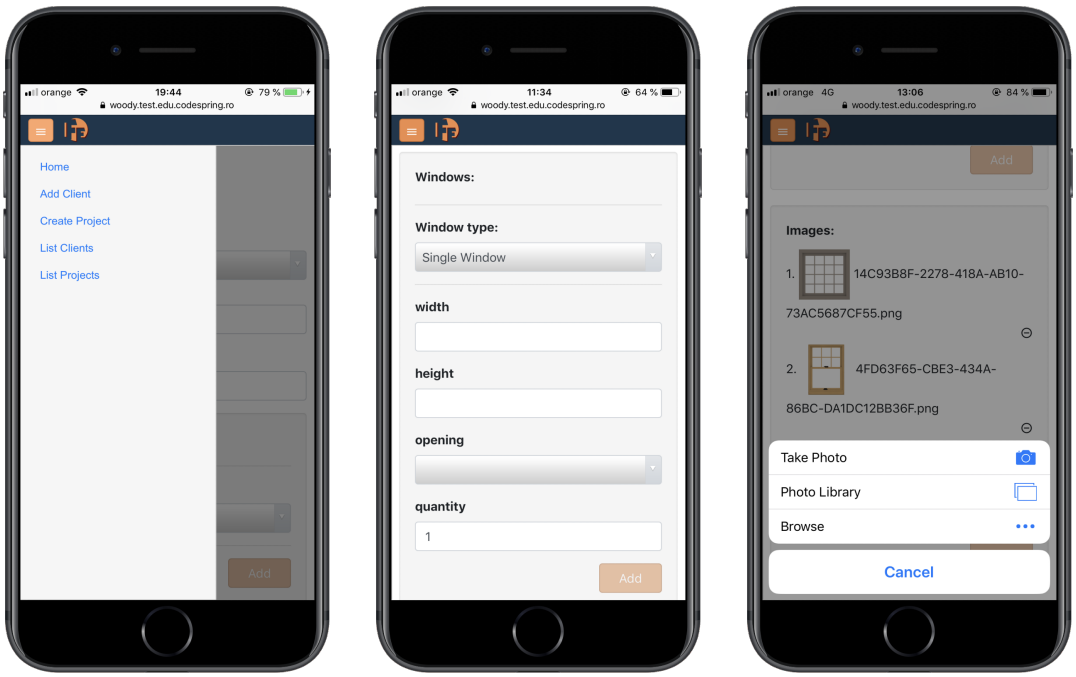
A weboldal megnyitásakor a *Dashboard* fogadja a felhasználót, amely könnyű hozzáférést biztosít az alkalmazás leghasználtabb funkcionalitásaihoz, a 11. ábrán látható módon. Az első téglalapban található egy-egy link azokra a megrendelésekre, amelyeknek a legközelebbi a határideje. Ugyancsak itt található két gomb, az egyik a projektek létrehozásához irányít át, a másik pedig a projektek listájához. A következő egység a képek csatolását hivatott megkönnyíteni. Itt a felhasználó a legördülő listából kiválaszthat egy projektet, majd a *Browse* gombra kattintva egy vagy több képet tud csatolni hozzá. Ezek alatt található egy harmadik téglalap is, amely magában foglalja a nemrég létrehozott kliensekre mutató linkek listáját és egy gombot, ami egy új kliens hozzáadásához irányít át.

A *Dashboard*, akárcsak az oldal többi része, reszponzív elv alapján van megvalósítva, hogy kisebb képernyőkön is ugyanolyan kényelmesen lehessen használni. Mobilon a kezdőoldal téglalapjai egymás alatt helyezkednek el.

A weboldal menüje nagyobb képernyőkön a kijelző bal oldalán található (13. ábra), kisebbeken, mint például telefonokon, a képernyő bal felső sarkában levő, narancssárga gombra ("hamburgerre") kattintva érhető el. Az utóbbi esetben a menü elfoglalja a képernyő bal felét (12a. ábra), ez akkor tűnik el, ha a felhasználó valamelyik menüpontra, a "hamburgerre" vagy a menü mellé kattint. A menü segítségével a felhasználó könnyedén navigálhat a weboldalon, a megfelelő menüpontot kiválasztva a hozzátartozó oldal jelenik meg. 5 opció közül lehet választani: az előbb említett dashboard, kliens hozzáadása, projekt létrehozása, kliensek listázása és projektek megtekintése.



11. ábra. *Dashboard* - A felhasználót fogadó weboldal



(a) Lenyitott oldalmenü mobil eszközön (b) A kiválasztott ablaktípushoz tartozó beviteli mezők (c) Képek készítése és feltöltése projekthez

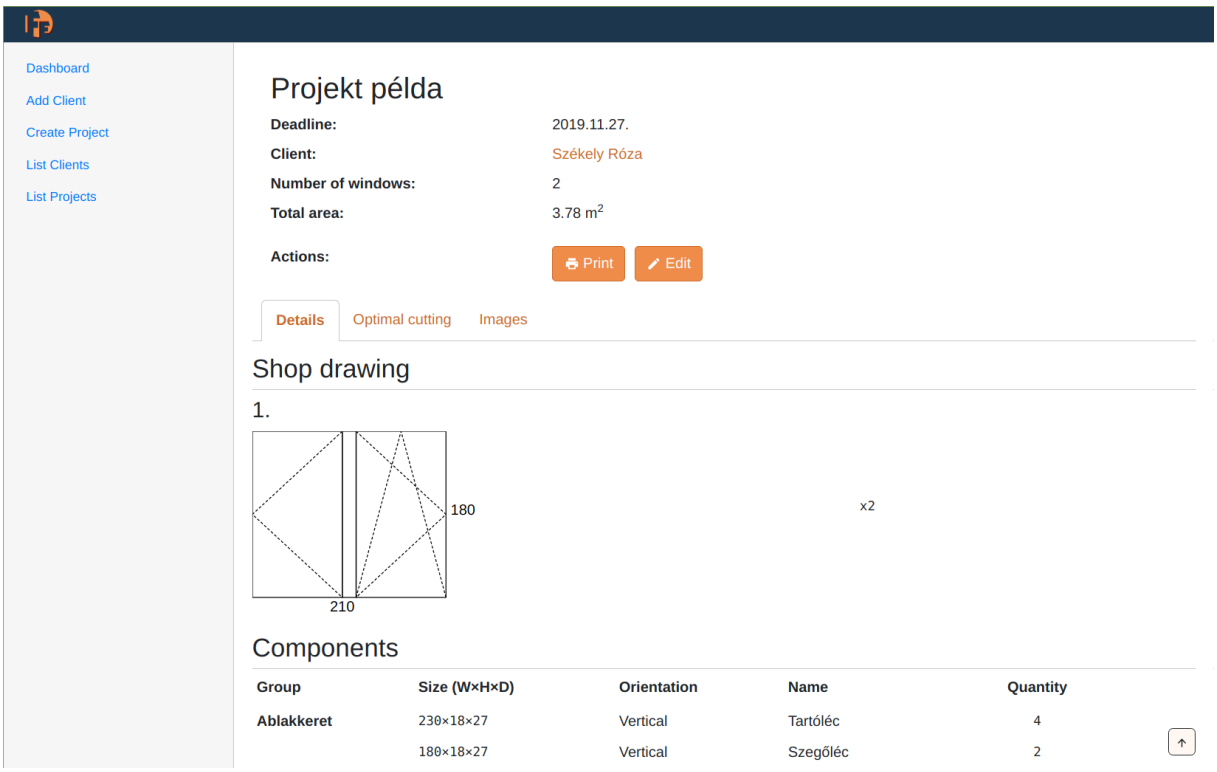
12. ábra. A weboldal kinézete mobil eszközön

A kliens hozzáadásakor egy négy mezőből álló űrlapot kell kitölteni. A név, telefonszám és lakcím mezők kitöltése kötelező (ezt egy-egy narancssárga csillag jelzi), míg az e-mail megadása opcionális. Az *Add* gomb csak akkor válik elérhetővé, ha minden bevitt adat érvényes, vagyis ha telefonszám, e-mailcím még nem szerepelnek a rendszerben (a megrendelők duplikálásának elkerülése végett) és ha az összes kötelező beviteli mező ki van töltve. Az *Add* gombbal lehet elmenteni az új klienst, ha a hozzáadás sikeres volt, akkor egy kis zöld üzenetdoboz jelenik meg a képernyő jobb alsó sarkában, ha viszont hiba történt a művelet elvégzése során, akkor ugyanitt egy piros üzenetdoboz jelenik meg, a megfelelő hibaüzenettel.

A kliensek kilistázásakor a klienseknek az adatait lehet megtekinteni egy táblázatban. Egy kliensre kattintva megjelennek a hozzá tartozó rendelések, árajánlatok és egy *Edit* gomb, amely a kliens adatainak módosítását teszi lehetővé.

Ha a felhasználó a *Create Project* menüpontot választja, akkor egy erre a célra megalkotott űrlapot kell kitöltsön. Minden megrendeléshez kell tartozzon egy megrendelő, akinek az adatai már a rendszerben vannak, a megfelelő klienst egy legördülő listából lehet kiválasztani. Emellett még kötelezően meg kell adni egy nevet és egy határidőt, az egyszerűség kedvéért a dátumot egy kis naptárnézetből is ki lehet választani. A *Save* gomb csak akkor lesz elérhető, ha az előbb említett adatok meg lettek adva. A gombra kattintva lehet elmenteni a projektet. A művelet sikerességét a kliensek hozzáadásánál leírt módon jelzi a weboldal a felhasználó számára.

Egy új projekt létrehozásakor új ablakokat és képeket is lehet csatolni a megrendeléshez. A felhasználó egy legördülő listából tudja kiválasztani, hogy milyen típusú ablakot szeretne



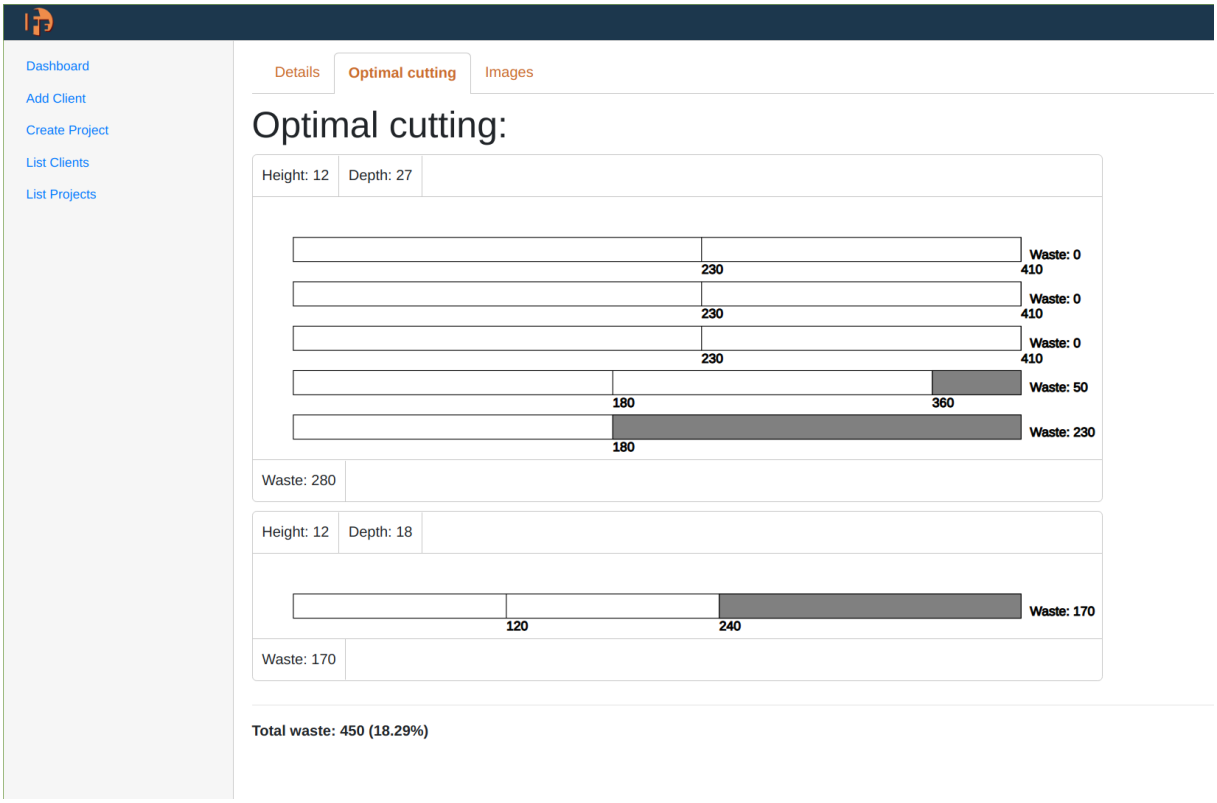
13. ábra. *Details* - Projektekhez tartozó részletes információnézet műhelyrajzzal és szabásjegyzékkel

hozzáadni a projekthez. Ha ez megtörtént, akkor megjelennek az adott típusnak megfelelő beviteli mezők, amelyek különböző változókat képviselnek. A 12b. ábrán látható a *Single Window* kiválasztása után megjelenő ürlap. Az *Add* gomb csak akkor válik elérhetővé, ha minden változóhoz került érték. A gomb lenyomása után az ablak a megrendeléshez tartozó "kosárba" kerül. A kosár tartalma a *Windows* alcím alatt jelenik meg. Egy-egy ablakot a mellette levő kicsi kerek gombbal (amelyen egy "-" jel található) lehet eltávolítani innen.

Képek csatolásakor a *Browse* gombra kattintva a felhasználó választhat, hogy az eszközről melyik képet szeretné feltölteni. Mobil esetén a 12c. ábrán látható módon új kép készítése mellett is lehet dönteni. Az *Add* gomb megnyomásával a kép bekerül egy kosárba, amely hasonlóan működik az ablakok esetében ismertetett kosárhoz.

Az utolsó menüpont a projektek listázása. Ezt választva a felhasználó megtekinthet egy összesítő táblázatot, amelyben néhány adat (projekt neve, kliens neve, határidő, ablakok száma) jelenik meg a projektekről. Bővebb információért a *Details* gombra kell kattintani, ekkor a 13. ábrán látható nézet fogadja a felhasználót. Az oldal tetején továbbra is láthatóak a projekthez tartozó adatok, illetve megjelennek a lehetséges műveleteket jelző gombok és egy *tab bar*.

Három tab közül lehet választani: *Details, Optimal Cutting, Images*. Az első az alapértelmezetten kiválasztott nézet, ezen a megrendeléshez tartozó ablakok ábrái (műhelyrajz) és az őket alkotó komponensek csoportosított listája (szabásjegyzék) látható. Ezt a felhasználó ki is tudja nyomtatni a műveleteknél megjelenő *Print* gombra kattintva. A nyomtatott formátum a fejléc



14. ábra. Optimális vágási stratégia

adatait is tartalmazza egy összefoglaló táblázat formájában.

A második opció az optimális vágási stratégiát szemlélteti, a 14. ábrán látható módon. A deszkák mélység és szélesség szerint vannak csoportosítva, minden csoportra külön meg van jelenítve a veszteség, a weboldal legalján pedig az összesített veszteség is megtekinthető. Ezt a nézetet is ki lehet nyomtatni. A harmadik tabot választva a projekthez tartozó képeket tekintheti meg a felhasználó.

Mindvégig látható a műveleteknél egy *Edit* gomb. Erre kattintva az épp kiválasztott projektet lehet módosítani. A módosítás lehetőséget ad a projekt nevének, határidejének és a megrendelőjének megváltoztatására, ugyanakkor lehetővé teszi az új képek, ablakok csatolását és a régiek törlését.

6. Következtetések és továbbfejlesztési lehetőségek

A jelen dolgozatban bemutatott Woody projekt fejlesztése során sikerült egy olyan szoftverrendszert létrehozni, mely egy webes felületet biztosít a fafeldolgozó műhelyek rendeléseinek a kezelésére, valamint a nyílászárók gyártásának megtervezésére.

A műhely szükségletei és a fejlesztés során felmerült ötletek alapján több továbbfejlesztési lehetőség is megfogalmazódott, mint például:

- vizuális szerkesztőfelület új ablaktípus létrehozására, frissíthető műhelyrajz előnézettel változtatható paraméterezés mellett;
- egy *Anyagszükségletek* nézet, mely a típusból kiindulva listázza a szükséges nyersanyagokat, több összesítést magába foglalva, úgy mint:
  - Vasalatmegrendelő táblázat
  - Üvegmegrendelő táblázat
  - Tömítőgéder mennyiség
  - Festékszükséglet
  - Szereléshez szükséges kellékek (csavar, purhab, stb.)
- kliensekhez hozzáfűzhető megjegyzések tárolása, melyekhez akár projekt is kapcsolódhat, árajánlatok, különleges igények és levelezés nyilvántartására és könnyű visszakeresésére;
- naptárnézet a projektek határidejeivel.

A projekt egyik alapötlete egy olyan alkalmazás készítése volt a faipar számára, ami elég általánosan írja le a nyílászárók gyártási folyamatát ahhoz, hogy könnyű legyen ezt karbantartani, valamint bővíteni a forráskód újrafordítása nélkül. Ennek eredményeképp más műhelyek is könnyen hasznát vehetik a dolgozatban bemutatott szoftverrendszernek, miután a saját ajtó- és ablaktípusaikat definiálják a bemutatott szakterület-specifikus nyelvben.

Hivatkozások

[1] Scott Chacon és Ben Straub. *Pro Git*. 2nd. Berkely, CA, USA: Apress, 2014, oldal 13–15. ISBN: 9781484200773.

[2] Hivatalos TypeScript dokumentáció. URL: https://www.typescriptlang.org/ (utolsó elérés dátuma: 2019. márc. 26.)

[3] Hivatalos Bootstrap dokumentáció. *Introduction · Bootstrap*. URL: https://getbootstrap.com/docs/4.3/getting-started/introduction/ (utolsó elérés dátuma: 2019. márc. 26.)

[4] Hivatalos EF Core dokumentáció. *Overview - EF Core | Microsoft Docs*. URL: https://docs.microsoft.com/en-us/ef/core/ (utolsó elérés dátuma: 2019. márc. 29.)

[5] Hivatalos GitLab dokumentáció. *GitLab Continuous Integration & Delivery*. URL: https://about.gitlab.com/product/continuous-integration/ (utolsó elérés dátuma: 2019. ápr. 2.)

[6] Hivatalos GitLab dokumentáció. *What is GitLab?* URL: https://about.gitlab.com/what-is-gitlab/ (utolsó elérés dátuma: 2019. márc. 31.)

[7] Hivatalos InversifyJS dokumentáció. *InversifyJS a powerful IoC container for JavaScript apps powered by TypeScript*. URL: http://inversify.io/ (utolsó elérés dátuma: 2019. márc. 26.)

[8] Hivatalos MobX dokumentáció. *Defining data stores | MobX*. URL: https://mobx.js.org/best/store.html (utolsó elérés dátuma: 2019. márc. 26.)

[9] Hivatalos MobX dokumentáció. *Introduction | MobX*. URL: https://mobx.js.org/index.html#introduction (utolsó elérés dátuma: 2019. márc. 26.)

[10] Hivatalos NUnit dokumentáció. URL: https://nunit.org/ (utolsó elérés dátuma: 2019. márc. 31.)

[11] Hivatalos PostgreSQL dokumentáció. *PostgreSQL: About*. URL: https://www.postgresql.org/about/ (utolsó elérés dátuma: 2019. márc. 28.)

[12] Hivatalos React Router dokumentáció. *React Router: Declarative Routing for React.js*. URL: https://reacttraining.com/react-router/core/guides/philosophy (utolsó elérés dátuma: 2019. ápr. 27.)

[13] Hivatalos React.js dokumentáció. *Tutorial: Intro to React*. URL: https://reactjs.org/tutorial/tutorial.html#what-is-react (utolsó elérés dátuma: 2019. márc. 25.)

[14] Hivatalos TSLint dokumentáció. *TSLint*. URL: https://palantir.github.io/tslint/ (utolsó elérés dátuma: 2019. márc. 31.)

[15] Hivatalos Webpack dokumentáció. *Concepts | webpack*. URL: https://webpack.js.org/concepts (utolsó elérés dátuma: 2019. márc. 25.)

[16] Hivatalos styled-components dokumentáció. *styled-components: Basics*. URL: https://www.styled-components.com/docs/basics (utolsó elérés dátuma: 2019. ápr. 11.)

[17] Vincent Driessen. *A successful Git branching model*. 2010. URL: https://nvie.com/posts/a-successful-git-branching-model/ (utolsó elérés dátuma: 2019. ápr. 1.)

[18] Claire Drumond. *Scrum - what it is, how it works, and why it's awesome*. URL: https://www.atlassian.com/agile/scrum (utolsó elérés dátuma: 2019. ápr. 1.)

[19] Roy T Fielding. *Architectural styles and the design of network-based software architectures*. University of California, Irvine Irvine, USA, 2000. 5. fejezet.

[20] Jack. *Unit testing frameworks in C#: Comparing XUnit, NUnit and Visual Studio*. 2017. URL: https://raygun.com/blog/unit-testing-frameworks-c/ (utolsó elérés dátuma: 2019. ápr. 29.)

[21] Thorben Janssen. *Design Patterns Explained – Dependency Injection with Code Examples*. 2018. URL: https://stackify.com/dependency-injection/ (utolsó elérés dátuma: 2019. ápr. 27.)

[22] Michael Parker. *Linting C# in 2019 – StyleCop, Sonar, Resharper, Visual Studio and Roslyn*. 2019. URL: https://medium.com/@michaelparkerdev/linting-c-in-2019-stylecop-sonar-resharper-and-roslyn-73e88af57ebd (utolsó elérés dátuma: 2019. ápr. 29.)

[23] Max Rehkopf. *What is a Kanban Board?* URL: https://www.atlassian.com/agile/kanban/boards (utolsó elérés dátuma: 2019. ápr. 1.)

[24] Guntram Scheithauer és Johannes Terno. „A branch&bound algorithm for solving one-dimensional cutting stock problems exactly". *Applicationes Mathematicae* 23. évfolyam, 2. szám (1995), 151–167. oldal.

[25] Hivatalos ASP.NET Core tájékoztató. *What is ASP.NET Core*. URL: https://dotnet.microsoft.com/learn/web/what-is-aspnet-core (utolsó elérés dátuma: 2019. márc. 29.)

[26] *What is Docker?* URL: https://opensource.com/resources/what-docker (utolsó elérés dátuma: 2019. márc. 31.)

[27] Anastasia Z. *What's the Difference Between Single-Page and Multi-Page Apps*. 2018. URL: https://rubygarage.org/blog/single-page-app-vs-multi-page-app (utolsó elérés dátuma: 2019. márc. 23.)