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

Bordák Tamás

KOLLABORÁCIÓS KERETRENDSZER KÉSZÍTÉSE VALÓS IDEJŰ WEBES TECHNOLÓGIÁKKAL

KONZULENS

Albert István

BUDAPEST, 2020

Tartalomjegyzék

HALLGATÓI NYILATKOZAT

Alulírott **Bordák Tamás**, szigorló hallgató kijelentem, hogy ezt a 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(k), 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. 20.

...…….
Bordák Tamás

Összefoglaló

A mai világban az online jelenlét szinte nélkülözhetetlen, ebből táplálkoznak napjaink oly sikeresnek vélt közösségi média szolgáltatói. Ez nem meglepő, hiszen ez a legegyszerűbb módja annak, hogy üzenetünket tömegekhez juttassuk el.

A közösségi média használatát bárki meg tudja tanulni, pont ez az egyik legfontosabb erőssége. Egy másik szemszögből viszont rendkívül limitáló, mert minimális interakcióra ad lehetőséget, és a formátum is jóval korlátozottabb, mint a háttérben húzódó webes világ engedné.

Dolgozatomban bemutatom az általam készített keretrendszert és hozzá tartozó eszközkészletet, aminek legfontosabb célja, hogy a hétköznapi embert is közelebb hozza a webfejlesztés mesterségéhez.

Munkám során rendkívül fontosnak tartottam, hogy ne egy játékszert fejlesszek, ami a közérthetőségig alacsonyodik, hanem egy olyan fejlesztői eszközt készítsek, ami felveszi a versenyt a jelenleg standardnak mondható folyamatokkal.

Pontosan meghatároztam, hogyan épülnek fel az általam elképzelt speciális dokumentumok, és hogy milyen eszközökre van szükség ahhoz, hogy ezeket hatékonyan szerkeszteni lehessen.

A szerkesztés folyamatát a technika legmodernebb vívmányai segítségével keltettem életre, melynek eredménye, hogy egy dokumentumot egyidőben egyszerre többen is szerkeszthetnek. A változtatások zökkenőmentesen szinkronizálódnak, és a végső dokumentum felhasználók közös szándékát tükrözi.

Abstract

In today's world online presence is crucial, and that is feeding the success of the currently so relevant social media corporations. But that should not come as a surprise since this is the easiest way to get one's message to the masses.

Anyone can learn the use of social media, and that is one of its greatest advantages. From another perspective this medium is very limiting because it only allows very little interaction, and the format is also much more restricting than the underlying web technologies.

In my thesis I present the framework I developed and the toolkit that has the goal to bring everyday people closer to the craft of creating web interfaces.

Through my work I kept in mind that I am not creating a toy that dumbs down the current processes, so that anyone can understand them, but that I am creating professional tools that are able to compete with the processes that are standard today.

I determined exactly the structure of the special documents I envisioned for my solution and specified the tools that are necessary for the efficient handling of these structures.

For the process of editing, I used state of the art technology, to bring these documents to life in a way that allows multiple users to make edits simultaneously. The result is that these documents are smoothly synchronized and reflect the collective intentions of the individual editors.

1 Bevezetés

1.1 Motiváció, feladat bemutatása

A korábbi munkáim során rengeteg lehetőségem volt kipróbálni és megismerni a webfejlesztés különböző aspektusait. Az elmúlt években is aktívan fejlődnek a releváns technológiák, ez persze egyben áldás és átok is. Átok mert folyamatosan újabb trendek jelennek meg és ilyenkor a régi tudásunk elavulttá válik. Áldás viszont, mert a folyamatosan megjelenő újabb és újabb megoldások megkönnyíthetik a fejlesztők mindennapjait.

Az én figyelmemet is pontosan ez a gondolat fogta meg. Szeretnék egy olyan lehetőséget kínálni a modern alkalmazások fejlesztésére, ami bizonyos esetekben kényelmesebb és egyszerűbb, mint minden eddigi megoldás. Ezek mentén kezdtem el fejleszteni a keretrendszeremet.

Ennek a keretrendszernek alapvető eszméje, hogy az alkalmazásokat csapatok közösen fejlesztik. A fejlesztőknek többnyire közös a céljuk és a rendelkezésükre álló fejlesztői eszközök a szerszámok, amelyekkel a kívánt célt elérik.

A fejlesztői munka jellegéből adódóan nagyon sokrétű lehet, de mindenképpen találhatunk közös pontokat, amelyek szinte minden alkalmazás fejlesztésénél megjelennek. A dolgozatomban a felhasználói felületekre fókuszálok, mivel azt gondolom, hogy itt lehet a legnagyobb jelentősége a modern technológiák bevezetésének.

Ha felhasználói felületekről beszélgetünk, egyeztetünk, akkor sokkal könnyebb a dolgokat megmutatni, mint elmagyarázni. A vizuális elemeket nagyon nehéz szavakkal leírni, vagy a működés dinamikáját elmagyarázni. Ezért az a felvetésem, hogy a felhasználói felületek fejlesztése különösen érett az innovációra.

Pontosan arra gondolok, hogy szakítunk a hagyományos megoldással, miszerint rajzok alapján írunk kódot úgy, hogy az elkészült felület a lehető legjobban hasonlítson a rajzokhoz. Természetesen a rajzoknak is van létjogosultsága, viszont sok esetben duplán készül el a felület. Egyszer grafikusan, képszerkesztő programban, majd ismét úgy ahogy az alkalmazásban is szerepelni fog.

Ezt a két változatot jellemzően különböző emberek, különböző szaktudással készítik, és a fejlesztés további részeiben is karban kell tartani ezt a töréspontot. Sok

esetben további egyeztetés szükséges ahhoz, hogy a végeredmény tényleg olyan legyen, mint ahogy az eredetileg meg lett álmodva.

Az én elképzelésem szerint ez a két különváló dolog egybeolvad, azt gondolom, hogy nem szükséges kétszer elkészíteni ugyanazt a felületet hogyha rendelkezésünkre állnak a megfelelő eszközök. Pontosan egy olyan eszközre gondolok, ahol az alkalmazás felülete fejlesztői képzettség nélkül is elkészíthető. Ezt így nehéz bármihez is hasonlítani, de úgy kell elképzelni, mint egy szövegszerkesztő alkalmazás csak éppen a végső dokumentum egy alkalmazás felhasználói felülete.

1.2 A webes környezet adottságai

A böngészők hihetetlen fejlődésen mentek át a közelmúltban, ami részben annak a versenynek köszönhető, ami kialakult ezen a területen. A felhasználók könnyen válthatnak másik böngészőre, ebből ered a verseny, ami így hajtja előre a böngészők fejlődését.

A modern webes alkalmazások viszont már gyakran nem a böngésző által szolgáltatott programozói felületek közvetlen használatával, hanem modern webes keretrendszerek segítségével készülnek. Ez azt jelenti, hogy az alkalmazás továbbra is JavaScript nyelven készülhet, viszont mégis csak fordítás utáni kódot futtatunk a böngészőben. Ennek az egyik oka lehet az, a fordító képes olyan kódot előállítani, ami régebbi böngészőkkel is kompatibilis. Ezenkívül számos előnye lehet a webalkalmazások fordításának, például használhatunk TypeScript nyelvet, amely a JavaScript-ben nem létező típusinformációval egészíti ki a kódot, ami által precízebben definiálhatók a szoftverelemek.

A modern webes technológiák egyik alapkoncepciója, hogy az alkalmazásokat komponensekbe szervezzük. A komponensek alapvető tulajdonsága, hogy a belső komplexitást egy egyszerű interfész mögé rejtik. A megfelelően elkészített komponensek az explicit definiált függőségeik mentén hatékonyan újra felhasználhatók.

1.3 Modern webes technológiák

A böngészők mai formájukat több évtizedes fejlesztések során nyerték el. Ennek az evolúciónak még mindig nincs vége. A kezdetekben csak egyszerű szöveges

dokumentumok között lehetett linkekkel navigálni. Később a HTML 2.0[1] már képek megjelenítését is szabványosította. Majd megjelentek a stíluslapok, és a JavaScript által még programozhatók is lettek a weboldalak. Az újabbnál újabb kiegészítések mind szabványosítva lettek, és a felhasználókhoz is hamar eljutottak.

Ez az izgalmas fejlődés még a napjainkban sem állt le, a böngészők egyre többet tudnak, bár az alapvető funkciók már évtizedek óta elérhetők mindig újabb frontokat hódítanak meg a böngészők. A mai böngészők képesek rendkívül hatékonyan, alacsony késleltetéssel kommunikálni WebSocket[2] vagy WebRTC[3] segítségével még akár peer-to-peer[4] módon is. Egy másik jó példa webes grafika. A WebGL[5] nagyobb erőforrásigényű, akár 3D grafikák megjelenítését is lehetővé teszi azáltal, hogy az OpenGL ES[6] nyújtotta lehetőségeket böngészőben is elérhetővé teszi.

1.4 Webes keretrendszerek

A böngészők fejlesztői sokat tettek azért, hogy egyre kifinomultabb eszközökkel segítsék a fejlesztők munkáját. A fejlesztés során mégis gyakran előfordul, hogy újból és újból ugyanazt a problémát, vagy problémakört kell megoldani. Ilyen például az, hogy hogyan szervezzük komponensekbe az alkalmazásunkat, és hogy hogyan oldjuk meg köztük a kommunikációt. Az is ide tartozik, hogy hogyan kezeljük a stíluslapokat, és hogy kezeljük a különféle hálózati erőforrásokat hatékonyan.

Ezekre a kérdésekre nem csak egy jó válasz van, és a böngészők nem kínálnak rá kész megoldást. Ezután szinte szükségszerűen megjelentek keretrendszerek, amelyek egyszerűsítik a fejlesztői munkát, sőt bizonyos keretrendszerek arra is iránymutatást adnak, hogy hogyan érdemes szerveznünk a projektünk forráskódját. Ez igen hasznos mert, egyrészt nem kell minden egyes projekt során ezeket külön lefektetni. Ha két projekt

[1] Hypertext Markup Language specifikációjának második verziója

[2] A WebSocket egy kétirányú kommunikációs megoldás webalkalmazásokhoz

[3] Valósidejű kommunikációt tesz lehetővé webalkalmazásokban

[4] Központi kitüntetett csomópont nélküli kommunikációs megoldás

[5] https://developer.mozilla.org/en-US/docs/Web/API/WebGL_API

[6] Az OpenGL beágyazott eszközökre szánt változata

ugyanazt a keretrendszert használja, akkor a fejlesztők számára könnyebb az átállás, jóval rövidebb idő szükséges a projekttel való ismerkedésre.

A legelterjedtebb ilyen modern webes keretrendszerek ma az Angular[7], a React[8] és a Vue.js[9] [1]. Mindegyiknek megvannak a maga sajátosságai. Munkám során az Angular keretrendszert ismertem meg a legjobban, a dolgozatom témájához kifejezetten jó választás mert alapértelmezetten támogatja a TypeScript alapú fejlesztést [2]. Az általam célként kitűzött keretrendszer elkészítését is sokban segíti ez, hiszen minél bonyolultabb rendszert fejlesztünk annál inkább kifizetődő típusinformációval is ellátni a kódot. A kisebb hibákat még fordítási időben el tudjuk kapni, és nem kell hosszan keresgélni a hibákat az alkalmazásban.

Az Angular fontos adottsága még, hogy rengeteg különféle komponens elérhető hozzá, melyek jó része ingyenesen letölthető és szabadon felhasználható. Komoly előnyt jelent a fejlesztés során, hogyha az Angular közösség által már kifejlesztett komponenseket használhatjuk, hiszen ezek jól tesztelt megbízható komponensek.

Fontos még az is, hogy különféle stílusú komponenscsomagok közül válogathatunk. Ez azt jelenti, hogy sokkal könnyebben készíthetünk egységes kinézetű alkalmazásokat egy ilyen komponenscsomaggal mint ha magunk kezdünk el saját komponenseket tervezni és fejleszteni. Arról nem is beszélve, hogy például az általam is használt Material[10] design mögött a Google áll [3], így komponensek széles választéka ízléses kivitelben elérhető.

[7] https://angular.io/

[8] https://reactjs.org/

[9] https://vuejs.org/

[10] https://material.io/

2 Irodalomkutatás

2.1 Hagyományos felhasználói felületek összeállítása

A dolgozatom központi része a felhasználói felületek készítése. A felhasználói felületek alatt gondolok én minden olyan megjelenésére a programoknak, amikkel az emberek interakcióba léphetnek. Egy alapelv, hogy a felhasználói felületek mindig törekszenek a hatékony kommunikációra ember és gép között. A számítógépek fejlődésével természetesen fejlődött az is ahogyan a számítógépeket használjuk. Ma már a színes, ikonokkal és animációkkal gazdagított grafikus felületek használata a norma.

Ez nem volt mindig így, és nem árt, ha néha belegondolunk, hogy milyen messzire is jutottunk. Az írógépektől, a szobányi számítógépektől, az első grafikus megjelenítőkig is igen hosszú út vezetett és ez az út ma is épül, gondolhatunk itt a kiterjesztett valóságra és a holografikus kijelzőkre, melyek ma még csak gyerekcipőben járnak.

Az alkalmazásfejlesztés viszont komoly múltra tekint vissza, eleinte igen ritka és nehézkes volt a számítógépek használata, nyilván terjedésükkel a programok íróinak köre is szélesedett. Hajdanán igen nagy erőfeszítést igényelt egy-egy program elkészítése, és szintén nagy kihívás volt ezen programoknak terjesztése, felhasználókhoz juttatása. Hamar világossá vált, érdemes az ezzel kapcsolatos infrastruktúrákba fektetni, hiszen a számítógéphasználók köre robbanásszerű növekedést mutatott.

Pont úgy, ahogyan a számítógépek, a szoftverfejlesztés is csak a szakemberek egy szűk körének volt kiváltsága. Manapság már igen alacsony a belépési küszöb, egy hétköznapi ember is képes olyan alkalmazást vagy weboldalt készíteni, ami a világ bármely részéről könnyen elérhető. Ez forradalmasítja társadalmunkat, üzleti és személyes kapcsolatokat és közösségek épülnek fizikai kapcsolat nélkül a virtuális térben.

Az online jelenlét fontosabb, mint valaha. Mind személyes, mind üzleti szempontból a jelentősége tagadhatatlan. Az, hogy potenciálisan sokkal olcsóbban és sokkal szélesebb körben kelthetjük fel a jövőbeli ügyfeleink érdeklődést, igencsak az online platformok malmára hajtja a vizet.

Pontosan mit is kell tenni ahhoz, hogy a hétköznapi ember kapjon egy szeletet ebből a tortából? Tegyük fel, hogy egy kis családi vállalkozás szeretné fellendíteni az üzletet. Mit tehet értük a modern technika, vagy inkább az a kérdés, hogy mit tehetnek

magukért a vállalkozók. A leginkább célravezető, hogyha egy weboldal segítségével próbálnak szélesebb kört elérni. De mi is kell pontosan ahhoz, hogy valaki megjelenhessen a weben.

A dolgozatomban nem célom, hogy teljes képet adjak weboldalak üzemeltetésére, ezért azt nem részletezem, hogy hogyan lehet egy webes alkalmazást publikálni, és hogy milyen szolgáltatásokra kell ehhez előfizetni. A választott témám szempontjából az a fontos, hogy hogyan lehet elkészíteni azokat a felületeket, amik majd eljutnak a felhasználókhoz.

Ezeket a felületeket első körben mindenképpen a weboldal gazdájának kell megtervezni. Nyilván neki kell átgondolni, hogy mi ezzel az egésszel a célja, és hogy mégis hogyan tervezi azt elérni. Ez jelentheti azt, hogy pár mondatban megfogalmazza, de akár le is rajzolhatja, hogy nagyjából mit szeretne eredményként látni. Ezután célszerű egy kicsit konkrétabban a platform adottságait is figyelembe véve, webdesigner bevonásával megtervezni az oldalt. Itt fontos átgondolni, hogy pontosan hogyan szeretnénk tagolni, rendszerezni a mondandónkat. Ezután szokás elkezdeni pontos terveket készíteni, melyek során elkészülnek a grafikus, és szöveges tartalmak is. Ennek a folyamatnak a végén már teljesen összeáll a terv, így már többnyire látszik, hogy mi lesz az eredmény, csak az interaktív elemeket kell hozzáképzelni. Ezen tervek alapján tudják a fejlesztők elkészíteni a végső weboldal kódját. Ez a borzasztóan leegyszerűsített menete a klasszikus webes alkalmazások elkészítésének.

Ez a módszer adja a legjobb minőségű eredményt, de ennek meg kell fizetni az árát. A jó szakemberek munkája nem olcsó, és gyakran egy-egy feladatról csak menet közben derül ki, hogy pontosan mennyire nehéz megoldani.

Vannak viszont olcsóbb megoldások is. Nincs mindenkinek szüksége teljesen egyedi megoldásokra. Sok esetben egy sablonszerű megoldással több különféle weboldal is elkészíthető. Ebben az esetben ezeknek a weboldalaknak a struktúrája azonos lesz, viszont a tartalom tetszőleges. A gyakorlott szem könnyen kiszúrja, hogy egy sablonszerű weboldalról van szó, viszont sokkal könnyebben elkészíthető így egy weboldal, akár szakemberek bevonása nélkül is.

Ezeket a rendszereket gyakran tartalomkezelőnek is szokták hívni, mivel weboldalkészítő valójában csak tartalommal tölti fel a sablont. A legelterjedtebb ilyen

tartalomkezelő a WordPress[11] rendszer [4], melynek segítségével rövid betanulás után szinte bárki nekiláthat weboldalak készítésének. Sokféle sablon szabadon letölthető és felhasználható, ezenkívül kiegészítők is telepíthetők, melyek komplexebb működésekkel láthatják el a WordPress oldalkat. Persze van árnyoldala is ennek a megoldásnak. Alapvetően teljesítmény szempontjából egy kevésbé optimális megoldás ez, hiszen a sablonok úgy készülnek, hogy sokféleképpen felhasználhatók legyenek és nem célirányosan az adott weboldal igényeit veszik figyelembe.

2.2 Törésvonal a fejlesztésben

Szoftverfejlesztői munkásságom során nem tudtam nem észrevenni, hogy mennyire sokféle szakember munkája és tudása kerül a végső termékbe, mégis azt véltem felfedezni, hogy alapvetően két táborra oszlanak az emberek, mondhatjuk két különböző nyelvet beszélnek. Az egyik tábor, aki rajzokban nem interaktív illusztrációkban gondolkodik, a másik tábor pedig a szoftver nyelvét beszéli, és fordítja a grafikus terveket kóddá.

Ahogyan ez a gondolat kikristályosodott bennem, egyre nagyobb szükségét kezdtem látni, annak, hogy ez a határvonal elmosódjon. A meglátásom szerint mindkét félnek érdeke volna közös nyelvet beszélni, viszont ehhez még nem állnak rendelkezésre a megfelelő eszközök. Bár be kell látnom nem teljesen új a bennem megfogalmazódó gondolat, és vannak jó próbálkozások, de én úgy látom, hogy még egyetlen eszköznek sem sikerül. Viszont bemutatok egy jó és működő elképzelést, és megmutatom, hogy én hogyan fejlesztem tovább.

2.3 Figma

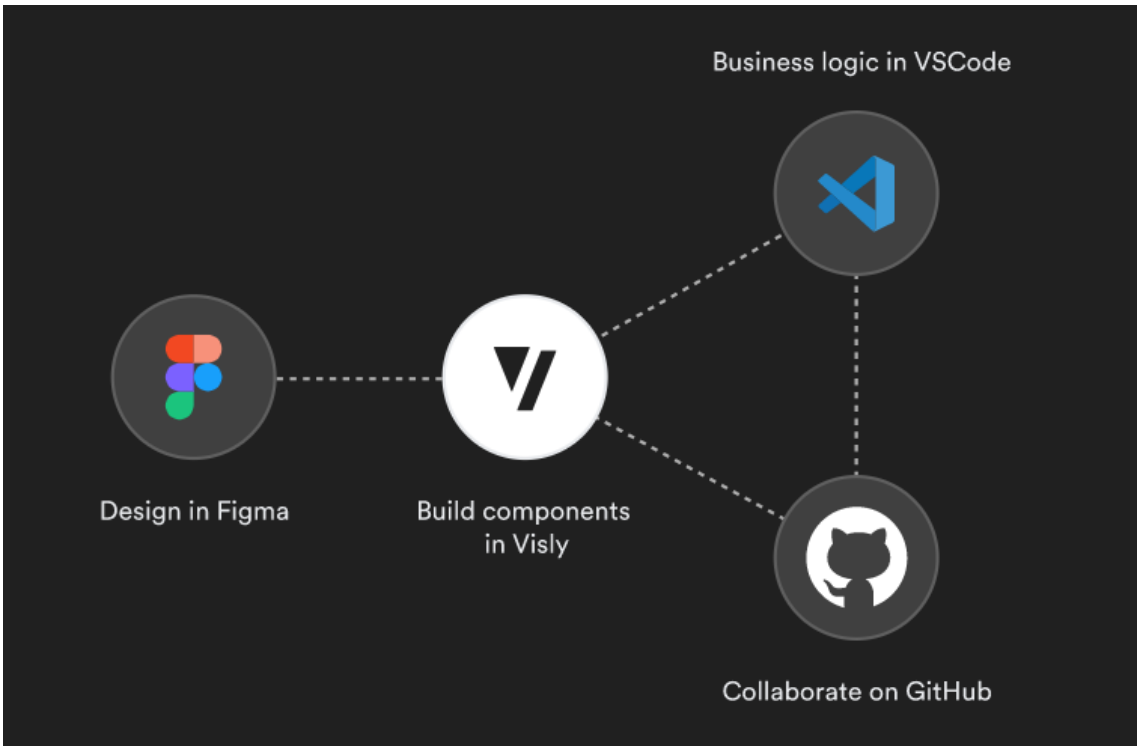
A Figma egy elsősorban web alapú prototípusok készítésére használható eszköz. Alapvetően nem raszteresen lehet szerkeszteni a felületeket, hanem komponenseket lehet definiálni. Ezek a komponensek tetszőlegesen igazíthatók az egyes felületi terveken belül. A felületi terv készítésének első lépése, hogy eszköztípust és képernyőméretet kell választani. Ezután saját, vagy importált komponensek segítségével elkészíthetők a kívánt felületek. A Figma képes kezelni egyszerű interakciót, elsősorban ez a képernyők közötti navigációra használható [5]. A Figma előnye, hogy az így készülő terveket sokkal

[11] https://wordpress.com, Tartalomkezelő és blog-rendszer webes tartalmak megjelenítéséhez

könnyebb karbantartani, mert egy komponens példányainak megjelenését a prototípus frissítésével elvégezhetjük, nem szükséges minden egyes előfordulását egyesével frissíteni. Természetesen az egyes példányokat lehet egyesével szerkeszteni, ilyenkor az adott példány kapcsolata megszakad a prototípus komponenssel.

2.4 Visly

A Visly[12] egy innovatív kiegészítő eszköz Figmához, ami segíti frontend komponensek készítését. A legfontosabb tulajdonsága, hogy valójában egyetlen korábbi eszközt sem vált ki, hanem a már ismert Figma tervek megvalósítását segíti, ennek eredményeként React komponensek készülnek [6], amiket fel lehet használni a végső alkalmazás fejlesztésekor. A Visly elsősorban kisebb felületi elemek, primitívek fejlesztésére lett kifejlesztve, és nagyon leegyszerűsíti a már meglévő komponensek testre szabását [7]. Az alábbi ábrával szemlélteti Visly a weboldalán, hogy rendszerük hogyan integrálódik a már ismerős fejlesztői eszközökkel.



1. ábra: Visly a fejleszői folyamatban

[12] https://visly.app/

2.5 Kollaboratív fejlesztés

A kollaboratív alkalmazások iránt napjainkban igen nagy a kereslet, különösen igaz ez azokra az online eszközökre, amelyek az online produktivitást, munkavégzést támogatják. Egy közismert példa erre a Google Docs[13] dokumentumszerkesztő, ami az igencsak népszerű Microsoft Word típusú dokumentumokhoz hasonló módon teszi lehetővé a szövegszerkesztést. A Google Docs viszont igen korán áthidalta a Word használatával járó nehézségeket, pontosan arra gondolok, hogy egy Word dokumentumot egyszerre egy ember szerkeszthetett. Hogyha egy terjedelmesebb írást többen készítettek, akkor jellemzően a különböző részeket különböző dokumentumokban kellett megírni, majd a munka végeztével kellett ezeken kézzel egyesíteni. Ez igen kellemetlen főleg, ha a végleges dokumentumon kisebb javításokat kellett eszközölni az azt jelentette, hogy a frissített dokumentumok az összes szerkesztőnek el kellett juttatni, hiszen, ha ők is további módosításokat szeretnének eszközölni, akkor fontos azokat a legfrissebb dokumentumon végezzék.

Összefoglalva a Microsoft Word sokáig nem kínált jó megoldást a dokumentumok verzióinak kezelésére, szerencsére mára már sokkal jobb a helyzet. A Google Docs akkoriban mégis meghatározó újítás volt, mert verziókezelés problémáját nem a szoftverfejlesztők számára ismerős elosztott verziókezeléssel oldotta meg, hanem életre keltette a dokumentumokat és közel valós időben megjelenítette [8], hogy ki éppen hol és hogyan szerkeszti a dokumentumot. A különböző felhasználók kurzorai egyszerre megjelentek, ezáltal láthatóvá vált, hogy ki éppen min dolgozik. Ez az újítás teljesen megváltoztatta azt, ahogyan gondolunk az online kollaborációra.

A ma sikeresnek mondható fejlesztői eszközökben is fellelhetőek ezek a kollaborációt segítő elemek. A már említett Figma is lehetővé teszi a közös szerkesztést, és hasonló módon a különböző felhasználók mutatója jelzi, hogy ki éppen min dolgozik.

Az Atlassian[14] fejlesztőknek szánt eszközeiben is fontos szerepet kap az ilyen típusú kollaboráció. Csak hogy egy konkrét példát is mondjak, a Confluence[15] nevű

[13] https://docs.google.com/

[14] Szoftverfejlesztői eszközöket gyártó ausztrál vállalat

[15] Wiki szerű dokumentumok közös szerkesztését teszi lehetővé

tudásbázis szerkesztő termék is támogatja a már bemutatott többkurzoros dokumentumszerkesztést.

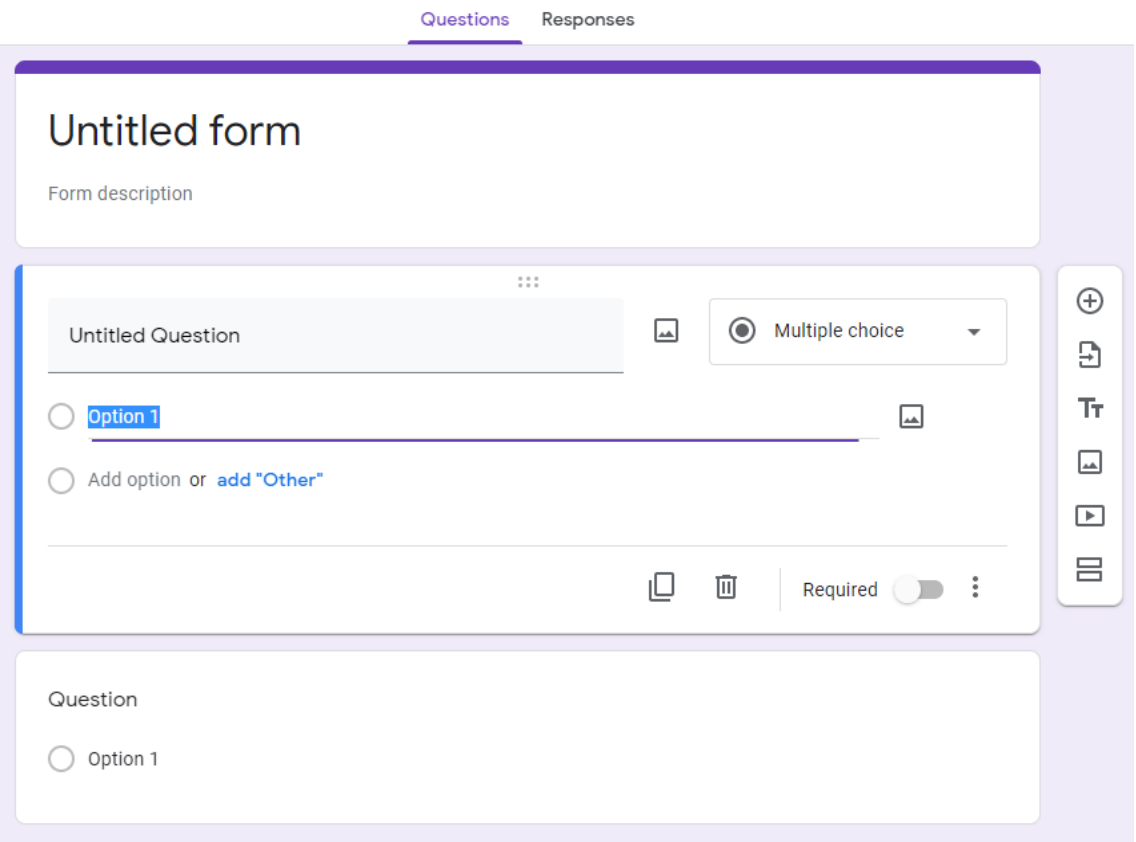
2.6 Felhasználó által definiálható felhasználói felület

Eddig arról írtam, hogy mik azok a technikai elemek, amik forradalmasították az online kollaborációt, most viszont arról is írok, hogy miben látom én a lehetőséget, és hogy hol látom én azt a rést, amit még ezek az eszközök nem fednek le teljesen. Azt gondolom jó okkal nincs még erre a problémakörre kiforrott megoldás, mert közel sem nyilvánvaló, hogy mi a jó megközelítés. Nekem az volt a legfontosabb kérdés, hogy hogyan lehet olyan eszközt készíteni, ami nem emeli meg a belépési küszöböt egyik fél számára sem. Végig szem előtt tartottam, hogy nem reális, hogy a designer programozzon, és az sem reális, hogy a fejlesztő kompromisszumra kényszerüljön a végső alkalmazással kapcsolatban.

Ebben a fejezetben az első kérdéssel foglalkozok, pontosan azzal, hogy hogyan tudja a designer jelentősen megkönnyíteni a fejlesztő munkáját, anélkül, hogy kódot írna. Ebben a tekintetben a Figma komoly előrelépés, hiszen a tervek nem csupán rajzok, hanem a kiválasztott komponensek jelennek meg, ami egy új absztrakciós réteg. És én is pont ebben az irányzatban látom a legnagyobb potenciált, hogy a designer inkább többnyire válogat a már meglévő felületi elemekből, és azokat olyan struktúrában helyezi el, ami fejlesztői szemszögből is kellően precíz, és a továbbiakban nem kell manuálisan átfordítani egy másik nyelvre, hanem egy az egyben felhasználható.

Egy érdekes és közismert ilyen megoldás a Google Forms[16], melynek segítségével egyszerű kérdőíveket lehet készíteni. A kérdőívek készítésekor az előre megadott kérdéstípusokból lehet választani. Igen sokféle kérdőívet össze lehet állítani a néhány rendelkezésre álló kérdéstípusból. Az egyek kérdéstípusok testreszabhatók, nyilván a kérdések szövegét a felhasználó adja meg, és a lehetséges válaszok is tetszés szerint definiálhatók. Ez az eszköz igazán jól használható kérdőívek kitöltésére, de hamar nyilvánvalóvá válik, hogy mennyire limitáltak a lehetőségek. A kérdőívek kizárólag lineáris kitöltést tesznek lehetővé. Arra van mód, hogy egy adott kérdésre adott válasz alapján jelenjen meg a következő szekció, de ezzel ki is merülnek a lehetőségeink.

[16] https://www.google.com/forms/about/



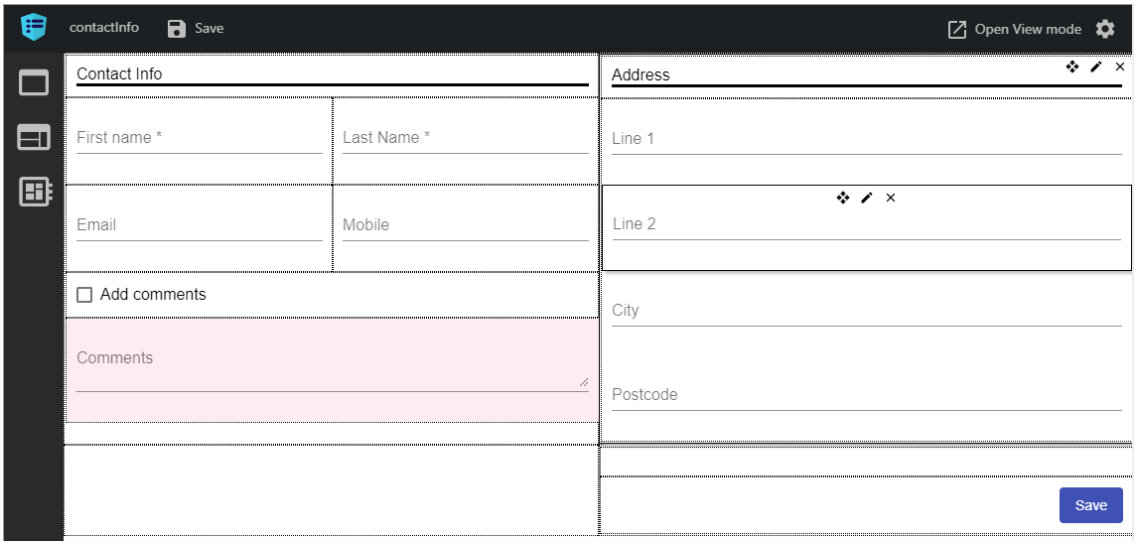
2. ábra: Google Forms szerkesztő

A fenti ábrán látható a Google Forms szerkesztői felülete, ami csak lineáris megjelenítést tartalmaz. A kérdéseket szekciókba lehet csoportosítani, és egyes kérdésekre adott válaszok alapján kihagyhatók későbbi szekciók.

Én a lehető legnagyobb fokú szabadságot szeretném biztosítani, ami csak elérhető. Egy számomra érdekes megoldás a FormQL[17] keretrendszer, aminek az a célja, hogy lehetővé tegye dinamikus űrlapok készítését Angular keretrendszerrel. A felületi elemeket egy felhasználóbarát szerkesztő segítségével helyezhetjük el a kívánt konfigurációban. Az elemeket fogd és vidd elven is mozgathatjuk a különböző felületi elemek között. Ebben a keretrendszerben már megjelenik az a koncepció, hogy vannak olyan felületi elemek, amelyek más elemeket tartalmaznak. Ezeknek a tartalmazó elemeknek a feladata a további komponensek rendezése, rendszerezése. A FormQL alapértelmezetten egymás alá helyezi a komponenseket, viszont a keretrendszer tartalmaz olyan komponenst, ami kettéosztja az adott oszlopot. A már kettéosztott hasáb tetszőlegesen tovább osztható, nyilván nem érdemes túlzásba vinni, mert a komponensek

[17] https://formql.io/

többsége meghatároz minimális szélességet. Ebben az esetben, ha ennél a minimális szélességnél kisebb helyre próbálunk beszúrni egy komponenst akkor a komponens az általa meghatározott minimális szélességet fogja felvenni. Ez jobb esetben azzal jár, hogy az adott oszlop aránytalanul széles lesz, rossz esetben viszont az egész felület elcsúszhat, és nem kívánt görgetősáv is megjelenhet.



3. ábra: formQL demo

A fenti ábrán látható a formQL oldalán elérhető demó, ami kinézetre sokban hasonlít az általam fejlesztett keretrendszerhez.

2.7 Releváns technológiák

2.7.1 HTML5

A HTML5[18] rengeteg újítást hozott be a webfejlesztés világába, a legfontosabb jelentősége, hogy egységesíti a böngészők programozói interfészét. Fontos újítás, hogy már skálázható vektoros grafikákat is meg tud jeleníteni, ezenkívül fontos még az audio és a video elemek szabványosítása. Kulcsfontosságú volt a web fejlődésében, hogy a különböző böngészők egységesen kezeljék a média tartalmakat is. A HTML5 eredetileg több mindent tartalmazott [9], viszont világossá vált, hogy a gyorsabb fejlődés érdekében, érdemes a különféle modulokat külön definiálni. Így külön specifikációval rendelkezik a canvas, WebSocket és WebRTC is. Azáltal, hogy ezek nem részei a HTML5

[18] HTML ötödik, egyben utolsó verziója

specifikációnak, a böngészők külön-külön, fokozatosan implementálhatták ezeket az egyre bonyolultabb alrendszereket, és azokat el is juttathatták a felhasználókhoz.

2.7.2 Virtual DOM

A DOM[19] rövidítés a HTML dokumentumok modelljére utal. Ez az alapja minden webes tartalomnak. A böngészők a letöltött HTML dokumentumokat értelmezik és az így előállt DOM-ot jelenítik meg. Ez a dokumentum modell dinamikusan szerkeszthető JavaScript segítségével, viszont eredetileg nem ez volt a rendeltetése. Ebből fakad, hogy a DOM szerkesztése egy relatíve költséges művelet, mivel minden változtatás alkalmával a böngészőnek a módosított elem környezetét is értesítenie kell a változtatásokról, ez adott esetben több szülő, testvér és gyerek elemet is jelenthet.

Pontosan ebből az okból jelent meg a virtuális DOM használata. A lényeg, hogy a virtuális modellt alkotó objektumok frissítése nem költséges, így a modern keretrendszerek bátran frissíthetik, ezt a virtuális dokumentum modellt. Az frissített virtuális modell alapján lehetséges, hogy egy lépésben egyszerre több elemi változtatás kerüljön át a valódi DOM-ba. Ez a megoldást főként a React keretrendszer terjesztette el, de azóta több keretrendszer átvette.

Felmerül a kérdés, hogy ez mennyire jó megoldás, tekintve, hogy a probléma forrása a böngészők belső működéséből fakad, és ez a trükk inkább csak tüneti kezelésnek mondható. A valódi megoldás az lenne, ha a böngészők letisztáznák a belső frissítési ciklusok menetét, és egységesen, hatékony megoldást kínálnának a DOM frissítésére.

Ezen a részen érdemes még megemlíteni az inkrementális DOM fogalmát is. Elsősorban a Google által fejlesztett megoldás, azzal a küldetéssel, hogy mobil eszközökön is a lehető legjobb teljesítményt nyújtsa. Ebből a szempontból fontos az alacsony memória használat és az is, hogy a lehető legkisebb legyen a webalkalmazás csomagmérete. Ezért az Inkrementális DOM úgy közelíti meg a problémát, hogy a nem használt felületi elemeket egy tisztítási fázis során el lehessen távolítani a végső alkalmazás csomagból. Ehhez fontos, hogy a komponensek kis összefüggő kóddarabokra tagolódjanak úgy, hogy a komponens az adatok változásával helyben tudja frissíteni a hozzá tartozó tényleges DOM elemeket.

[19] Document Object Model, a weboldalak objektum modellje

2.7.3 JavaScript

A JavaScript napjaink legmeghatározóbb programozási nyelve annak dacára, hogy a nyelv eredeti prototípusa mindössze 10 nap alatt készült el, a Netscape[20] böngészőhöz [10]. Azóta az ECMA[21] szabványosítja és használható mind kliensoldalon, főként böngészők programozására, és használható a kiszolgálói oldalon is például Node.js[22] formájában.

A JavaScript alapvetően egy gyengén típusos, prototípus alapú nyelv. A klasszikus objektumorientált programozásból ismeretes koncepciók, mint például az osztályok csak ES6[23]-ban kerültek a szabványba. Viszont fontos tudni, hogy ezek az osztályok is a prototípus rendszerre épülnek. Ezáltal nem adnak olyan szigorú megkötéseket, mint a klasszikus objektumorientált nyelvek, viszont az alapvető céljuk itt is az, hogy egy megadott minta alapján lehessen objektumokat létrehozni. A gyakorlott JavaScript használók az osztályokra inkább speciális függvényekként gondolnak, az öröklés kifejezés helyett pedig inkább a prototípus fa koncepciója a használatos.

A JavaScript sikerében közrejátszik az is, hogy a JSON (JavaScript Objektum Jelölés) formátuma szintén igen sikeres. Az XML[24] közvetlen alternatívájának mondható, és igen nagy előrelépésnek tekinthető, hiszen sokkal olvashatóbb emberek számára, ezenkívül könnyű szerkeszteni és formázni is. A JSON sikere szintén szabványokban szilárdult meg, ezután semmi sem gátolhatta a terjedését, mára minden releváns programozási nyelv képes JSON objektumok kezelésére.

2.7.4 TypeScript

A JavaScript persze bőven kap kritikát is, hiszen a gyenge típusosság igencsak szembe megy az alacsonyabb szintű, tipikusan elterjedt programozási nyelvekkel. Ennek van némi alapja, mert JavaScript használata során gyakran csak futásidőben derül ki, hogy hibás a kód. Ezek a hibák hagyományosan, például C programozási nyelvnél fordítás

[20] Az egyik legelső böngésző, ma már nem releváns

[21] https://en.wikipedia.org/wiki/Ecma_International

[22] Kiszolgáló oldali JavaScript futtatókörnyezet

[23] Az ECMAScript hatodik változata, ECMAScript 2015 néven is ismert

[24] Extensible Markup Language, egy általános jelölőnyelv

során kiderülnek. A statikus elemzés vitathatatlanul egy hasznos eszköz, ami a fejlesztő figyelmét már korán felhívja a hibás kódrészletre.

A TypeScript pontosan ezt a feladatot látja el, statikus típusinformációval látja el a JavaScript kódot, ezáltal megoldást nyújt arra, hogy a függvényeket és osztályokat explicit típusinformációval lássuk el. A TypeScript programozási nyelvet a Microsoft fejleszti és tartja karban, jórészt ennek köszönhető, hogy ilyen gyorsan ilyen jó nevet tudott magának szerezni ez az igen fiatalnak mondható nyelv. Fontos, hogy a TypeScipt kód futtatás előtt JavaScript-re fordul, ez a lépés viszont jellemzően a fejlesztési folyamat része. A felhasználókhoz már csak az optimalizált, futtatásra kész JavaScript kódot kell eljuttatni.

2.7.5 Komponens alapú fejlesztés

A komponens alapú fejlesztés lényege, hogy a szoftvert, az alkalmazást kisebb, egymástól jól elkülöníthető komponensekre bontjuk. Ez fontos rendező elve a szoftverfejlesztésnek, hiszen törekszünk arra, hogy a különböző részegységek gyengén csatoltak legyenek. A gyenge csatolás lehetővé teszi azt, hogy az egyes alkotóelemeket leválasszuk, egyesével lemérjük, leteszteljük, vagy akár adott esetben le is cseréljük. Ellenkező esetben a szoftver csak egy nagy egységként kezelhető, ez könnyen belátható, hogy nem ideális, mert előfordulhat, hogy valamilyen probléma merül fel az alkalmazással, és egy bizonyos részét újra kell tervezni majd implementálni. Ha az alkalmazás nincsen komponensekre bontva és fejlesztés során nem volt szempont az, hogy a különböző egységek egymástól elválaszthatóak legyenek, akkor a módosítás jóval költségesebb lesz, hiszen az átalakítandó részhez szorosan nem kapcsolódó egységek működését is meg kell érteni, és az átalakítás során jóval több szempontnak kell majd megfelelnie az újraírt komponensnek. Ennek az az eredménye, hogy egy látszólag kis módosítás is nagy munkával jár.

Az már világos, hogy miért érdemes figyelni arra, hogy a szoftver gyengén csatolt egységekből álljon. Ennek a jelentősége már egy projekten belül is megmutatkozik, viszont az újrafelhasználhatóság kérdése lehet, hogy csak több projekt után tűnik fel. A szoftverfejlesztés alapvető adottsága, hogy ugyanazzal, vagy közel ugyanazzal a problémával találjuk magunkat szemben. Ilyenkor válik nyilvánvalóvá, hogy az adott logikát érdemes könyvtárba szervezni, és megosztani a projektek között. Az

újrafelhasználhatóság pontosan arra utal, hogy az adott komponens a környezetétől függetlenül képes feladatát ellátni.

Az újrafelhasználhatóság kulcsa, hogy a komponens a függőségeit és szolgáltatásait explicit definiálja. Ebben az esetben a fejlesztő könnyen eldöntheti, hogy az adott komponens felhasználható-e az adott környezetben, és hogy rendelkezik-e az elvárt működéssel.

A komponensek természetesen kommunikálnak egymással, de a kommunikáció megvalósítása sokféle lehet. A leggyakoribb megoldások az üzenet alapú kommunikáció és a megosztott memórián alapuló kommunikáció. Az általam ismertetett webes környezetben az eseménykezelők, és referenciával átadott objektumok használata az elterjedt.

2.7.6 Single-page alkalmazások

A web alapvető adottsága volt, hogy amikor a szervertől új információt kértünk, az egész oldal újratöltött. Ez belátható, hogy nem hatékony, és vannak olyan esetek, amikor szeretnénk megtartani a kliensoldali állapotot úgy, hogy közben a szerver új információt szolgáltat. Eredetileg ez nem volt lehetséges, viszont 2006 környékén széles körben elérhetővé vált az XMLHttpRequest röviden XHR [11], melynek célja az, hogy JavaScript környezetből is lehessen hálózati kéréseket kezdeményezni.

Az XHR kulcsfontosságú volt abból a szempontból, hogy a webalkalmazások interaktívak legyenek. Ettől a pontól a web már nem csak linkekkel összekötött dokumentumok, hanem önjáró alkalmazások. A hagyományos weboldalakhoz képest nagyban javítja a felhasználói élményt, hogy nem kell a teljes oldalt újratölteni, hanem elég csak az adott részét lecserélni úgy, hogy friss adatokat mutasson.

Az alkalmazásoknak ezt az új típusát szokás Single-Page Application néven röviden SPA-ként emlegetni. Ezek az alkalmazások betöltés szempontjából úgy néznek ki, hogy az alkalmazás betöltésekor az egész alkalmazás letöltődik, ez jellemzően egy kisebb HTML fájl, ami a belépési pontként szolgál. Ebben a dokumentumban van meghatározva, hogy milyen további erőforrásokat kell az induláshoz letölteni. Ez jellemzően egy nagyméretű JavaScript fáj, ami a teljes alkalmazás forráskódját tartalmazza, becsomagolt formában, ezenfelül szükséges még a stíluslapok betöltése is.

A JavaScript csomag betöltése után indul az alkalmazás, ami a klasszikus szerveroldalon összeállított weboldalakkal összehasonlítva lassúnak mondható. Ebben a kontextusban a betöltés elindításától az első értelmezhető megjelenésig eltelt időtartamot szokás vizsgálni. Az SPA előnyei csak a huzamosabb használat során jelentkeznek, hiszen a betöltés után már csak az alkalmazás által használt adatokat kell hálózatról betölteni, az oldalak struktúráját már a böngésző állítja össze.

2.7.7 Angular, React, Vue

Eddig többnyire általánosan mutattam be, hogy mik voltak az előfeltételei a modern webes alkalmazásoknak, és hogy hogyan jutottunk el a modern SPA megoldásokhoz, de ezek a fejlesztések nagyon szorosan kapcsolódnak a keretrendszerekhez, melyek ilyen népszerűvé tették. Éppen ezért szeretném azt is bemutatni, hogy az apró vívmányok, hogy formálódtak keretrendszerekké.

A nagymennyiségű DOM manipulációnak a költségességét a Facebook által fejlesztett React keretrendszer próbálta megoldani a virtuális DOM alkalmazásával [12]. Ezenkívül a JavaScript-be ágyazott XML, röviden JSX[25] kifejlesztésével hozta közelebb a felületek deklaratív definícióját a JavaScript programozókhoz.

Az Angular keretrendszer egy TypeScript alapú, szintén SPA keretrendszer. A legnagyobb hangsúlyt a karbantartható alkalmazás fejlesztésre helyezi. Az Angular erős ajánlásokat nyújt arra, hogy hogyan szervezzük az alkalmazásaink fájljait. Ezenkívül fontos az is, hogy a visszafele kompatibilitás kérdését igen komolyan veszi, a keretrendszer automatizált megoldásokat kínál arra, hogy a keretrendszer korábbi verzióival készült projekteket a legújabb verziójára frissíthessük.

A Vue.js keretrendszer a legfiatalabb a három közül, és a leginkább dinamikusan fejlődő. Előnyét részben annak köszönheti, hogy a korábbi próbálkozások hibáiból tanult, és mint új keretrendszer, kevésbé kell a kompatibilitási kérdésekkel foglalkoznia. Érdekessége, hogy eredetileg az Angular egy könnyűsúlyú alternatívájaként indult, és csak alapvető funkciókat szolgáltatott. Mára komplett keretrendszerré szélesedett ki, és támogatást ad a már ismertetett JSX szerű dokumentumok használatára is [13].

[25] https://en.wikipedia.org/wiki/React_(web_framework)#JSX

2.7.8 Websocket

A már ismertetett feladat megoldásához valós idejű kommunikáció megvalósítása is szükséges. A hagyományos HTTP[26] szabványnak még nem volt célja a valós idejű kommunikáció támogatása. A kapcsolatok lezárásra kerülnek miután a böngésző kérésére a szerver a válaszát elküldte. Ez nem hatékony, mert új kapcsolat kialakítása több erőforrást igényel és extra késleltetést okoz.

Az eredeti HTTP szabvány a 90-es évek közepén lett szabványosítva, viszont a WebScoket-et szabványosító RFC[27] csak 2011-ben jelent meg [14]. A WebSocket megjelenése komoly lendületet adott a közel valós idejű, full-duplex[28] kommunikációt igénylő alkalmazásoknak. Alkalmazások egy teljesen új családjának nyílt meg a webes platform, mint célpont.

A WebSocket alapjában véve továbbra is a HTTP protokollra támaszkodik. A kapcsolat kiépítése egy speciális HTTP kéréssel kezdődik, amely egy Upgrade fejléccel kérvényezi a full-duplex kapcsolatot. Amennyiben a kiszolgáló ezt a módot támogatja, akkor a kapcsolat kiépül. Az újonnan kiépült csatornán folyhat szöveges kommunikáció, vagy akár nyers bináris adatfolyam is.

Az efféle Websocket kapcsolat azáltal tudja az adatátvitelt hatékonyabban lebonyolítani, hogy a HTTP kapcsolatok csak igen speciális módon engedik használni a háttérben lévő TCP[29] kapcsolatot. A WebScoket viszont a TCP-kapcsolatot közvetlenebbül engedi használni, így oda vissza üzenhet a szerver és a kliens. Természetesen a TCP kapcsolatból adódóan az üzenetek kézbesítéséről nyugtát kap a feladó, így ezen és a csomagok megérkezésének helyes sorrendjén sem kell aggódnia a webfejlesztőknek, viszont ez csak arra ad garanciát, hogy a feldolgozás a beérkezés sorrendjében indul, a feldolgozás végeztével a kiszolgáló dolga hogy megfelelő sorrendben küldje vissza a válaszokat [15].

---

[26] Hypertext Transfer Protocol, a web alapját képező kommunikációs protokoll

[27] https://en.wikipedia.org/wiki/Request_for_Comments

[28] A két fél egyszerre kommunikálhat, nem kell megvárni, hogy a másik fél befejezze az adást

[29] Transmission Control Protocol, az OSI modell szállítási réteg beli protokollja

2.8 Valósidejű kollaboráció

2.8.1 Konzisztencia

A megosztott hozzáférésből fakad az adatok konzisztenciájának kérdése. A megosztott adathalmaznak mindig olyan képet kell mutatnia, ami az alkalmazás szempontjából értelmezhető, és ráadásul a felhasználói interakciók mentén a felhasználói beavatkozások egy értelmes sorozatát reprezentálja.

Ezek a problémák és a lehetséges megoldásuk szemléltethetők az adatbázis kezelő szoftverekre jellemző tranzakciós modellel. Az adatbázis szoftver fogadja a beérkező kéréseket, és ezeket atomi műveletként kezeli. Az atomiság azt jelenti, hogy egy lépésként kezelendő, vagy sikeresen végrehajtódik minden lépése, vagy egyetlen lépése sem jut érvényre, hogyha valamely része sikertelen.

A konzisztencia kérdése felmerül az általam fejlesztett keretrendszernél is. Hogyan lehet kezelni a nagyméretű adatmodellt, miközben annak különböző részeit más-más felhasználók egyszerre szerkesztik.

Erre egy igen modern megközelítés a konfliktusmentes[30] replikált[31] adattípusok használata. Ennek a megközelítésnek a lényege, hogy a replikák, az adatmodell másolatai egymástól függetlenül, párhuzamosan, akár központi koordináció nélkül is frissíthetők.

2.8.2 Késleltetés

A késleltetés is fontos kérdés egy ilyen alkalmazás fejlesztésekor. Az elképzeléseim szerint a felhasználók egyidőben, közösen szerkesztik a dokumentumokat. Amennyiben valaki a dokumentum egy régebbi verzióján végez változtatásokat, azok a változtatások, amiket az elavult dokumentumot módosítják, lehet, hogy értelmezhetetlenek vagy ütköznek a dokumentum friss változatával. Ezeket a vitás helyzeteket a központi kiszolgáló tudja csak feloldani.

---

[30] Az adatbázisokon végzett tranzakciók közti összeférhetetlenséget szokás konfliktusnak nevezni. A konfliktusmentes adattípusok lényege, hogy a tranzakciók sosem lesznek összeférhetetlenek.

[31] A replikálás az adat másolásának folyamata. Eredményeként az adat több különböző helyen tárolódik, ebben az esetben a replikáció távoli számítógépek között történik.

Amennyiben a felhasználó utolsó változtatását a kiszolgáló elfogadta, azt nyugtával jelzi. Ha ezt a nyugtát a felhasználó megkapta, biztos lehet benne, hogy a munkája érvényre jutott, és mások már csak az ő változtatásaira építve tudnak további változtatásokat menteni.

Tegyük fel, hogy egyszerre ketten szerkesztik a dokumentum egyazon részét. A gyorsabb felhasználó változtatását a kiszolgáló elfogadja, és nyugtázza is, viszont a második felhasználó változtatását a kiszolgáló elutasítja. Ezt a helyzetet többféleképpen fel lehet oldani. A legegyszerűbb megoldás az, hogy a második felhasználó értesítést kap arról, hogy a változtatása óta új verziója keletkezett a dokumentumnak, és a friss módosításokat ezen a dokumentumon kell elvégeznie. Ezzel az a gond, hogy a második felhasználó munkája elveszett, és újra meg kell csinálnia. Ennek az egyszerű megoldásnak ez az átka, viszont a gyakorlatban mégis működhet.

Ha a kiszolgáló és a felhasználók közötti kapcsolat kellően gyors, és az adatok másodpercenként többször szinkronizálhatók, akkor ez az elveszett változtatás nem okoz túl nagy fejfájást, hiszen a felhasználó egyből látja, hogy hogyan módosult a felület és tetszése szerint korrigálhat.

3 Specifikáció

3.1 Alapvető követelmények

Az elkészült keretrendszer többféle nézetet támogat. A keretrendszer a keretrendszert használó fejlesztő által konfigurálható, hogy éppen milyen módban szeretné megjeleníteni az adatmodellt. Az adatmodell emberi szemmel is értelmezhető, JSON dokumentum formájában kezelhető. A dokumentumot alkotó konkrét kulcs-érték párok sémájára a keretrendszer nem ad szigorú feltételeket. A keretrendszerben megvalósított komponensek az adatmodellt saját működésükhöz igazíthatják.

3.1.1 Megtekintő nézet

A megtekintő nézet lényegében a végső felhasználói nézet. Ekkor van a legkevesebb lehetőség a szerkesztésre, módosításra. A komponensek teljes mértékben maguk határozzák meg, hogy hogyan viselkednek a megtekintő nézetben. A szerkesztés során használatos kisegítő funkciók egyike sem elérhető. A megjelenő komponensek felhasználhatják azt az információt, hogy éppen megtekintő nézetben jelennek meg.

A megtekintő nézet jellemzően úgy néz ki, hogy a felhasználó beviteli mezőket lát, valamilyen elrendezésben. Ezeket a beviteli mezőket tetszése szerint kitöltheti tetszőleges értékekkel, viszont az oldal struktúráját nem tudja megváltoztatni. Kivételnek tűnhetnek azok a komponensek, amelyek például egy lista megadására szolgálnak. Valójában ebben az értelemben ez az oldal struktúráját nem változtatja meg. A lista megjelenítő akkor is lista megjelenítő marad, ha már eggyel több elemet tartalmaz.

A megtekintő nézetben a keretrendszer a beírt értékeket kezeli, és a fejlesztő számára elérhetővé teszi. A beírt értékek kiolvashatók, az értékek változásáról értesülni tudnak másik alkalmazáskomponensek.

3.1.2 Szerkesztő nézet

A szerkesztő nézet mindent tud, amit a megtekintő nézet, viszont vannak további extra lehetőségek. Ebben a módban a felhasználó módosíthatja a felület struktúráját. Erre különböző eszközök állnak rendelkezésre. Alapvetően a dokumentumban kattintásra egyesével kijelölhetők a komponensek. A szerkesztő eszköztár ilyenkor a kiválasztott komponens tulajdonságait részletesen mutatja. A komponenshez tartozó kulcs-érték párok szerkeszthetők is.

Komponensekből épülő fa, ami a dokumentumot adja, bejárható a szerkesztő nézetben. Egy kiválasztott komponensnek mindig meghatározható pontosan egy szülője, kivéve, ha a gyökér komponens van kiválasztva. A szerkesztő nézet könnyen lehetővé teszi, hogy a kijelölést egy komponensről a komponens szülőjére helyezzük.

A szerkesztő nézetben a komponensfa szerkesztését segíti, a másolás és beillesztés funkció. A kimásolt összetett adatmodell pont úgy illeszthető be, mintha egy egyszerű komponens lenne.

A dokumentumban szerkesztés során a komponensek lecserélhetők más komponensekre. Azok a komponensek, amelyek további komponensek tartalmazására képesek, azok alkalmazhatják az üres komponens megvalósítását. Az üres komponenst a keretrendszer szolgáltatja, és bármikor lecserélhető egyéb komponensre. Az üres komponensnek csak a szerkesztő módban van jelentősége, és azt jelzi a felhasználónak, hogy az adott helyre tetszés szerint komponenst helyezhet.

A szerkesztés során a felhasználó egy katalógusból, listából választhat komponenseket. Ebben a listában jelennek meg az éppen elérhető komponensek. Ennek a listának a bővítése úgy lehetséges, hogy a felhasználó beszerzi a kívánt komponenseket, mondjuk az internetről szabadon elérhető forrásból, vagy ha a kívánt komponensek még nem állnak rendelkezésre, akkor fejlesztői munkával előállíthatók.

3.2 Kiegészíthetőség

A keretrendszer használata szempontjából nagyon fontos kérdés, hogy hogyan lehet egyedi, speciális megoldásokkal kiegészíteni a már meglévő megoldásokat. A keretrendszer készítésekor jelentős figyelmet fordítottam a kiegészíthetőségre, mert célom túlmutat azon, hogy az általam fejlesztett komponenseket mások felhasználhassák.

A keretrendszer kialakításakor fontosabb, hogy mások is könnyen produktívvá válhassanak, és a saját szükségleteik mentén saját komponenseket fejleszthessenek.

Ez az szoftver szempontjából azt jelenti, hogy a lehető legvilágosabban meg kell határozni, hogy mi szükséges egy új komponens fejlesztéséhez. Ebben az esetben ez azt jelenti, hogy az új komponens fejlesztője könnyedén, akár segítő példák mentén el tud kezdeni új komponenseket készíteni. Már említettem, hogy a keretrendszernek szervesen része az üres komponens, viszont egyéb alapvető komponenseket is célszerű a keretrendszerrel együtt terjeszteni azért, hogy egy újonnan elkészülő komponens legalább alap szinten kipróbálható legyen. Így szükséges, hogy a keretrendszer rendelkezzen olyan komponenssel, ami képes más komponenst, komponenseket tartalmazni, például egy lista komponens. Ez egyszerű példák bemutatásához is szükséges, ezért elengedhetetlen része a keretrendszernek.

Ezenkívül a keretrendszer szükségszerű eleme még valamilyen beviteli mező, enélkül nincsen mód arra, hogy új értékek kerüljenek a dokumentumba. Ennek okára később részletesen is kitérek.

A kiegészíthetőség szempontjából fontos, hogy a keretrendszer ne csak a minimálisan szükséges elemeket implementálja. Célszerű ebből az okból olyan példákat is készíteni, amelyek a komponensek közötti komplexebb interakciókat is bemutatják.

3.3 Újrafelhasználhatóság

Az újrafelhasználhatóságról már írtam általánosságban, mint szoftverfejlesztési irányelv. Ebben az alfejezetben viszont az általam készített keretrendszerben írt komponensek és dokumentumok újrafelhasználhatóságát és hordozhatóságát fogalmazom meg.

Azt már meghatároztam, hogy a komponenseknek meg kell felelni a keretrendszer által előírt formai követelményeknek. Az ilyen módon írt komponensek a keretrendszer által hordozhatók lesznek, tehát belső változtatás nélkül felhasználhatók leszek több különböző dokumentum készítéséhez is.

Az újrafelhasználhatóság az általam írt keretrendszerben készített dokumentumokra is érvényes, nem csak a különálló komponensekre. Különálló, egymástól teljesen független dokumentumok egyesíthetők egy dokumentummá. Egy tetszőlegesen komplex dokumentum rendelkezik ugyanazokkal a tulajdonságokkal,

amelyekkel egy egyszerű komponens rendelkezik. Az így összeállított dokumentumok részelemei tetszőlegesen másolhatók, duplikálhatók, de akár külön dokumentumba is leválaszthatók.

3.4 Adatstruktúra

A megjelenített felület mögött álló adatstruktúra tárolja a felület struktúráját és tárol minden olyan adatot a felületről, amit paraméteresen meg lehet változtatni. Az adatstruktúra típusát illetően egy JSON adatstruktúra, ami a JavaScript natív adatstruktúrája.

Ehhez az adatstruktúrához, pontosabban annak részfáihoz vannak hozzákötve a felületen megjelenő komponensek. Ezek a komponensek az adatstruktúra alapján alakíthatják a megjelenésüket és a belső struktúrájukat. A komponensek feladata, hogy a tartalmazott alkomponensek felé továbbadják az adatstruktúrának a megfelelő részeit.

Az adatok referencia-ként vannak kezelve, így minden komponense a rá tartozó adatrészeket közvetlenül módosíthatja, és ezenfelül a tartalmazott komponensek adatstruktúrájához is hozzáfér

JSON leíróját minden komponens magának deklarálja, de meghatároztam néhány konvenciót, amit érdemes betartani, hogy az általam készített alapvető komponensekkel kompatibilis legyen az újonnan készített komponens. Ilyen konvenció például, hogy a tartalmazott komponensek az *elements* kulcs alatt vannak felsorolva egy tömbben, még akkor is, hogyha csak egy tartalmazott elem lehet, akkor is ajánlott azt tömbben tárolni.

Alapvető elve az általam fejlesztett keretrendszernek, hogy nem ad szigorú megkötéseket arra, hogy hogyan kell a komponenseket definiálni. Viszont, ha az általam lefektetett alapokkal valaki szakítani szeretne, akkor az általam definiált alapvető építőelemeket újból definiálnia kell majd úgy, hogy a saját elképzeléseivel egységesek legyenek azok.

3.5 Fejlesztői felület

Egy ilyen keretrendszer sikere többnyire a fejlesztők hozzáálláson áll vagy bukik, a felhasználókat nem igazán érdekli, hogy mi van a háttérben. Ezért a fejlesztői oldal kidolgozása és dokumentálása prioritást élvez.

Eddig sokat beszéltem már a szerkesztő eszközről, amivel a hétköznapi ember is képes felhasználói felületet összeállítani, viszont nem fejtettem ki ennek a másik oldalát. Pontosan arra gondolok, hogy hogyan lehet ezeket a dokumentumokat egy komolyabb alkalmazásban is megjeleníteni.

Erre több megoldást is kínál a keretrendszerem, a legegyszerűbb például, hogy a szerkesztővel elkészített dokumentum adatmodelljét valamilyen adatbázisba elmentjük, majd később igény szerint megjelenítjük. Ebben az esetben a kiszolgáló nem értelmezi, és egyáltalán nem módosítja a dokumentum adatmodelljét. Ennél egy bonyolultabb, de rugalmasabb megközelítés, hogy a dokumentumot a kiszolgáló értelmezi és tetszés szerint manipulálja. Amennyiben a végső adatmodell érvényes, és a felhasznált komponensek számára is maradéktalanul értelmezhető, akkor az gond nélkül megjeleníthető, de akár szerkeszthető is.

A keretrendszer képes lehet az adatmodellt elkülöníteni strukturális és értek modellekre. A keretrendszerrel együtt fejlesztett komponensek rendelkeznek ilyen viselkedéssel, mert ez lehetővé teszi, hogy további értelmezés nélkül, hálózati kiszolgáló bevonása nélkül kinyerjük és tetszőleges célra felhasználjuk a már említett megtekintő nézetben bevitt értékeket.

A most említett érték modellje egy dokumentumnak praktikusan követi a komponensek hierarchiáját. Ha egy komponens szekvenciálisan tartalmaz további komponenseket, akkor az adott komponens érték modellje is szekvenciálisan fogja tartalmazni az alárendelt komponensek érték modelljeit.

3.6 Kommunikáció a kiszolgálóval

A keretrendszer fontos tulajdonsága, hogy támogatja a valós idejű kollaborációt. Ez elsősorban arra szolgál, hogy dokumentumokon egyszerre többen is dolgozhassanak, viszont erre nincs minden esetben szükség. Egy fontos célterülete az általam készített megoldásnak a kérdőívszerű adatbekérés. Ebben az esetben a már említett módok egyikével elkészül a kérdőív és a kitöltés eredményét visszaküldi a felhasználó. Ebben a példában nem kap szerepet a kollaborációs eszközkészlet, viszont szerepet kap egy kiszolgáló, ami fogadja a kitöltés eredményét. Arra a kérdésre, hogy a kitöltés eredménye hogyan jut el a kiszolgálóhoz, az lehet egy komponens feladata, de akár a keretrendszeren kívül is megoldható.

A valósidejű kollaborációt megvalósító részegység szükségszerűen kommunikál a hálózaton, viszont ehhez nem kell feltétlenül központi kiszolgáló egység. Az általam alkalmazott kollaborációs megoldás lehetővé teszi azt is, hogy a felhasználók egymással közvetlenül kommunikáljanak peer-to-peer módon. Teljesítmény szempontjából viszont jellemzően a központi kiszolgáló alkalmazása előnyösebb, mert központi kiszolgáló felé az átlagos késleltetés jellemzően kisebb, és az egyes résztvevők hálózaton forgalmazott adatmennyisége is jóval alacsonyabb.

4 Megvalósítás

4.1 Angular dinamikus komponens példányosítás

Az Angular alkalmazásokat, mint már említettem komponensekbe szokás szervezni. Arról viszont még nem írtam, hogy pontosan miből áll egy komponens definíciója.

Új komponens készítésének a legegyszerűbb módja az, hogy az Anugular parancssoros eszközével generáltatjuk. Ekkor a kiválasztott könyvtárba a megadott néven legenerálásra kerül három fájl. Egy *.ts* kiterjesztésű, egy *.html* és egy stílusfájl is.

A komponens definíciója a *.ts* fájlban van és innen hivatkozik a HTML és a stílusfájlra. Az így létrehozott komponenst a definícióban szereplő selector-ral lehet másik HTML fájlban hivatkozni. Az alkalmazás a gyökérkomponensében ilyen módon hivatkozhat egyéb komponensekre, amelyek további alkomponensekre hivatkozhatnak.

Az alkalmazásom megvalósításához ennél egy kicsit rugalmasabb megoldásra van szükségem. Mivel az alkalmazás fordításakor még nem tudom pontosan, hogy milyen struktúrában hogyan lesznek egymásba ágyazva a komponensek.

Esetemben ez csak futás közben dől el, és teljes mértékben a felhasználók döntésein múlik.

Tehát szükséges egy olyan megoldás, amire az alkalmazás gyökérkomponense komponensként hivatkozhat, viszont ez a komponens már futási időben, rugalmasan tudja létrehozni a saját gyerekkomponenseit.

Az én megvalósításomban rekurzívan jönnek létre a komponensek, tehát nem egy komponens feladata, hogy teljes mélységében felépítse ezt a komponensfát, hanem minden lépésnél egy ilyen dinamikus típusú komponens kerül példányosításra, ami saját maga eldönti az adatstruktúra alapján, hogy milyen megjelenítési komponensként fog végeredményben a felületre kerülni.

4.2 Angular adatkötés dinamikus komponenseknél

Az Angular keretrendszer a komponensek közti kommunikációt adatkötés útján bonyolítja le. Alapvetően háromféle adatkötési módot lehet választani [16].

Az első, a modell kötése nézetbe, melynek során a modell egy változóját követi a nézet állapota. Az Angular keretrendszer ebben az esetben automatikusan figyeli a modell változásait, és csak akkor frissíti a nézetet, hogyha a modell ténylegesen változott.

A második eset, hogy a nézet frissíti a modellt. Ez tipikusan események kezelésére szolgál, például egy gomb lenyomása generál egy eseményt, melyet a modell tetszés szerint feldolgoz. Belátható, hogy a gomb állapota nem függ a modelltől, ez a legfontosabb különbség ez előző esethez képest.

A harmadik eset a kétirányú kötés, mely során a modell változása frissíti a nézetet és a nézeten végzett változtatás is frissíti a modellt. Erre tipikus példa egy szöveges beviteli mező, aminek értéke változtatható a modellből is, és úgy is, hogy a felületen beleírunk valamit.

Tipikus webes alkalmazások fejlesztésére ez a három féle adatkötés remekül működik, amikor a klasszikus HTML sablonnal definiáljuk a komponenseket, viszont esetemben ez nem működik. Nekem arra van szükségem, hogy a felhasználó által menet közben megadott adatmodell alapján jelenjenek meg a komponensek.

4.3 Minimális adatstruktúra kialakítása

Minden komponens teljes hozzáféréssel rendelkezik saját, és az általa tartalmazott komponensek adatmodelljéhez. Ez közvetlen következménye annak, ahogyan a komponensfa felépítésre kerül. Mindig a szülő feladata, hogy a gyerekeit megjelenítse, és hogy azokat az adatmodell megfelelő részével lássa el, mint bemeneti paraméter. A kétirányú adatkötések teszik lehetővé, hogy a szülő komponensek mindig a gyerekeik friss adatmodelljét érjék el.

A keretrendszer fejlesztése során úgy döntöttem, hogy először a megjelenítést valósítom meg. Ezt úgy közelítettem meg, hogy kézzel létrehoztam pár egyszerű példát. Ezeket nevezhetjük akár tesztesetnek is. A megjelenítő fejlesztése során rendszeresen kipróbáltam, hogy hogyan reagál a megjelenítő a különféle bemenetekre.

4.4 Alapvető komponensek kialakítása

A keretrendszerrel szemben alapvető követelmény, hogy definiáljon néhány létfontosságú komponenst. A keretrendszer szempontjából ez elengedhetetlen, mert a

komponensek paraméterezésére szolgáló felületeket ezekből a komponensekből lehet összeállítani. A keretrendszer alapvető tulajdonsága, hogy a komponensek maguk határozzák meg azt a felületleírót, amivel szerkeszteni lehet a paramétereiket. Ennek fényében tudatosan haladtam sorban az egyes komponensek fejlesztésével úgy, hogy az elkészült egységeket mindig ki tudjam próbálni.

4.4.1 Szövegdoboz

A szöveges beviteli mező valószínűleg a legfontosabb egyszerű komponens, mivel ez a legegyszerűbb megoldás arra, hogy tetszőleges értékek kerülhessenek az adatmodellbe. A kezdetben a keretrendszer teszteléséhez olyan szövegdobozt készítettem, ami az értékül kapott szöveget JSON-ként értelmezte. Így az adott kulcshoz nem csak szöveges értéket lehetett felvinni, hanem tetszőlegesen komplex, beágyazott adatstruktúrát is. Ezzel menet közben is tudtam tesztelni olyan komponenseket, amelyek az adatmodell még csak olvasni tudták, írni még nem. A szövegdoboz komponens példányosításkor bemenetként várja a központi adatmodellt *data* néven, ezenkívül két kimenetet definiáltam. Az egyik az érték adatmodell változását jelző esemény, a másik kimenet pedig azt jelzi, hogy éppen rákattintottak a komponensre.

```
static manipulator(data, value) {
  data.text = value[0];
  data.value = value[1];
}

static editorForm(data) {
  return {
      'type': ElementTypes.ListContainer,
      'elements': [
          {
            'type': ElementTypes.Input,
            'text': 'Text',
            'value': data.text
          },
          {
            'type': ElementTypes.Input,
            'text': 'Value',
            'value': data.value
          }
      ]
  };
}
```

A fenti kódrészlet jól ábrázolja a keretrendszer egyik alapvető koncepcióját, miszerint a komponensek maguk definiálják, hogy hogyan szerkeszthetőek. A

kódrészleten az látható, hogy az *editorForm* függvény létrehoz egy olyan adatmodellt, ami egy *ListContainer* komponensbe foglal két *Input* típusú elemet.

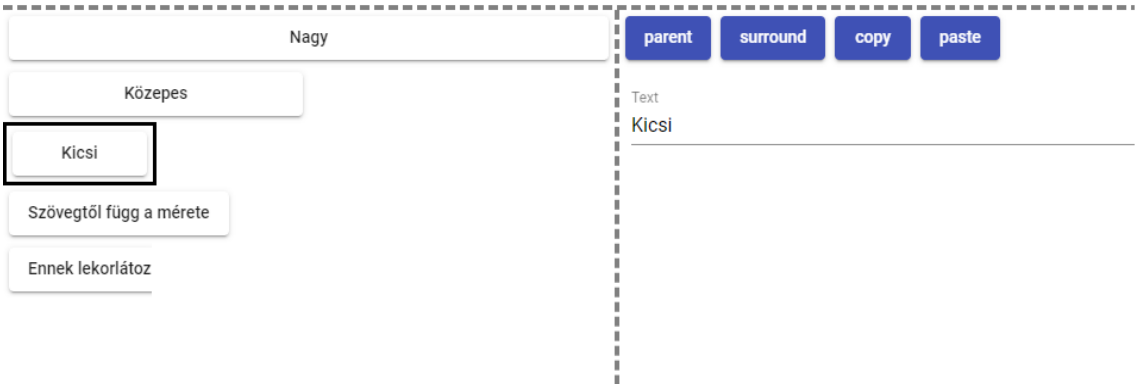
A szerkesztő eszköz feladata lesz majd, hogy ezt az adatmodell megjelenítse. Ha ezt az adatmodellt a felhasználó szerkeszti, annak az eredménye jut a manipulátor függvénybe.

A manipulátor dolga, hogy a beérkező érték adatmodell alapján frissítse a szerkesztett komponens adatmodelljét. A kódrészletben az is látszik, hogy a manipulátor index alapján éri el az egyes értékeket. Már említettem, hogy az érték adatmodell struktúrája követi a komponensek hierarchiáját, itt ezt látjuk a gyakorlatban, a lehető legegyszerűbb példán keresztül. A gyökérelem egy lista, és ebből indexelhető nullával és eggyel a két szöveges mező értéke.

4.4.2 Gomb

A gomb komponens érdekessége, hogy alapvetően egy gomb működéséből arra lehetne következtetni, hogy ennek a komponensnek nem kell az érték modellt szerkesztenie. Az én megvalósításomban a gomb megnyomása és elengedése adatmodell változással jár. Erre nem lenne feltétlenül szükség, viszont a keretrendszer egyelőre nem definiál eseményeket az egyszerűség kedvéért. Úgy döntöttem, hogy a célnak megfelelő megoldás az, hogy az érték modellben logikai típus reprezentálja, a gomb állapotát. Így, ha a gomb meg van nyomva, akkor a modellben igaz érték szerepel, ha nincs megnyomva hamis érték szerepel.

Felmerülhet a kérdés, hogy az ilyen és ehhez hasonló tranziens állapotokat hogyan érdemes kezelni. Az általam alkalmazott megvalósításnak érdekes vonzatai vannak. Például az, hogy a gombnyomás a kollaborációs szerkesztés során is minden felhasználónál megjelenik, ez adott esetben meglepő lehet. Viszont egyben hasznos funkció is lehet, mert az adatmodellen történő minden változást egységesen kezel a keretrendszer, így például minden egyes gombnyomás visszakereshető az adatmodell történetében. Persze hogy ez jó vagy rossz, az teljes mértékben a felhasználástól függ.



4. ábra: Gombok méretezése

A fent látható képen öt gomb szerepel egymás alatt. A jobb oldalon látható, hogy a gomb egy mezőt kínál szerkesztésre Text néven. A szöveges mezőbe írt érték azonnal frissül a bal oldalon kijelölt gombban is. A fent látható képen a gombok méretét a tartalmazó elemek szabályozzák, a negyedik gomb azt mutatja be, hogy a tartalmazó elem a szélességre nem ad erős kényszert. Az ötödik gomb esetén a szélességre erős kényszer van megadva, ezt a kényszert a gomb bemutatott implementációja nem tudja teljesíteni, ebben az esetben kilógna a neki szánt területről. Ez a példa olyan esetet mutat be, ahol ez nem elfogadható, így azt a tartalmazó elem levágja. Egy megfelelően implementált gomb komponens képes lehet az erős kényszernek is eleget tenni, mondjuk úgy, hogy a szöveget hármasponttal félbeszakítja úgy, hogy a gomb jobb oldali kerete is helyesen meg tudjon jelenni.

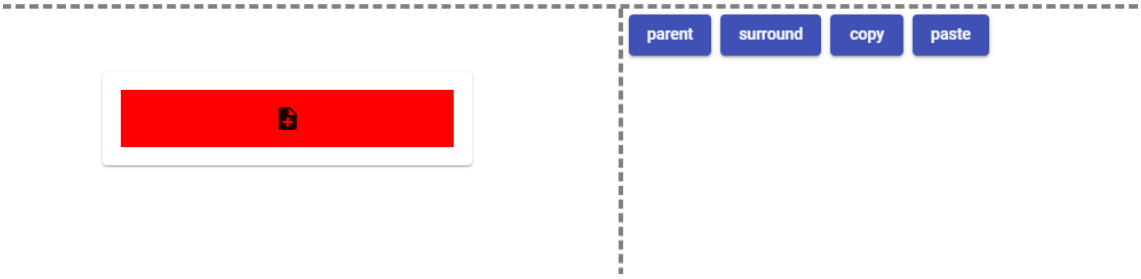
4.5 Tartalmazó komponensek kialakítása

A megjelenítő nézet kialakítása során fontos mérföldkő azoknak a komponenseknek az implementálása, amelyek képesek beágyazott módon komponenseket megjeleníteni. Már említettem, hogy fontos építőeleme a keretrendszernek az üres komponens, amit többek között azért definiáltam, hogy legyen mód arra, hogy az újonnan létrehozott elemek tudják jelezni, hogy az adott helyre tetszőleges komponenst lehet beágyazni.

4.5.1 Kártya

A legegyszerűbb tartalmazó komponens, amit be tudok mutatni az a kártya. A Material Design egyik alapvető stíluseleme, a tartalmazott elemek körül egy vékony keretként jelenik meg, ezenfelül az árnyékolásnak köszönhetően egy kicsit kiemelkedik.

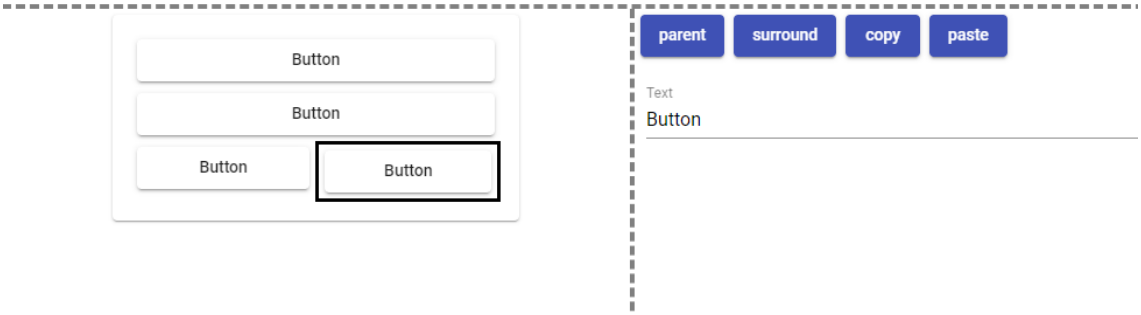
A fejlesztés során egy ponton szükségesnek láttam, hogy extra jelöléseket vezessek be, mert gyakran magam sem láttam át, hogy hogyan épülnek fel a felületek. Az egyik ilyen technikám, hogy különféle színekkel láttam el a speciális jelentőségű elemeket. Erről részletesebben írok majd a szerkesztő bemutatása során, addigis elég annyit tudni, hogy az üres komponens jelölése a piros háttér.



5. ábra: Üres kártya

A fenti a képen látszik egy új kártya, közvetlenül létrehozás után. A piros területre kattintva a szerkesztő automatikusan felkínálja a komponensek listáját, amiből választva tetszőlegesre cserélhetjük.

A következő példában a kártyába lista elemet helyezek, a listát három eleműre bővítem, az első két elembe gombot a harmadik elembe horizontális listát helyezek, aminek szintén mindkét elemébe gombot rakok. Ez a példa bemutatja, hogy hogyan igazodik a kártya a tartalmazott komponensekhez, és hogy nagyjából használat közben mi jár a szerkesztő fejében. Az előbb felsorolt lépések eredménye így jelenik meg a szerkesztőben.



6. ábra: Gombok kártyában

A megértést segíti, ha vetünk egy pillantást az elkészült felület adatmodelljére. Ez az adatmodell egy egyszerűsített változata, viszont a kialakuló hierarchiát hűen ábrázolja, és bevezetést ad a következőkben bemutatott komponensek működésébe.

```
{
     "type": ElementTypes.Card,
     "elements": [
         {
             "type": ElementTypes.ListContainer,
             "elements": [
                 {
                     "type": ElementTypes.Button,
                     "text": "Button",
                 },
                 {
                     "type": ElementTypes.Button,
                     "text": "Button"
                 },
                 {
                     "type": ElementTypes.SplitContainer,
                     "elements": [
                         {
                             "type": ElementTypes.Button,
                             "text": "Button",
                         },
                         {
                             "type": ElementTypes.Button,
                             "text": "Button",
                         }
                     ],
                     "sizes": [
                         1,
                         1
                     ],
                 }
             ]
         }
     ]
 }
```

4.5.2 Lista nézet

A lista nézet létezésére már több ízben utaltam, viszont részeltekben még nem mutattam be, mert úgy gondolom önmagában nehezen megérthető a működése. Alapvetően arról van szó, hogy az adatmodellben egy listában reprezentálja a tartalmazott elemeket, a felületen pedig a tartalmazott elemeket sorban, egymás alatt megjeleníti.

7. ábra: Lista szerkesztése

A fenti ábrán látható lista komponens négy elemű, a harmadik elem egy gomb, az összes többi hely üres komponenssel van feltöltve. A lista komponens ebben a példában bővíthető a bal oldali zöld gombbal, vagy a jobb oldali *Add element* gombbal. A végeredmény ugyanaz, viszont a háttérben a mechanizmust teljesen más.

Az alábbi kódrészlet a lista komponense HTML sablonja. Az *ng-container* elemen levő **ngFor* strukturális direktíva sorban megjeleníti a beágyazott elemeket. Ezután következik az *app-adder* elem, ami egységes megjelenést biztosít az olyan komponensek számára, amelyek bővíthető listákkal dolgoznak. Fontos még azt is megjegyezni, hogy az itt bemutatott lista komponens a zöld hozzáadás gombot csak szerkesztő módban mutatja, megtekintő nézetben nem.

```
<div style="display: flex; width: 100%; flex-direction: column"
  (mouseover)="hover=true"
  (mouseleave)="hover=false"
  (click)="click.emit($event)">
    <ng-container *ngFor="let a of objectKeys(data.elements)">
        <app-container
          (delete)="onDelete($event)"
          style="display: flex;"
          [clickSelect]="clickSelect"
          [options]="options"
          [data]="data.elements[a]"
          (value)="valueChange($event, a)">
        </app-container>
    </ng-container>
    <app-adder (click)="add($event)"
      *ngIf="options.layoutMode && options.canAdd && (options.hoverAdd &&
hover || !options.hoverAdd)"
      style="display: flex; min-width: 40px; height: 50px;"
      [options]="options">
    </app-adder>
</div>
```

Az *app-container* komponens a keretrendszer egyik kulcs eleme. Ennek a komponensnek a részletes bemutatását későbbre hagyom, ezen a ponton annyit érdemes róla tudni, hogy az a dolga, hogy a kódrészletben is látható *[data]* adatkötésen keresztül kapott adatmodell alapján létrehozza a megfelelő komponenst és továbbadja az adatmodellt.

```
  static editorForm(data): any {
    // ...
    const options = (data.elements || ['']).map(x => {
      return elementGen(ElementTypes[x.type]);
    });
    return {
      'type': ElementTypes.ListContainer,
      'elements': [
        {'type': ElementTypes.Button, 'text': 'Add element'},
        {'type': ElementTypes.ListContainer, 'elements': options}
      ]
    };
  }

  static manipulator(data, value): any {
    if (value[0]) {
      data.elements.push({type: ElementTypes.Empty});
    }
    // ...
```

A fenti kódrészlet bemutatja, hogy a fenti ábra jobb oldalán hogyan jelenik meg az *Add element* gomb. Az *editorForm* függvényben a lista komponens definiálja, hogy kijelölés esetén hogyan jelenjen meg a szerkesztőben. Itt látható, hogy a lista első elemeként szerepel a már említett gomb, ezután egy második listában a további opciókat tartalmazza az *options* változó, amit a lista elemeinek értékeként helyettesítek. Ennek az

options változónak a létrehozása is látszik a kódrészleten, viszont az *elementGen* függvényt nem fejtem ki, elég róla annyit tudni, hogy a felületen látható lista négy eleméhez a négy színes sort állítja elő. A négy sor mindegyikének jobb oldalán két gomb látható, ezekkel lehet a lista elemeinek sorrendjét megváltoztatni.

A kódrészletben látható még a lista komponens *manipulator* függvényének eleje, amit a szerkesztő hívhat meg, hogy jelezze a komponensnek, hogy hogyan frissítse magát. Itt a *value[0]* arra utal, hogy az *Add element* gomb meg van-e nyomva. Amint látható, megnyomás esetén a lista új elemmel bővíti magát, aminek típusa üres komponens lesz.

Az *app-container* elem, mint már írtam a keretrendszer központi eleme, egyik fontos szerepe, hogy egységes képet mutat a szülőkomponens felé, attól függetlenül, hogy milyen alkomponenst kell megjelenítenie. A most bemutatott lista komponens használja az app-container *delete* kimenetét. Ha egy beágyazott komponens törölni szeretné magát, akkor erre úgy van lehetősége, hogy elsüti ezt a *delete* eseményt. A *delete* esemény kezelése mindig a tartalmazó, tehát a szülő komponens feladata. Ez az esemény alapú kommunikáció az eddig ismertetett adatmodell alapú kommunikációt kiegészíti. Ezáltal lehetséges, hogy a tartalmazó komponens dönthessen arról, hogy mi történjen az esemény hatására.

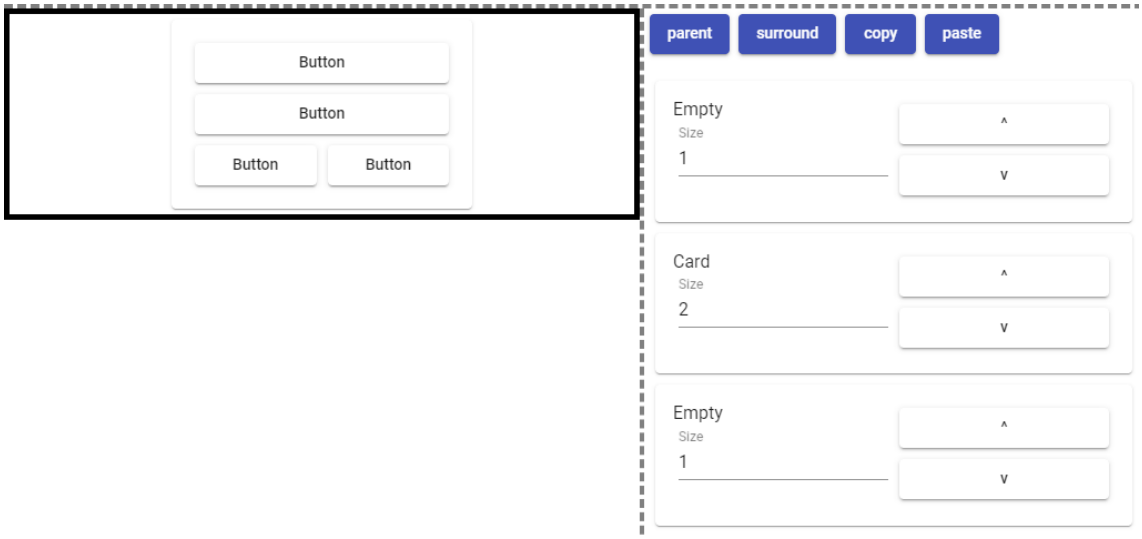
Az általam megvalósított lista komponens úgy viselkedik, hogy mindig legalább két elemmel rendelkezik. Ha egyeleművé válna, akkor automatikusan leegyszerűsíti az adatmodellt, és az egyelemű listát lecseréli úgy, hogy csak a tartalmazott elem maradjon. Hasonlóképpen szerkesztés közben, ha tetszőleges elemet listává alakítjuk, akkor a kiválasztott elem nem eltűnik, hanem egy két elemű listába kerül, ahol második elem üres komponens lesz. Ez jól ábrázolja, hogy a komponenseknek mennyire sok beleszólása van abba, hogy hogyan lehet őket törölni és hogy hogyan lehet belőlük törölni elemeket.

4.5.3 Oszlopos nézet

Az oszlopos nézet nagyban megegyezik a lista nézettel. Eddig nem említettem, hogy a keretrendszerem erősen épít a modern CSS Flexbox[32] elrendezési technikára. A Flexbox hatékonyságát mutatja az is, hogy a lista és az oszlopos nézet között megjelenés

[32]https://en.wikipedia.org/wiki/CSS_Flexible_Box_Layout

szempontjából csak a *flex-direction* értéke különbözik. Lista esetén *column*, oszlopos elrendezésnél pedig *row* a rendezési irány.



8. ábra: Oszlopos elrendezés

A fenti ábrán látható, hogy hogyan néz ki az oszlopos nézet szerkesztő felülete. A jobb oldali panelen a listához hasonló a megjelenés, hiszen itt is lehet az elemek sorrendjét cserélni, és a tartalmazott elem típusa is megjelenik, hogy több elem esetén lehessen tudni, hogy éppen mit módosítunk. Új elem viszont a szöveges doboz, amivel az oszlop mérete állítható. A fenti példában a kártyától jobbra és balra levő üres komponensek szélessége egy egység, a kártya szélessége pedig két egység. Ezek a számok relatív értékek, így a rendelkezésre álló hely négy részre van osztva, amiből a kártya a középső kettőt foglalja el. A Flexbox pont ilyen felületek készítésére lett kitalálva ezért kézenfekvő a használata.

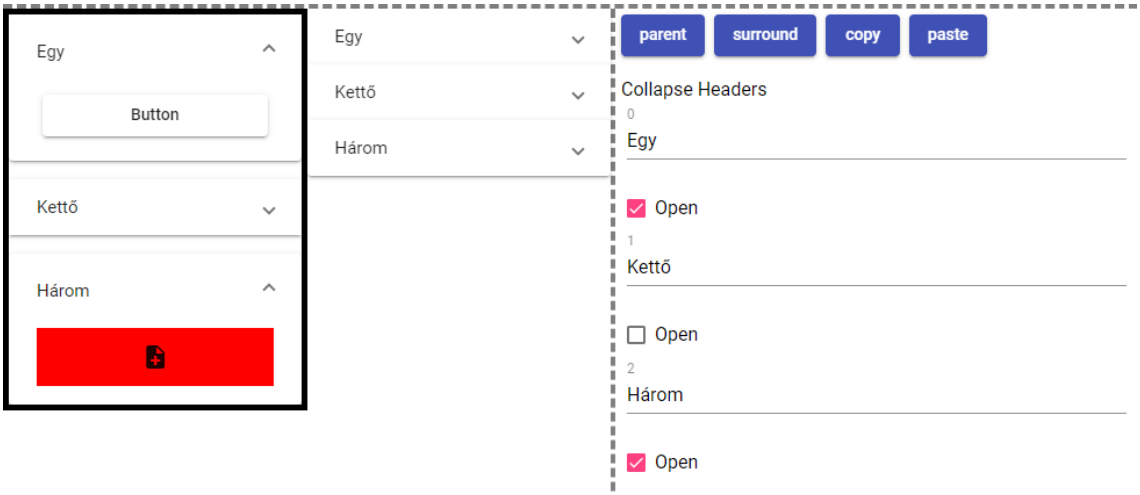
4.5.4 Összecsukható nézet

Az összecsukható nézet a listának egy kifinomultabb változata. Itt ez a komponenst azzal a céllal mutatom be, hogy szemléltessem, hogy nem csak a legalapvetőbb webes megoldásokból lehet építkezni, hanem hogy már létező komponenskönyvtárak elemeit is fel lehet használni.

Ebben a példában azt ismertetem, hogy hogyan emeltem az Angular Material komponenskönyvtár Expansion Panel[33] komponensét a keretrendszerbe. Ez általános

[33] https://material.angular.io/components/expansion/overview

rálátást nyújt arra, hogy hogyan tudok már létező komponenseket felhasználni a keretrendszer komponenseiben.



9. ábra: Összecsukható lista

A bal oldali panelen látható két összecsukható lista mutatja, hogy hogyan néznek ki a lista elemei kinyitott és becsukott állapotban. A piros mező ismét az üres komponenst ábrázolja, melyre kattintva előjön a komponens választó. A bal oldali első kártyába a gomb helyén először ugyanilyen piros mező volt, ezt cseréltem gombra. A jobb oldali panelen látható, hogy melyik elem van éppen kinyitva, vagy becsukva, és hogy mi az adott elem címe. Az alábbi kódrészlet mutatja az összecsukható lista HTML kódjának lényegi részét.

```
<mat-accordion>
  <mat-expansion-panel *ngFor="let a of objectKeys(data.elements)"
      [(expanded)]="data.options[a].open">
      <mat-expansion-panel-header>
          <mat-panel-title>
              {{data.options[a].text}}
          </mat-panel-title>
      </mat-expansion-panel-header>
      <app-container
          *ngIf="true"
          (delete)="onDelete($event)"
          style="display: flex; flex-grow: 1;"
          [clickSelect]="clickSelect"
          [options]="options"
          [data]="data.elements[a]"
          (value)="valueChange($event, a)">
      </app-container>
  </mat-expansion-panel>
</mat-accordion>
```

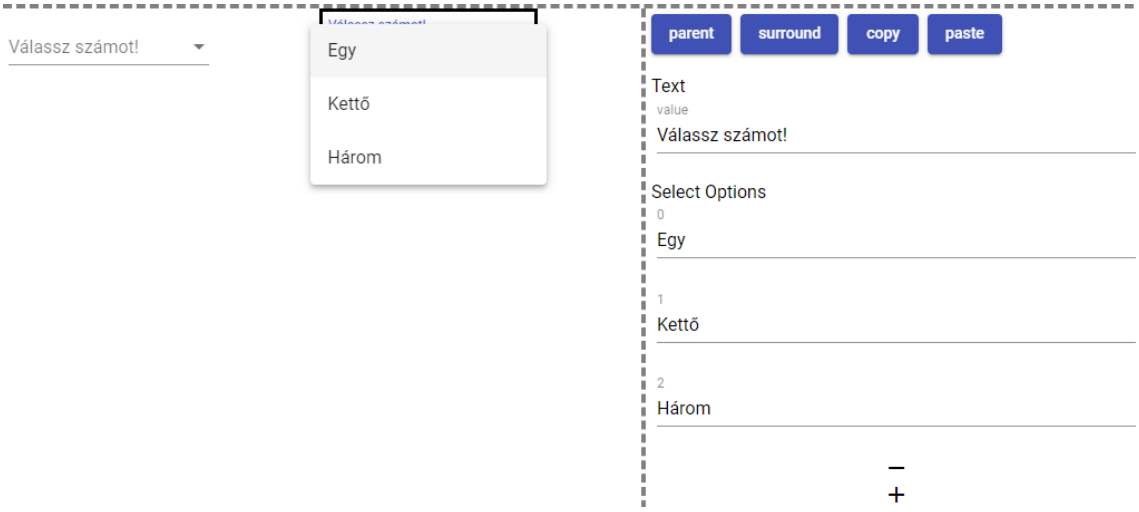
Az Expansion Panel használata nem tér el a szokványostól, a *mat-accordion* elemben van elhelyezve a meghatározott számú *mat-expansion-panel*, melyeknek a

fejlécébe írom az elemhez tartozó szöveget, a törzsébe pedig elhelyezem a már ismerős *app-container* komponenst. Érdemes megfigyelni, hogy a mat-extension-panel *extended* mezőjébe kétirányú adatkötéssel van kötve az adatmodell megfelelő értéke, ami arról gondoskodik, hogy mind az Expansion Panel komponens becsukása frissítse az általam kezelt adatmodell, és hogy az adatmodellem frissítésekor az Expansion Panel is frissítse az állapotát.

4.6 További érdekes komponensek

4.6.1 Legördülő lista

A legördülő lista feladata, hogy egy előre meghatározott értékkészletből enged választani. A választott érték ugyanúgy kezelhető, mint mondjuk a szöveges beviteli mező értéke. Az általam fejlesztett keretrendszer szempontjából a legördülő lista komponens azért érdekes, mert lehetővé teszi az értékkészlet pontos meghatározását. Az alábbi ábra mutatja, hogy hogyan néz ki a legördülő lista alapállásban, és hogy hogyan kínálja fel a választható értékeket.



10. ábra: Legördülő lista megjelenése és a választható értékek listája

A jobb oldali panelen látható az értékkészlet szerkesztésére szolgáló felület. Az első beviteli mező a legördülő lista neve, ami arra utal, hogy minek az értékét határozzuk meg ezzel a beviteli mezővel. Ezután a lehetséges opciók felsorolása látható, melyek mindegyike egy szöveges beviteli mező. A beviteli mezők címkéjében látható, hogy az adott érték éppen hányadik helyet foglalja el a tömbben. Legalul látható egy plusz és egy

mínusz jel. A pluszjellel lehet bővíteni a listát, a mínuszjellel pedig az utolsó elem törölhető.

```
static editorForm(data) {
  const options = (data.options || []).map((x, i) => {
    return { type:  ElementTypes.Input, text: i, value: x };
  });
  return {
    type: ElementTypes.ListContainer,
    elements: [
      { type: ElementTypes.Text, text: 'Text' },
      { type:  ElementTypes.Input, text: 'value', value: data.text },
      { type: ElementTypes.Text, text: 'Select Options' },
      { type: ElementTypes.UserList, text: 'Button', elements: options }
    ]
  };
}

static manipulator(data, value) {
  data.text = value[1];
  data.options = value[3];
}
```

A fenti kódrészlet jól mutatja, hogy milyen könnyű ezeket a szerkesztő felületeket összeállítani az általam fejlesztett felületi leíróval. Az *editorForm* a függvényben először összeállítom a beviteli mező listát az aktuális adatmodell *options* változója alapján, majd ezt illesztem be a fent látható módon. A *UserList* típusú komponens feladatát eddig részletesen nem mutattam be, viszont a nevéből is kitalálható, hogy arra ad lehetőséget, hogy megtekintő nézetben szerkeszthessen a felhasználó listát. Ez a komponens felelős az előbbi ábrán látható jobb oldali bővíthető listáért.

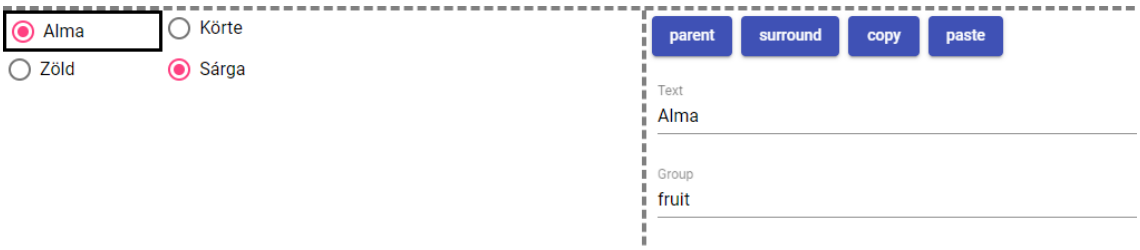
4.6.2 Választógomb

A választógomb komponens neve angolul radio button, melynek feladata, hogy tetszőlegesen sok elem közül egy lehessen kiválasztva. Alapvetően nem egy bonyolult komponens, de mégis rávilágít az általam fejlesztett keretrendszer egy újabb érdekes tulajdonságára. A választógombot úgy szerettem volna megvalósítani, hogy az aktuális dokumentumban tetszőleges helyre beillesztés esetén is megfelelően tudjon működni. Az alapvető koncepcióm az volt, hogy ezek a gombok valamilyen módon csoportokba rendezhetők úgy, hogy az egy csoporton belüli választógombok közül mindig pontosan egy lehessen aktív. Ebből a definícióból következik, hogy egy adott választógombnak összeköttetésben kell lennie az összes vele egy csoportban lévő választógombbal, mert aktiválás esetén értesíteni kell, hogy a csoport kiválasztott eleme egy másik elemre módosult.

A keretrendszerem nem ad egységes megoldást erre problémakörre, mivel az azzal járna, hogy a komponensek rendszeresen feltérképezik a teljes adatmodellt azért, hogy megtalálhassák azokat a komponenseket, amelyekkel együtt kell működniük.

A már korábban is említett Angular Material komponenskönyvtár választógomb komponense is pont ugyanígy értelmezi a problémát, és szem előtt tartja, hogy bizonyos esetekben a választógombok a DOM struktúrában szétszórva helyezkedhetnek el. A megoldás erre, hogy *MatRadioGroup* segítségével az egymástól független elemek közt ki tud alakulni a kapcsolat.

Úgy döntöttem, hogy ez a leginkább járható út arra, hogy azonos típusú, de egymástól teljesen független komponenspéldányok kommunikálni tudjanak. Amennyiben további ilyen típusú komponensre van igény, érdemes alapul venni az Angular Material *RadioButton* komponensben megvalósított szinkronizációs megoldást.



11. ábra: Választó gomb megjelenése

A fenti ábrán látható, hogy a bemutatott komponens hogyan jelenik meg a szerkesztőben. A bal oldali panelen két választási csoport van, mindkét csoportban két elemmel. A jobb oldalon látszik, hogy az Alma feliratú opció a gyümölcsök csoportjába tartozik. Ha a Körte lehetőséget jelölöm ki akkor értelemszerűen az lesz kiválasztva az Alma helyett. Ugyanilyen módon, de a gyümölcsöktől függetlenül lehet a két szín közül választani. A csoport megadása tetszőleges objektummal történhet az Angular Material szerint, amennyiben összehasonlítás esetén egyenlőséget mutatnak, akkor ugyanabba a csoportba tartoznak. A fenti példában látható, hogy szöveges értékként lehet a csoportot meghatározni, ezt célszerű úgy megadni, hogy utaljon arra, hogy milyen opciók közül választhatunk.

4.7 Stílusozás

A keretrendszerem szabad kezet ad arra nézve, hogy az egyek komponenseket milyen stílusokkal láthatjuk el. Az eddig bemutatott példákból már nyilvánvaló, hogy

tetszőleges Angular komponenskönyvtárra építhetnek a keretrendszeremmel kompatibilis komponensek. Az én kedvencem az Angular Material, és az eddigiekben csupa jó tapasztalatom volt vele.

Egy másik érdekes megközelítés lehet valamilyen CSS keretrendszer használata. A legelterjedtebb ilyen jellegű keretrendszerek a Bootstrap[34] és a Tailwind[35]. Ezek legfontosabb különbsége egy Angular komponenskönyvtárhoz képest, hogy csakis globálisan definiált CSS szabályokra alapoznak. Ez azt jelenti, hogy ha egy komponens függ mondjuk a Tailwind keretrendszer szabályaitól egy másik komponens függ a Bootstrap szabályaitól akkor a két komponens csak akkor használható megbízhatóan egyszerre ugyanabban a dokumentumban, ha nincs ütközés a két keretrendszer szabályai között. Persze ez a probléma megkerülhető, ha a keretrendszerek minden szabályát prefix-el látjuk el, de ezt én nem javaslom, mert csak megtévesztő, ha ugyanarra a célra több hasonló megoldás is van főleg, ha ezek csak egy kicsi térnek el egymástól.

Sokkal inkább előremutatónak tartom azt a fajta modularizált stílusozást amit az Angular alapvető funkcióként kínál. Az egyes CSS szabályok csak az aktuális komponenshez definiált sablonban kerülnek érvényre. Az ilyen módon hatáskörrel ellátott szabályok nem befolyásolhatják más komponensek működését, így az esetleges hibák sokkal könnyebben felderíthetők, és sokkal egyszerűbben javíthatók, mert biztosak lehetünk benne, hogy a változtatásnak nem lesz nem várt mellékhatása a komponensen kívül.

4.7.1 Másolat vagy feltételes működés

Egy fontos fejlesztői kérdés, hogy mikor érdemes módosítani egy komponenst, és mikor kell azt lemásolni, és függetlenül kezelni. Egy remek példa erre a listás és az oszlopos komponens. Amikor erről volt szó, ott is említettem, hogy a két komponens gyakorlatilag azonos egy CSS tulajdonságot leszámítva. Ennek ellenére én mégis külön komponensként kezelem ezeket az elemeket, aminek az az oka, hogy koncepcionálisan szerintem ez így helyes, még akkor is, ha ez egy kicsit kevésbé karbantartható.

[34] https://getbootstrap.com/

[35] https://tailwindcss.com/

Ebben az alfejezetben mégis felvázolom, hogy hogyan nézne ki, ha ezt a két komponenst egy paraméterezhetőre cserélném. A leginkább szembetűnő változás az lenne, hogy eggyel több paramétere lenne a komponensnek, ami magában nem gond, de a szerkesztő alapvetően ezt nem tudja figyelembe venni, így beszúráskor ugyanazt a komponenst kéne választani, akkor is, ha egymás alatt vagy akkor is egymás mellett szeretnénk látni az elemeket. Szerintem ez önmagában meggyőző, de tovább erősíti az álláspontom, hogy hamar jelentkezett az igény arra, hogy megadható legyen, hogy az egyes oszlopok milyen arányban töltik ki a rendelkezésre álló helyet. Ehhez további CSS szabályok is kellettek, de nagyobb különbség, hogy a szerkesztőben megjelenő komponens szerkesztő felületnek is fogadnia kell a szélesség értékeket, illetve az értékek feldolgozásához is új logikát kell írni.

4.7.2 CSS változók használata

A CSS lehetőséget nyújt arra, hogy változókat definiáljunk, majd ezeknek a változónak az értékét hivatkozzuk az egyes CSS szabályokban. Ezek használatával könnyen és karbantarthatóan definiálhatunk olyan értékeket, amelyek több komponensre egyszerre vonatkozik. Egy jó példa lehet CSS változók használatára a sötét mód megvalósítása. Én ezzel a kérdéssel részletesen nem foglalkoztam, így ilyen módot sem készítettem, de azt mindenképpen szeretném kiemelni, hogy a CSS változók használatának gondolata teljes mértékben összeegyeztethető az általam fejlesztett keretrendszerrel.

4.8 Új komponens készítése fejlesztőként

Már jónéhány komponenst bemutattam, de mégsem adtam átfogó képet arról, hogy mi a menete egy új komponens létrehozásának, és pontosan mit kell tenni, ahhoz, hogy az új komponens a szerkesztőben is megjelenjen.

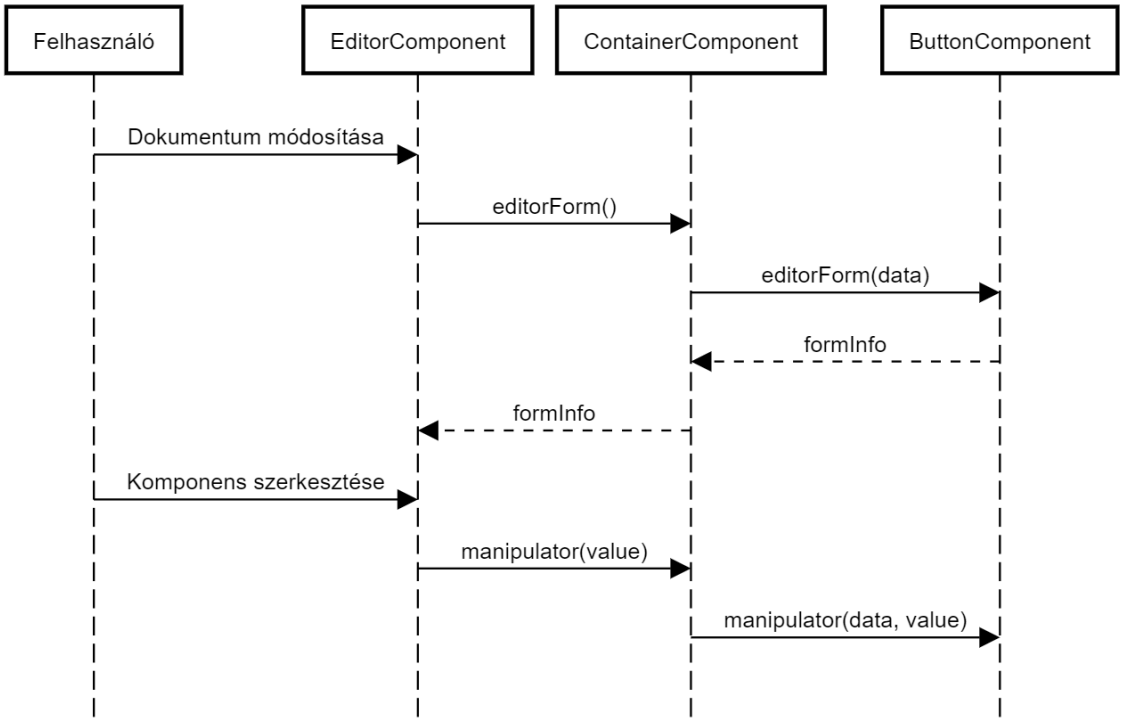
Az első lépés, hogy létre kell hozni egy új Angular komponenst a kívánt mappában, amire legegyszerűbb megoldás az Angular CLI[36] használata. Konvencióm szerint az *app/components* mappába kerülnek az általános célú komponensek. A kívánt mappába lépve az *ng g c <komponens neve>* parancs futtatása létrehozza az új

[36] Angular fejlesztői eszköz, parancssorból hívható utasításokkal segíti az Angular projektek karbantartását.

komponenst. A parancs hatására egy új mappában létrejön három fájl, egy *.ts* egy *.html* és egy *.css* fájl. Ezenkívül a releváns modul fájlba is bekerül az új komponens deklarációja.

Ezzel még nem végeztünk, eddig csak egy Angular komponenst készítettünk, ahhoz, hogy az általam készített szerkesztő kezelni tudja a komponenst, még további lépések szükségesek.

Az új komponensnek meg kell valósítania a *DeformatorComponent* osztály publikus interfészét. Itt ajánlatos az *extends* kulcsszó használata, így a statikus kódelemzés is segíti a munkánkat. A *DeformatorComponent* három függvény implementálását várja el, név szerint ezek az *editorForm* a *manipulator* és az *onEditorAdd* függvények.



12. ábra: Komponens szerkesztő működése

Az *editorForm* és a *manipulator* működését ismertettem korábban, a fenti ábrán látható hogyan frissül a szerkesztő állapota, és hogy az hogyan frissíti a komponens állapotát.

Az *onEditorAdd* függvényről eddig még nem esett szó. Ennek a függvénynek akkor van szerepe, amikor a komponens egy másik helyére kerül. A csere pillanatában az új komponens megkapja a régi adatstruktúrát, aminek a helyét át fogja venni. Hogy ezzel

az adatmodellel mit csinál a komponens arról szabadon dönthet. Az egyszerűbb komponensek ezt simán figyelmen kívül hagyják, viszont ahogy már említettem a lista képes a lecserélt elemet az első elemmé tenni.

A komponensekkel kapcsolatban formai követelmény csak az előbbi három függvény megvalósítása, viszont érdemi működéshez nélkülözhetetlen még az adatmodell átvétele, és az is szükséges, hogy a komponens jelezze, ha rákattintottak, ez alapján tudja a szerkesztő, hogy éppen melyik komponens van kijelölve.

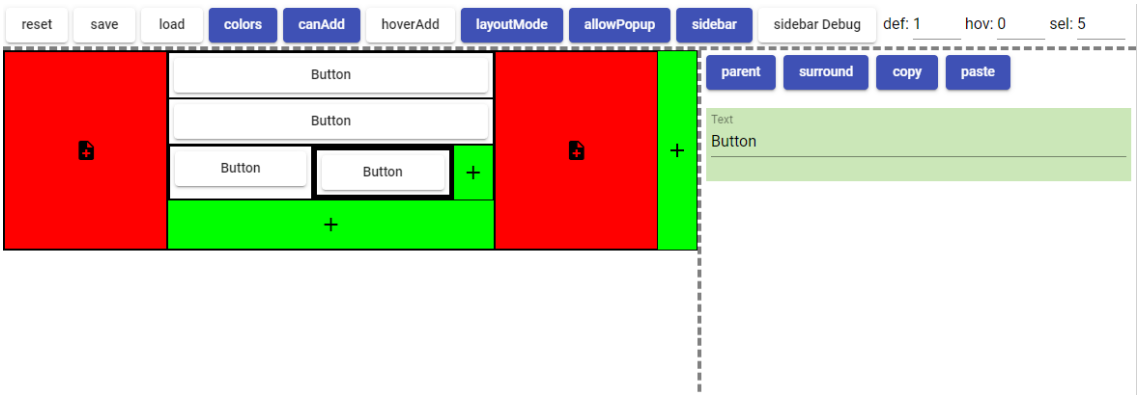
4.9 Szerkesztő

A szerkesztő eszköz funkcionalitásáról már sokat meséltem, ebben a fejezetben bemutatom a fő elemeit és funkcióit. Az alapvető elképzelés az, hogy ezzel a szerkesztővel, vagy ennek egy finomított változatával lehet összeállítani tetszőleges felhasználói felületeket. Egyelőre közel sem felhasználóbarát, inkább egy fejlesztői demónak mondhatót.

4.9.1 A szerkesztő felépítése

A szerkesztő felépítését tekintve három fő panelre osztódik, ezek között látható a vastag szaggatott szürke vonal.

A felső sáv a szerkesztő általános beállításait tartalmazza, amely beállítások a teljes szerkesztőre általánosan vonatkoznak, a bal oldali panel a fő szerkesztő terület, a jobb oldali pedig az aktuálisan kiválasztott komponens szerkesztésére szolgál. A felső sáv gombjai az első hármat leszámítva mind egy-egy bekapcsolható funkciót jelentenek. Ha a funkció be van kapcsolva akkor kék a gomb.



13. ábra: Szerkesztő felület

4.9.2 Általános beállítások

A felső sáv első három gombja az aktuális adatmodell manuális mentésére szolgál. Ezeknek a kollaborációs funkciók bevezetése előtt volt nagy jelentősége, mert ezekkel a gombokkal tudtam perzisztálni egy-egy adatmodellt. A mentés a böngésző LocalStorage tárolójába sorosítja az adatmodellt, a betöltés ezt értelmezi és tölti vissza a keretrendszerbe. A kollaborációs szinkronizáció ezt automatikusan végzi a háttérben minden helyi és távoli változtatás után. A *reset* gomb továbbra is teljes értékű, ennek megnyomására a gyökér komponens üres komponensre cserélődik.

A *colors* gombbal lehet ki-be kapcsolni, hogy a fenti ábrán is látható színekkel legyenek-e ellátva a kiemelt jelentőségű komponensek.

A *canAdd* gomb a fenti ábrán látható zöld elemek megjelenését szabályozza. Bekapcsolt állapotban ezek az elemek megjelennek, kikapcsolt állapotban nem.

A *hoverAdd* gomb lehetővé teszi, hogy csak azon komponensekben jelenjen meg a zöld hozzáadás gomb, melyek felett éppen lebeg a kurzor.

A *layoutMode* egy összetettebb funkció, bekapcsolt állapot reprezentálja a szerkesztő módot, kikapcsolt állapot jelenti a megtekintő módot. Ez a beállítás több különféle viselkedést fog össze. Például a zöld hozzáadás gombok csak és az üres elemek csak akkor látszanak, ha ez a beállítás bekapcsolt állapotban van.

Az *allowPopup* gombbal lehet letiltani a komponensválasztó megjelenését, mivel ezt bizonyos szituációkban zavarónak tartottam.

A *sidebar* gombbal a jobb oldali panelt lehet becsukni vagy kinyitni, mivel erre megtekintő nézetben nincs szükség.

Az *allowDebug* gommbal pedig a jobb oldali panelen lehet további fejlesztői információkat megjeleníteni.

4.9.3 Kiválasztott komponens szerkesztése

A jobb oldali panelen további négy gomb található, ezek az aktuálisan kiválasztott komponensre vonatkoznak.

A *parent* gomb segítségével lehet a jelenlegi komponens szülőjét kiválasztani, ha a gyökérelem van kiválasztva, akkor semmit nem csinál.

A *surround* funkcióval lehet úgy komponenst beszúrni, hogy a jelenleg kiválasztott legyen az első eleme az újonnan beszúrt komponensnek. A gomb megnyomására egyből a komponensválasztó funkció nyílik fel.

A *copy* és *paste* funkcióval lehet a kijelölt elemet vágólapra helyezni, majd a beilleszteni. Fontos megjegyezni, hogy a beillesztés a kijelölt elemet cseréli le a vágólap tartalmára.

4.10 Hálózati kommunikáció

Eddig sok szó volt az adatmodellről, és arról, hogy hogyan működik a szerkesztő, melyek fejlesztése során végig észben kellett tartanom a végső célom, miszerint a rendszer alkalmas a valós idejű kollaboráció támogatására. Az megjelenítő tervezése és fejlesztése során végig szempont volt, hogy egy központi adatmodell maradéktalanul meghatározza az adott felületet. Ennek most lesz igazán jelentősége, hogy az állapot szinkronizálásra kerül több szerkesztő között.

4.10.1 Yjs

Az Yjs egy JavaScript alapú CRDT[37] megvalósítás [17], aminek segítségével könnyedén megoldható, hogy az adatmodellen végzett párhuzamosan végrehajtott módosítások végül egységes állapotba konvergáló adatmodellt hozzanak létre. Fontos kikötés, hogy minden elemi módosítás után is formailag érvényes adatmodell jöjjön létre.

A megvalósításomban a *ManagerComponent* felelőssége, hogy az adatmodellt a hálózaton keresztül szinkronizálja. Egyrészt szükségszerű, hogy kialakítsa a hálózati kapcsolatokat és hogy a helyi változtatásokat közölje a többi fél felé. Ezenkívül feladata még, hogy a beérkező üzenetek mentén az új adatmodellt a szerkesztő felé frissítse.

Az Yjs alapjában csak az adattípusokat tartalmazza, de az ilyen módon létrehozott dokumentumok szinkronizálására több csomag is elérhető. A legjelentősebb az *y-webrtc* és az *y-websocket*. Az előbbi peer-to-peer módon, központi kiszolgáló nélkül képes szinkronizálni a változásokat, az utóbbi központi kiszolgáló segítségével végzi el ugyanezt. Mindkét implementációval működőképes az általam fejlesztett keretrendszer.

[37] Conflict-free replicated data type, magyarul konfliktusmentes replikált adattípus

A fejlesztés és tesztelés során inkább a WebScoket alapú implementációt használtam, mert ez az utolsó állapotot megőrzi akkor is, ha minden felhasználó lecsatlakozott.

4.10.2 Automerge

Az Automerge szintén egy CRDT implementáció, ami kifejezetten JSON formátumú dokumentumok szinkronizálására lett kifejlesztve. A legnagyobb különbség, hogy az Automerge objektumok immutábilisek[38] ami különösképpen fontos a React és a Redux[39] rendszerrel való együttműködés során.

Az én keretrendszerem képes működni mindkét megoldással, mivel nem éltem azzal a feltételezéssel, hogy az adatmodell immutábilis.

Az Yjs hangsúlyozza, hogy az objektumok mutálhatók, ennek köszönhetően hatékonyabban tudja kezelni a nagy méretű szöveges fájlokat is, ami azért jelentős, mert az egyik legfontosabb alkalmazása ezeknek az adattípusoknak az elosztott szövegszerkesztés. Több olyan összehasonlítás is elérhető az interneten, melyek e két keretrendszer teljesítménybeli különbségeit vizsgálják. Ezek többnyire azt mutatják, hogy az Yjs jelentősen hatékonyabb mind erőforrás, mind késleltetés szempontjából [18].

A számok ellenére mégis az Automerge a népszerűbb keretrendszer, a saját tapasztalataim szerint jó okkal. Az Automerge sokkal megközelíthetőbb a tipikus fejlesztő számára, sokkal könnyebben beépíthető egy alkalmazásba, ezenkívül rendelkezik kényelmi funkciókkal is, például az utolsó lépés visszavonására, vagy a visszavonás visszavonására kész megoldásokat kínál.

Az Yjs nyilvánvalóan egy nyersebb megvalósítás, ami egyszerre előnyt és hátrányt is jelent. Előny, mert jelentősen hatékonyabban tudja ugyanazt a szinkronizációs problémát megoldani, viszont hátrány, mert nem tud olyan könnyen kezelhető eszközöket kínálni.

---

[38] Olyan objektumok, amelyek módosítása során új objektum jön létre, az eredeti objektum nem módosítható.

[39] Alkalmazás állapot kezelő Reacthoz

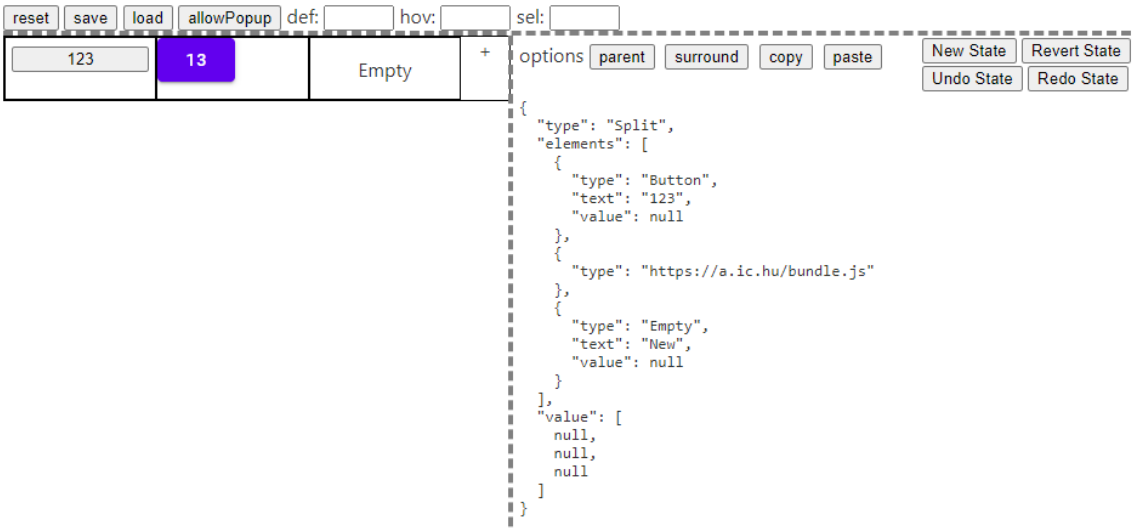
4.11 Svelte

A keretrendszer fejlesztése során sok érdekes problémával és megoldással volt dolgom, jobban megismertem a választott technológiák képességeit és határait.

Csak a fejlesztés közben szembesültem azzal a gondolattal, hogy milyen jó lenne, ha lehetséges lenne menet közben a szerkesztőben új komponenseket betölteni. Ezt viszont az Angular alapvetően nem támogatja. Különféle megoldások léteznek arra, hogy hogyan lehet az alkalmazást darabokban betölteni, de ezek egyike sem teszi lehetővé, hogy külön fordított komponenseket lehessen betölteni.

Nekem az volt az álmom, hogy a komponenseket a fejlesztők egymástól függetlenül elkészíthetik, lefordíthatják és publikálhatják, erre viszont az Angular egyelőre nem képes.

A Svelte[40] a korábban említett keretrendszerekkel szemben inkább egy fordító, mint keretrendszer, így lehetővé teszi, hogy komponensek egymástól függetlenül legyenek lefordítva, és futás közben betöltve az alkalmazásba. Ezen gondolat mentén kezdtem keretrendszerem második változatának fejlesztésébe, amihez ezt a napjainkban inkább feltörekvőnek mondható megoldást név szerint a Svelte-t választottam.



14. ábra: Svelte prototípus

A fenti ábrán egy prototípus látható, amit már Svelte-ben készítettem, a lényeg, hogy a lila gomb egy külön fordított komponens. A szerkesztőben csak a jobb oldalon

[40] https://svelte.dev/

látható linket adtam meg. Ezután a szerkesztő betöltötte az új komponenst és meg is jelenítette.

A Svelte további érdekes megoldásokat kínál, amiket itt már nem részletezek, de mindenképpen fontosnak tartottam megemlíteni a pozitív tapasztalatom.

4.12 Beüzemelés

A kollaborációs keretrendszerem önmagában alkalmas dokumentumok készítésére és megjelenítésére, ezenkívül valósidejű kollaborációt tesz lehetővé a hálozati kommunikációs megoldások segítségével.

Mivel ez rendszer már magában, további fejlesztői beavatkozás nélkül is érdekes lehet felhasználók számára, ezért fontosnak tartottam, hogy a munkám során született megoldás könnyen beüzemelhető legyen akár minimális szakértelemmel is.

Ezen a területen a konténerizácós technológia jelent áttörést, esetemben a legfontosabb előnye a hordozhatóság. Ez azt jelenti, hogy az általam készített csomag teljes mértékben meghatározza a futtatókörnyezettel szemben állított követelményeit és a telepítés lépéseit, így a beüzemelési folyamat teljes mértékben reprodukálható.

```
### Build ###
FROM node:15-alpine AS build
WORKDIR /usr/src/app
COPY package.json package-lock.json ./
RUN npm install
COPY . .
RUN npm run build
### Run ###
FROM nginx:1.19-alpine
COPY nginx.conf /etc/nginx/nginx.conf
COPY --from=build /usr/src/app/dist/deformator /usr/share/nginx/html
```

A fenti kódrészlet mutatja az általam használt Docker[41] konténerizácós megoldás konfigurációs fájlját, ami egy parancs futtatásával elkészíti a keretrendszerem *image* fájlját, ezután egy paranccsal futtatható a keretrendszer. Ez a két parancs az alábbi kódrészletben látható.

```
docker build -t deformator-multistage .
docker run -d -p 80:80 deformator-multistage
```

[41] https://www.docker.com/

5 Tesztelés

Ha karbantartható és továbbfejleszthető szoftvert szeretnénk készíteni kulcs fontosságú, hogy az tesztelhető is legyen. Meglátásom szerint a tesztelés fontossága a szoftver komplexitásával arányos. A fejlesztés során hamar világossá vált számomra, hogy célszerű automatizált tesztesetekkel ellátnom a keretrendszerem.

A komponensek belső működésének tesztelését már korábban volt lehetőségem megismerni, és úgy ítéltem meg, hogy ennél sokkal hasznosabb lenne a komponensek közti interakciók tesztelése.

Ezen interakciók viszont gyakran nem jelennek meg változók és függvényhívások formájában, hanem a felületi megjelenés az interakció.

Korábban már hallottam már az end-to-end[42] tesztelésről, és kíváncsi voltam, hogy hogyan tudnám ezeket a technikákat felhasználni munkám során.

5.1 Automatizált felületi tesztek

Végül úgy döntöttem, hogy automatizált felületi teszteket készítek. A választásom azért esett erre, mert így tesztelni tudom a komponensek közötti interakcióknak azt a részét is, ami csak a képernyőn látszik.

Erre a feladatra ideálisak az úgynevezett fej nélküli böngészők, amelyek teljesértékű böngészőként funkcionálnak, viszont nem jelennek meg a monitoron, csak virtuálisan adnak képet. Ez arra ideális, hogy egyes webes feladatokat automatizáljunk például azt, hogy egy webalkalmazásról képeket készítsünk.

Én a Puppeteer[43] nevű könyvtárat használtam, ami a háttérben Google Chrome böngésző segítségével dolgozik. A Puppeteer könnyen használható Node.js alapú programozói felületet biztosít.

Különféle megoldások léteznek arra, hogy hogyan lehet felvenni felhasználói interakciót, majd Puppeteer segítségével visszajátszani. Ennek egy hátránya lehet, hogy

[42] röviden E2E tesztelés melynek lényege, hogy egy teljes folyamatot elejétől végéig tesztel.

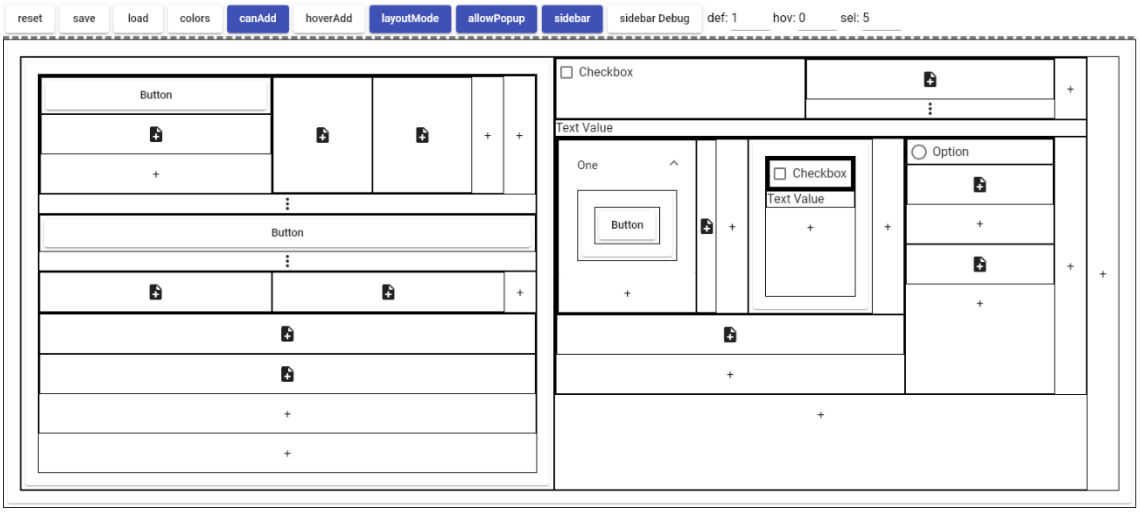
[43] https://pptr.dev/

az oldal nem mindig ugyanolyan gyorsan tölt be minden tesztelésnél. Ha esetleg egy alkalommal valami miatt lassan tölt be az oldal, akkor az téves pozitívot produkál, hiszen az alkalmazás rendeltetés szerűén működik csak a gombra a gomb megjelenése előtt kattintott az automatizált teszt.

Végül a legjobb megoldásnak az bizonyult, hogy a teszteseteket is automatikusan generálom. Az egérrel véletlenszerűen, de reprodukálható módon kattintok a különböző komponensekre, majd a lehetséges választási lehetőségek közül szintén véletlenszerűen, de reprodukálható módon választok. Egy ilyen eseménysorozatnak a végeredményeként komponensek véletlenszerű kompozíciójaként alakul ki egy dokumentum.

Ha több ilyen eseménysorozatot generálok akkor egészen jó képet kapok arról, hogy a komponensek hogyan viselkednek együttesen.

Az alábbi képen látható felület száz véletlenszerű lépés eredménye.



15. ábra: Automatikusan generált felület

Ez magában is érdekes, de különösen hasznos tud lenni, ha a fejlesztés lépései között rendszeresen lefuttatjuk, és a különbségeket vizsgáljuk.

A különbségek feltérképezéséhez a *resemblejs* csomagot használtam, ami képek összehasonlítását és képek közti különbségek felfedezését teszi lehetővé. Az általam készített tesztrendszerben megadható, hogy hány százaléknyi eltérést vegyen figyelembe. Úgy találtam, hogy a teljes egyezőség nem elvárható tesztelések között, mert az eltérő betöltési idő miatt a kurzor villogása is eltérést generálhat.

Ez a teljesen automatizált tesztelés kiegészíthető fejlesztő által megadott tesztesetekkel. Esetembe egyszerű interakciók, így kattintások pozíciója, és a kattintások közötti várakozási idő megadása elegendő a legtöbb esetben.

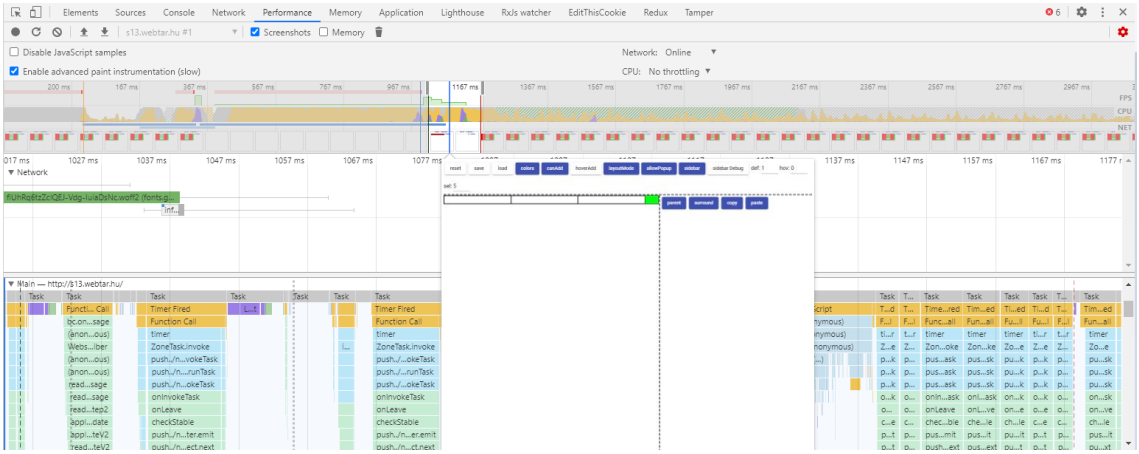
A tesztelés során tíz ilyen tesztet is készítettem, ezek egy-egy egyszerű felületet készítenek el automatikusan.

5.2 Késleltetés

A fejlesztés során fontosnak tartottam, hogy a megoldásom akár egy éles rendszerben is megállja a helyét. Itt különösen fontosnak tartom a megjelenítő nézet teljesítményét, hiszen ez magában is beágyazható, és a szerkesztő is erre épít.

A tesztjeim alapján azt a következtetést vontam, le, hogy nincs nagyságrendbeli különbség a megszokott Angular felületek és a keretrendszeremmel összeállított felületek között teljesítmény szempontjából.

Az alábbi képen látható egy teljesítménymérés eredménye, melyen az látható, hogy 120ms telik el az oldal első megjelenése és a friss állapot megjelenése között úgy, hogy ez az állapotot publikusan internetről lett betöltve a kollaborációs megvalósítás által.



16. ábra: Teljesítmény elemzése Chrome-al

6 Irodalomjegyzék

- [1] „Front-end frameworks and libraries,” 2019. [Online]. Available: https://2019.stateofjs.com/front-end-frameworks/. [Hozzáférés dátuma: 5. december 2020.].
- [2] „Angular TypeScript configuration,” [Online]. Available: https://angular.io/guide/typescript-configuration. [Hozzáférés dátuma: 6. december 2020].
- [3] „Material Design by Google,” Google, [Online]. Available: https://design.google/resources/. [Hozzáférés dátuma: 6. december 2020].
- [4] „Usage statistics of content management systems,” [Online]. Available: https://w3techs.com/technologies/overview/content_management. [Hozzáférés dátuma: 7. december 2020].
- [5] „Figma Prototyping Features,” [Online]. Available: https://www.figma.com/prototyping/. [Hozzáférés dátuma: 1. december 2020.].
- [6] „Visly - Building your first component,” [Online]. Available: https://visly.app/docs/building-your-first-component. [Hozzáférés dátuma: 8 december 2020].
- [7] „Visly - Designing Primitives,” [Online]. Available: https://visly.app/docs/designing-primitives. [Hozzáférés dátuma: 8. december 2020].
- [8] Y. Sun, D. Lambert, M. Uchida és N. Remy, *Collaboration in the Cloud at Google,* research.google.com, 2014.
- [9] „Websocket története,” [Online]. Available: https://en.wikipedia.org/wiki/WebSocket. [Hozzáférés dátuma: 3. december 2020.].
- [10] C. Severance, „JavaScript: Designing a Language in 10 Days,” február 2012. [Online]. Available:

https://www.computer.org/csdl/magazine/co/2012/02/mco2012020007/13rRUy08MzA. [Hozzáférés dátuma: 8. december 2020].

[11] S. Dutta, „IEBlog," 2006. [Online]. Available: https://docs.microsoft.com/en-us/archive/blogs/ie/. [Hozzáférés dátuma: 3. december 2020.].

[12] P. Krill, „React: Making faster, smoother UIs for data-driven Web apps," InfoWorld, 2014. [Online]. Available: https://www.infoworld.com/article/2608181/react--making-faster--smoother-uis-for-data-driven-web-apps.html. [Hozzáférés dátuma: 3. december 2020.].

[13] „Vue.js Render Functions & JSX," [Online]. Available: https://vuejs.org/v2/guide/render-function.html. [Hozzáférés dátuma: 5. december 2020].

[14] „WebSocket RFC," 2011. [Online]. Available: https://tools.ietf.org/html/rfc6455. [Hozzáférés dátuma: 3. december 2020.].

[15] S. Saffron, „WebSockets, caution required!," 2016.. [Online]. Available: https://samsaffron.com/archive/2015/12/29/websockets-caution-required. [Hozzáférés dátuma: 3. december 2020.].

[16] „Angular Binding syntax," [Online]. Available: https://angular.io/guide/binding-syntax. [Hozzáférés dátuma: 5. december 2020.].

[17] P. Nicolaescu, K. Jahns, M. Derntl és R. Klamma, Yjs: A Framework for Near Real-Time P2P Shared Editing on Arbitrary Data Types, 2015.

[18] K. Jahns, „CRDT benchmarks," [Online]. Available: https://github.com/dmonad/crdt-benchmarks. [Hozzáférés dátuma: 8. december 2020.].

[19] K. Nahtkasztlija, „Az idegen szavak toldalékolása," június 2009. [Online]. Available: http://www.pcguru.hu/blog/kredenc/az-idegen-szavak-toldalekolasa/5062.

[20] P. Koopman, „How to Write an Abstract," október 1997. [Online]. Available: https://users.ece.cmu.edu/~koopman/essays/abstract.html. [Hozzáférés dátuma: 20 október 2015].

[21] W3C, „HTML, The Web’s Core Language,” [Online]. Available: http://www.w3.org/html/. [Hozzáférés dátuma: 20 október 2015].

Függelék