SZAKDOLGOZAT

Biró Dávid

2019

Pannon Egyetem
Műszaki Informatika Kar
Rendszer- és Számítástudományi Tanszék
Programtervező informatikus BSc szak

SZAKDOLGOZAT

Webalapú ER modellező

Biró Dávid

Témavezető: Machalik Károly

2019

Ide jön az eredeti vagy a fénymásolt feladatkiírás.

Nyilatkozat

Alulírott Biró Dávid diplomázó hallgató kijelentem, hogy a szakdolgozatot a Pannon Egyetem Rendszer- és Számítástudományi Tanszékén készítettem Programtervező informatikus BSc szak (BSc in Computer Engineering) megszerzése érdekében.
Kijelentem, hogy a szakdolgozatban lévő érdemi rész saját munkám eredménye, az érdemi részen kívül csak a hivatkozott forrásokat (szakirodalom, eszközök, stb.) használtam fel.
Tudomásul veszem, hogy a szakdolgozatban foglalt eredményeket a Pannon Egyetem, valamint a feladatot kiíró szervezeti egység saját céljaira szabadon felhasználhatja.
Veszprém, 2019. május 10.

Aláírás

Alulírott Machalik Károly témavezető kijelentem, hogy a szakdolgozatot Biró Dávid a Pannon Egyetem Rendszer- és Számítástudományi Tanszékén készítette Programtervező informatikus BSc szak (BSc in Computer Engineering) megszerzése érdekében.
Kijelentem, hogy a szakdolgozat védésre bocsátását engedélyezem.

Veszprém, 2019. május 10.

Aláírás

Köszönetnyilvánítás

Köszönettel tartozom Machalik Károlynak, aki segített a szakdolgozat formai követelményeinek betartásában, valamint a szakdolgozat előadási módjának fejlesztésében. Köszönettel tartozom a Phoenix framework fejlesztőinek, hogy biztosítottak egy hasonló kódgenerátort, ami alapján ez a szakdolgozat létrejöhetett.

TARTALMI ÖSSZEFOGLALÓ

A szakdolgozat témája egy webalapú ER modellező készítése. Ezen belül is egy megfelelő eszköz biztosítása a rapid-prototyping fejlesztési módszer segítésére, első sorban web alapú alkalmazásokhoz.

A webes alkalmazások nagy része tartalmaz olyan közös elemeket, amelyek lehetővé teszik, hogy akár kevés információval is, de generálhatóak legyenek.

A szakdolgozatban bemutatott program egy módosított Entity Relationship modellt használ a legenerálandó erőforrások definiálására. Ehhez a modellhez biztosít egy tervező felületet, hogy a generátor használata a lehető legkényelmesebb, és leggyorsabb legyen.

Az átláthatóság érdekében a sablonok csomagokba lettek rendezve, melyeken belül találhatók továbbá a követelményfájlok, amelyek leírják, hogy milyen követelményeknek felelhet meg az aktuális sabloncsomag. Ezen követelmények mindegyike opcionális, így lehetőséget ad arra is hogy a nem kívánt funkciókat ne vigye bele a rendszerbe.

A program biztosít lehetőséget saját sabloncsomagok létrehozására és a meglévők módosítására is.

A kódgenerátorok nagy hátránya, hogy az absztrakt megfogalmazás már nem módosítható anélkül, hogy a kódot újra kelljen generálni. A szakdolgozat keretében elkészített alkalmazás ezen nem változtat, viszont ad egy megoldást erre a problémára a git verziókezelő rendszer használatával.

A program segítségével lehetőség nyílik arra, hogy ne csak egyszerű, hanem komplex szerver- és ezzel együtt kliens oldali kódokat tudjunk generálni, ezzel utat nyitva a jól strukturált, könnyen módosítható, webes alkalmazások felé.

Kulcsszavak: *scaffolding, web, kódgenerátor, rapid-prototyping, ER, modellező*

ABSTRACT

This thesis is for creating a web-based ER modeller and the right tool for fast-prototyping based development for websites at first.

The most of the web-pages share common properties that make them possible to be generated with just a little information. By this idea, the software introduced in this thesis is a code generator, which uses a modified entity-relationship model for the definition of generable resources. For this model, the application provides a designer interface in order to ease the use of the generator.

Respecting the readability the templates had been organized into packages. In packages, requirement files can be found, which shows, which requirements are suitable for the template package. All requirements are optional, so it gives the ability to skip unnecessary functionalities from the new system. In order to convince the users, the generator supports adding own template packages and modifying the existing ones.

So far, the code generators share a common disadvantage: The abstract definition can not be changed without re-generating the entire code. The software introduced in the thesis does not make it different but shows a solution using the Git version control system.

By the help of this application, opportunities open up for generating not just simple but complex server- and client side codes, with opening gates to the well-structured, easily modifiable web applications.

Keywords: *scaffolding, web, code, generator, rapid-prototyping, ER, modeller*

Tartalomjegyzék

1. Motiváció 1
   1.1. Egy sikertelen megrendelés története 1
   1.2. Miért ne használjunk nagy komponenseket, mint kész tartalomkezelő rendszereket 2
   1.3. Miért jó a “Rapid Prototyping” 3
   1.4. A megfelelő eszköz a Rapid Prototyping fejlesztési módszertan segítésére, weboldalakhoz 4
   1.5. Hasonló technológiák 6
      1.5.1. QuickAdminPanel a Laravel keretrendszerhez . . 6
      1.5.2. A Phoenix keretrendszer beépített generátora . . . 7
2. Rendszerterv 8
   2.1. Tervező 8
      2.1.1. Az Entity Relationship modell 8
      2.1.2. Létező Entity Relationship modell tervezők . . . 10
      2.1.3. Egy jobb tervező terve 11
      2.1.4. A Modell kiegészítései 12
   2.2. Generátor 13
      2.2.1. Példa egy egyszerű erőforrás kezelőre 13
      2.2.2. Lehetséges jelentések különböző modellekre . . . 16
      2.2.3. Módosíthatóság 19
      2.2.4. A nagy sablonválaszték elérése 22
   2.3. Modulok 23
      2.3.1. Irányelvek, konvenciók 23
      2.3.2. A rendszer fontosabb elemei 26
3. Felhasznált technológiák 27

3.1. Technológia kiválasztása az eszköz elkészítéséhez 27
3.2. Felhasznált saját külső könyvtárak 29
3.2.1. Bevezetés 29
3.2.2. Vecjs - Két dimenziós vektor számításokért felelős könyvtár 30
3.2.3. React-konva-anchors - Dinamikus pozícionálást és méretezést segítő könyvtár a react-konva számára 31
3.2.4. FastEJS - EJS-hez hasonló sablonmotor 34
4. Sablonok megvalósítása 35
4.1. Bevezetés 35
4.2. Nevek formázása 36
4.3. Sablon csomagok 37
4.4. Követelményfájlok 38
4.5. Sablonok 39
5. A Tervezőfelület megvalósítása 42
6. Hozzájárulás a fejlesztéshez 45

1. Motiváció

1.1. Egy sikertelen megrendelés története

Egy webáruház készítésére kaptam megrendelést. A követelmények nem voltak teljesen ismertek, és a legtöbb csak a fejlesztés során derült ki. Annak érdekében, hogy a weboldal gyorsan elkészüljön, komponens alapú fejlesztést alkalmaztam, azaz az első lépés a megfelelő komponens kiválasztása volt.

Mivel a webáruházak körében elterjedt a WooCommerce, ezért a döntésem is amellett állt. A WooCommerce egy plugin a Wordpress tartalomkezelő rendszerhez, amely az eredetileg blogokhoz szánt rendszert kiegészíti úgy, hogy webáruház lehessen az amúgy általános célú blogrendszerből.

A fejlesztés első szakaszában még látszólag előnyös volt. Az ügyfél hamar láthatott egy működő webáruházat, azonban a később érkező specifikációk miatt rá kellett jönnöm, hogy nem ez volt a legjobb választás.

Az ügyfél a legkisebb dolgokat is megszerette volna változtatni, azonban a rendszer ezekre nem adott lehetőséget. Ekkor váltani már költséges volt, ezért kénytelen voltam módosítani a rendszer alapvető funkcióit, ellehetetlenítve így a frissítés lehetőségét. A fejlesztést az is megnehezítette, hogy a kód azon részeihez nem volt dokumentáció, így egy ilyen módosítás akár nem megfelelő működést is eredményezhet. Így is lett. Az így beletett funkciók valamikor működtek, valamikor nem. Emellett a nem megfelelő architektúra miatt sebességbeli komplikációk léptek fel. Hosszas látszólag sehova nem haladó egyezkedések miatt, amikor már a projekt nagyon csődösnek lett ítélve, elkerülhetetlen volt a vége.

Egy lehetséges megoldás lett volna a megfelelő komponens kiválasztása a projekt kezdeti stádiumában. Azonban ez a választás nem tud alapulni megfontolt döntésen anélkül, hogy ismernénk a pontos követelményeket. Egy másik lehetséges megoldás lehet az alkalmazás fejlesztése kész Content Management System (CMS) nélkül. Ez a megoldás ad a lehető legjobban strukturált szoftvert az adott funkcionális követelményekre. A jó struktúra miatt feltehetően sebességre is jobban optimalizáltabb, azonban egyben ez a legdrágább is.

1.2. Miért ne használjunk nagy komponenseket, mint kész tartalomkezelő rendszereket

A komponens alapú fejlesztés a programozási módszertanok egyike. A lényege, hogy a program nagy részét kész komponensekből építjük fel, aztán a komponenseket megpróbáljuk minél jobban az igényekre szabni, ha ezeket nem tudja a komponens teljesíteni akkor meg lehet próbálni a követelmények módosítását, amennyiben az ügyfél elég rugalmas erre. Ha nincs megfelelő komponens azt le kell fejleszteni. [7]

A nagy komponensek előnye, hogy költségkímélőbb, mint ugyanazt a funkcionalitást újból megvalósítani.

Hátrányait minden komponensre egyedileg lehet megfogalmazni, de általánosságban legtöbb komponens dokumentációja csekély, a képességeik nagyon limitáltak, és továbbfejlesztésre nem adnak elég lehetőséget. Előfordul, hogy túl sok felesleges funkciót is tartalmaznak. Ebben az esetben sokszor az automatizáltság miatt nehéz az egyedi funkciók hozzáfejlesztése.

Az egyedi funkciók hozzáfejlesztése fontos, ugyanis az üzleti követelmények gyorsan változhatnak olyan irányba, ami egy nem előrelátható limitáltságot hoz elő a komponenseknél, ezért célszerű nem nagy komponenseket választani, hogy a cseréjük ne legyen költséges. Nagy komponensnek számít egy tartalom kezelő rendszer, mint például a WordPress. A tartalom kezelő rendszerek helyett, lehet használni keretrendszereket, amikhez csak kisebb komponensek érhetőek el, és közülük kevés biztosít felesleges funkcionalitást, valamint a cseréjük is egyszerűbb. [2]

1.3. Miért jó a “Rapid Prototyping”

Az ügyféltől nem szabad többet várni, minthogy megmondja, hogy jó -e így a program vagy nem. A Rapid Prototyping egy olyan programozási módszertan, aminek a lényege, hogy egy gyors prototípus segítségével a nem ismert követelményeket felderítsük. A gyors prototípus készítésére megfelelő eszközök szükségesek, ilyenek lehetnek például kódgenerátorok, terminálos kis eszközök vagy akár tervező programok. [6]

A Rapid Prototyping programozási módszer fejlesztő barát mert a programozó a fejlesztésre fókuszál és nem a komponensek integrációjára, másrészt segít a fejlesztőnek is a megfelelő project struktúra kialakításában, hiszen a prototípus még jelentősen módosítható.

A módszer egy jól strukturált, könnyen továbbfejleszthető minimális programot eredményez, ami teljesíti a szükséges követelményeket. A bővíthetősége jól illeszkedik a változó üzleti igényekre.

Habár a módszer előnyeit tekintve nagyon kedvező lehet, legtöbbször mégis inkább az aktuális igényeknek megfelelő, kész rendszereket integrálnak egy új fejlesztése helyett. Ez egyrészt azért lehet, mert egy új rendszer fejlesztése látszólag költségesebb, és az első értékelhető eredményt jóval később kapja meg az ügyfél. Weboldalak és azok hasonlóságaik miatt, ez a folyamat könnyen felgyorsítható, ha van rá megfelelő eszköz.

1.4. A megfelelő eszköz a Rapid Prototyping fejlesztési módszertan segítésére, weboldalakhoz

Kódgenerátorok alkalmazása jelentősen tudja növelni a fejlesztési sebességet. Anélkül, hogy a komponensek integrációjával törődnénk, a generált kód már tartalmazhat integrált apró komponenseket és egyben példakódot is a használatukra, ha esetleg másként szeretnénk használni. A megfelelő eszköz egy olyan kódgenerátor, ami már egyben stabil és felhasználásra kész programkódot generál, de megengedő annyira, hogy kikapcsolhatóak a nem kívánt funkciók generálása, a kívántak megtartásával.

A weboldalak nagy része csupán csak egy tartalom kezelő rendszer. Gondoljunk csak egy egyszerű blog rendszerre, ahol van 1 oldal a bejegyzések listázására, és kell lennie egy admin felületnek, ahol ezeket a bejegyzéseket menedzselik. Ezek az oldalak általában a tartalmat egyszerű űrlapokkal kezelik (pl: hozzáadás, szerkesztés, törlés). Ezek az űrlapok építik fel az egész webes alkalmazást, az adatbázis rétegtől a kinézetig, de teli vannak olyan közös elemekkel, amelyek lehetővé teszik, hogy az űrlapok, kevés információval is generálhatóak legyenek. Ezekhez az oldalakhoz általában elég egy erőforrás név (pl: bejegyzés), az erőforrás attribútumai (pl: a bejegyzés neve) és a típusa (pl: szöveg).

A legegyszerűbb tartalom kezelő rendszer ennyi adatból már generálható, azonban a komplexebb weboldalaknál több erőforrás is van, és ezek kapcsolódnak is egymáshoz (pl: a felhasználónak van több bejegyzése). Ha belevesszük a kapcsolatokat is az absztrakcióba, akkor a lehetőségek tárháza nyílik meg. Ha az “egy a sokhoz” (egy egyedhez több egyed tartozik) kapcsolatot nézzük, akkor létrejöhetnek olyan oldalak, amik ezt a kapcsolatot felhasználják:

- csak a felhasználó bejegyzéseinek a megjelenítése
- felhasználók és bejegyzések pároztatása
- bejegyzés hozzáadása (felhasználóként)

Ha létezne egy olyan kód generátor, ami ezen absztrakciónak a lehetséges kimeneteit legenerálja, akkor már szinte csak a frontend -el kellene foglalkozni. Frontend -nek hívjuk az applikáció azon részét, ami a felhasználónak látható felület megjelenítéséért felelős.

Az egyedi elemeket, mint például a kinézetet nem tudja kiváltani, de példakódot biztosíthat hozzá.

Egy kódgenerátornál fontos az is, hogy gyorsan el tudjon indulni, és az absztrakció megfogalmazását egyből megkezdhessük. Webfejlesztés során egy terminál kulcsfontosságú elem, és általában nyitva van a különböző eszközök miatt (pl: Node Package Manager (NPM)). A megnyitott terminál általában az aktuális projekt könyvtárába van lépve, ami kiváló lehetőséget ad arra, hogy innen indítsuk el az ezközt. Egy jó kódgenerátornak érdemes kezelnie egy bemenő paramétert arra a célra, hogy jelezze, hogy hol induljon el (pl: a Visual Studio Code biztosít lehetőséget arra, hogy könnyen elindítsuk az aktuális mappában: ’code .’)

Fontos, hogy az absztrakt megfogalmazásból minél több kódot lehessen generálni, ezzel levéve a terhet a programozók válláról. Az absztrakciónak a félúton kell lennie a specifikus és általános között, hogy sokatmondó legyen a modell, de egyben egyszerű is.

1.5. Hasonló technológiák

1.5.1. QuickAdminPanel a Laravel keretrendszerhez

A Laravel egy PHP alapú keretrendszer, ami MVC (Model View Controller) tervezési mintát követ, ezzel lehetőséget adva dinamikus weboldalak jól strukturált elkészítéséhez.

A QuickAdminPanel egy admin panel generátor a Laravel keretrendszerhez. A generátor internetes, és lokálisan nem elérhető. A generált állományokat le kell tölteni, és úgy integrálni a rendszerbe. A generátor az adatbázis rétegről a nézet rétegig mindent generál, így biztosítva a lehetőséget arra, hogy egy erőforrást lehessen listázni, hozzáadni, szerkeszteni és törölni. Kezeli az erőforrások kapcsolatát is, így akár komplexebb oldalakat is képes generálni.

Hátránya, hogy nem elég átlátható a felület. Új erőforrás létrehozása közben nem látszódnak a hozzá kapcsolható erőforrások. Bizonyos részei fizetősek, és nem biztosít elég segítséget az integrációra. A zártsága miatt nem kiegészíthető, és csak a Laravel keretrendszerhez érhető el.

1.5.2. A Phoenix keretrendszer beépített generátora

A Phoenix keretrendszer az egy elxir nyelven írt, a laravelhez hasonló MVC framework. Az elixir egy erlang alapú funkcionális nyelv.

A phoenix biztosít egy CLI eszközt a keretrendszer egyes lehetőségeinek automatizálására, így egy egyszerű generátort is. A generátor csak erőforrás listázást, szerkesztést, hozzáadást, törlést, valamint egy külön oldalt biztosít egy erőforrásnak. Kapcsolatokat támogatja, de a generált felületen a külsőkulcsokat csupán számként kezeli. A használata elég egyszerű:

```
mix phx.gen.html Accounts User users name:string age:integer
```

1. ábra. Példa a használatra

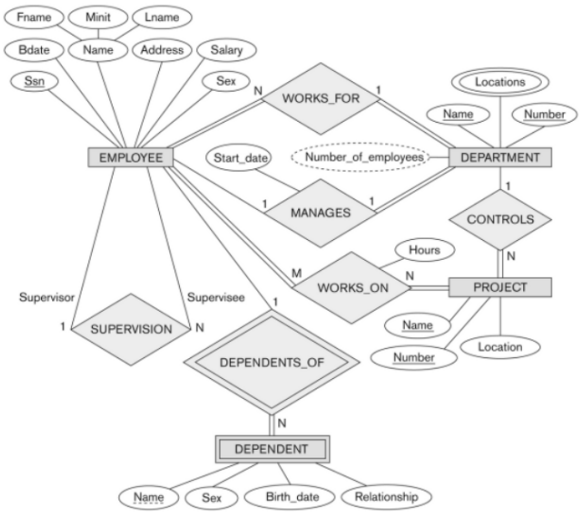
Hátránya, hogy egyszerre csak egy erőforrás generálására alkalmas, és keveset segít.

2. Rendszerterv

2.1. Tervező

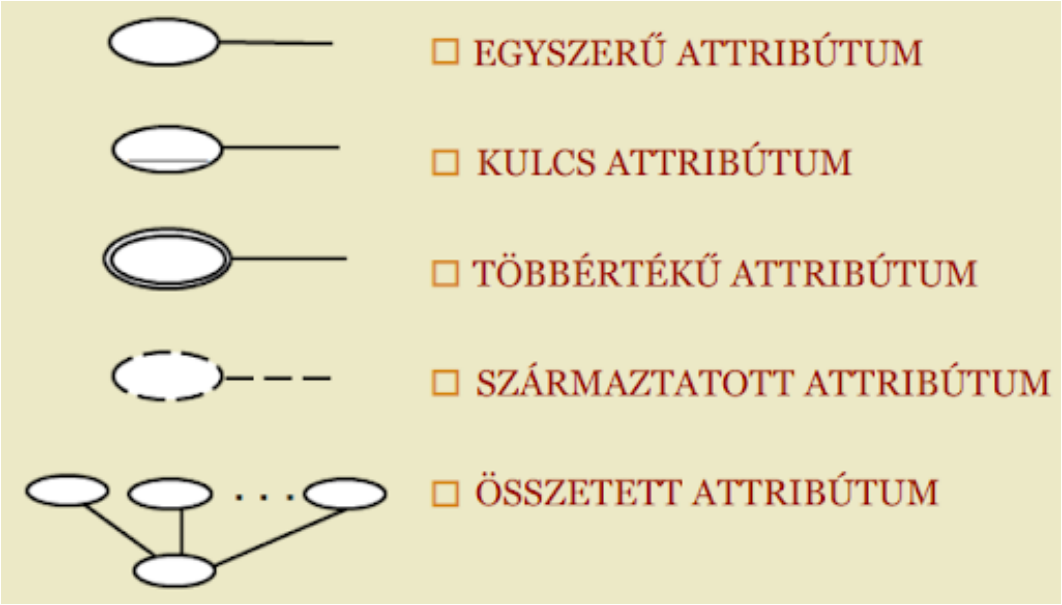
2.1.1. Az Entity Relationship modell

Egy kódgenerátornál fontos az is, hogy miképp adjuk át a szükséges információkat a projektről. Az egyik lehetőség a Command Line Interface (CLI) használata, ami gyors, de meg kell tanulni a használatát, valamint sok információ átadására nem alkalmas. Éppen ezért egy grafikus megfogalmazás előnyösebb lehet. Az előzőekben említett absztrakciós szintnek az Entity Relationship (ER) modell típusokkal kiegészített változata pont megfelel.



2. ábra. Példa az Entity Relationship modellre [3]

Az ER modell a nevéből is következtethetően entitásokat és azok kapcsolatait modellezi. Az entitás az 1db elem egy erőforrásból (pl: felhasználó a felhasználókból). A rombusz egy kapcsolatot jelöl két entitás között, itt 3 féle lehet: egy az egyhez (1db A entitáshoz maximum 1db B entitás tartozhat) egy a sokhoz (1db A entitáshoz több B entitás is tartozhat) sok a sokhoz (1db A entitáshoz több B entitás is tartozhat, és 1db B entitáshoz több A entitás is tartozhat Az ER modell különbséget tesz két fajta kapcsolat között: erős kapcsolat (a kapcsolat megléte kötelező) gyenge kapcsolat (a kapcsolat megléte opcionális) Minden entitásnak, és a kapcsolatnak lehetnek attribútumaik (pl: felhasználónak a felhasználónév). Az attribútumoknak a következő típusai vannak:



3. ábra. Az attribútumok típusai [1]

A modellt eredetileg arra tervezték, hogy akár az ügyfél által is validálható legyen a terv, így használata nagyon egyszerű, és jól dokumentálja a project szerkezetét. Hasonló absztrakció megfogalmazására képes az adatbázis logikai modell, ahol a kapcsolatok már kulcsokkal vannak ábrázolva, de az olvashatósága sokkal nehezebb, ami miatt az entity relationship modell a legmegfelelőbb a feladatra.

Eszköz	Egyed létrehozása	Attribútum hozzáadása	Összekapcsolás	Törlés
erdplus.com	2	4	3	2
draw.io	sok	sok	túl sok	2
gliffy.com	sok	sok	túl sok	2
sqldbm.com	3	sok	4	3
smartdraw.com	2	sok	túl sok	2

2.1.2. Létező Entity Relationship modell tervezők

Egy tervezőnél létfontosságú, hogy mindent gyorsan és egyszerűen meg lehessen benne tervezni. Az alábbi ábra az alapvető lépéseket hasonlítja össze különböző tervezőprogramoknál.

A mérésnél 1 lépésnek számított a kattintás, az elnevezés, a húzás, a billentyű lenyomása (ha nem szöveg írásra használtuk) és a szöveg írása. A mért adatok azóta változhattak.

Az összehasonlított rendszerek közül egyik sem volt felhasználóbarát. A sok panel miatt a diagramból kevés látszott. Az egyes elemek rendezése gondot okozott. Előfordult, hogy a kapcsolatok nem frissültek az egyedek mozgatása után.

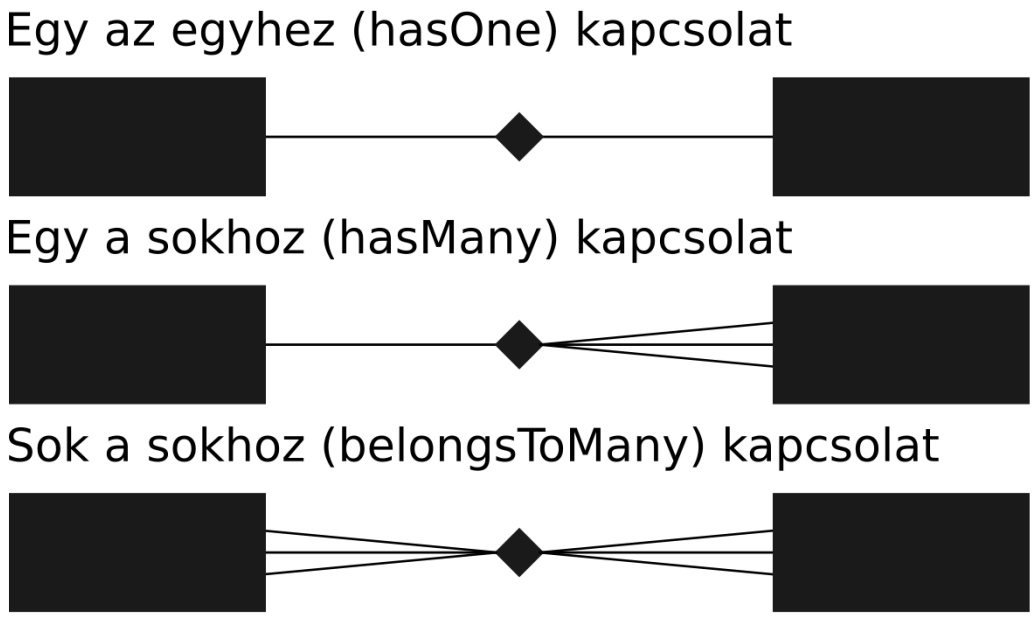
2.1.3. Egy jobb tervező terve

A legtöbb ER Modell tervező használhatatlan a kisebb képernyőkön, a sok képernyőterületet foglaló panelek miatt. Annak érdekében, hogy minél többet lássunk a modellből, a terv egy olyan ER modellező, ami majdnem teljesen panel mentes.

A mozgatás segítésére az attribútumok az entitások része, így mikor egy entitást mozgatunk, akkor mozgatjuk az attribútumait is. Az entitások közötti kapcsolat frissül, ha valamelyik entitás pozíciója a kapcsolatban megváltozik. Az objektumok elrendezése félig automatikus, ami azt jelenti, hogyha az elemek túl közel vannak egymáshoz, látszólag maguktól eltávolodnak.

A gyors modell tervezéséhez, minden felhasználói akció lehetőleg csak 1 lépést vesz igénybe.

A felhasználók meggyőzéséért egy modern, minimalista stílusú felülettel rendelkezik, jól látható elemekkel. A kapcsolatok leegyszerűsítése érdekében a kapcsolat 1..n, n..m, stb.. jelölések helyett grafikusan ábrázolja a kardinalitást, a következő módon:



4. ábra. Leegyszerűsített kapcsolatok

Ezen grafikus reprezentáció kényelmesebben olvasható, ugyanakkor, ha szükség van kötelező és opcionális kapcsolatok megfogalmazására akkor ugyanúgy jelölhetjük őket vastag, illetve vékony vonallal.

2.1.4. A Modell kiegészítései

Ahhoz, hogy a modellt használni lehessen webes alkalmazások generálására, szükséges pár módosítást elvégezni rajta. Az egyik ilyen módosítás az attribútumok típusokkal való kiegészítése (pl: name helyett name : string). A típusok szükségesek az űrlapok elkészítéséhez. A típusok adják meg azt, hogy milyen bemeneti mező szükséges az egyes attribútumok megadásához.

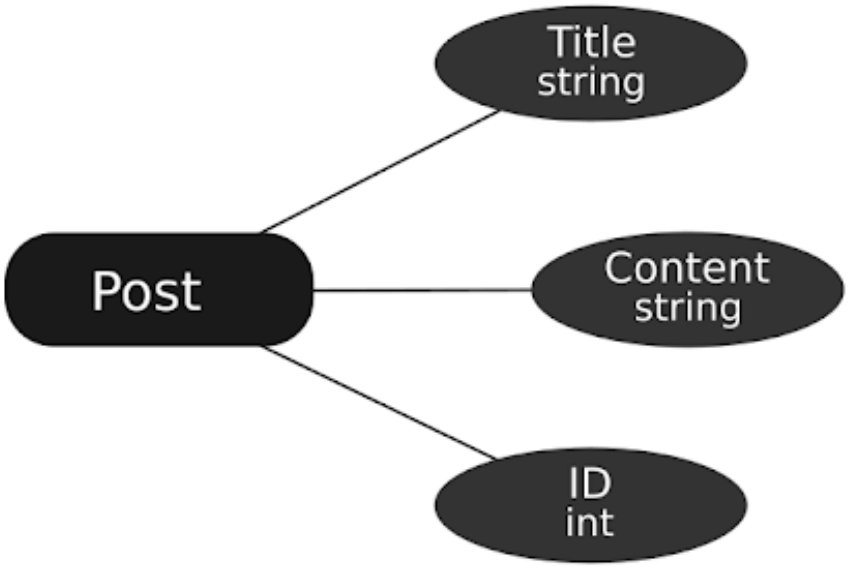
Egy kész webalkalmazás mindig foglalkozik a bemenet validálásával. A validáláshoz szükséges információkat a típus nevében szükséges lehet jelölni (pl: name : string[MIN=3, MAX= 15]).

Ha az entitások nevei az angol nyelvben megszámlálhatóak, akkor az adatbázis táblát az entitás nevének többes számú alakja alapján el lehet készíteni. Azonban, ha nem megszámlálhatóak, akkor biztosítani kell egy lehetőséget, hogy kézzel beírható legyen. Egyes Object Relation Mapping (ORM) rendszerek alapból az entitás többes számú alakját használják a Repository réteg neveként. A Repository réteg nevét fel lehet használni az adatbázistábla nevéhez is. Így az entitás nevéhez meg kell adni egy kiegészítő nevet is, ami a több ugyanolyan entitással foglalkozó absztrakcióra fog utalni. (Pl.: Music : MusicRepository)

2.2. Generátor

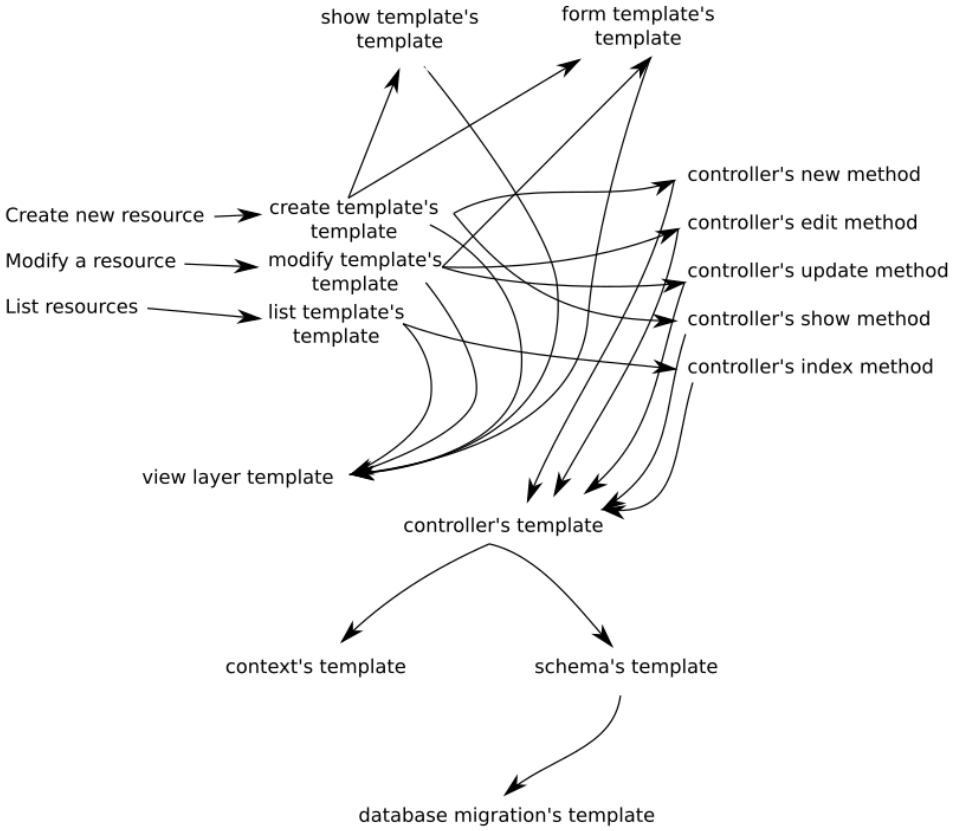
2.2.1. Példa egy egyszerű erőforrás kezelőre

Az egyszerűbb megértés érdekében, nézzünk egy példát egy erőforrás kezelésre. Legyen az erőforrás a bejegyzések egy blog rendszerben. Ebben az esetben szükség van egy Post nevű entitásra és pár alapvető attribútumra. Az egyik attribútum az ID, amivel azonosíthatjuk a bejegyzéseket, egy másik a Title, ami a bejegyzés neve, és egy harmadik a Content, ami a bejegyzés tartalma. Az ID típusa integer, a többi mind string. Ábrázolva a módosított ER modell segítségével, a következőképpen néz ki:



5. ábra. Egy bejegyzés modellje a kiegészített ER modellel ábrázolva

Ránézésre a következő oldalak jutnak eszünkbe a modell kapcsán: Bejegyzések listázása Bejegyzés hozzáadása Bejegyzés módosítása Ezek az oldalak jelentik a követelményeket a rendszerrel kapcsolatban. Ahhoz, hogy ezeket létrehozhassuk, szükséges egy olyan sabloncsomag, ami ezeknek a követelményeknek eleget tud tenni. A sablonoknak vannak függőségei, aminek célja, hogy a sablon funkcionalitásához szükséges egyéb sablonok is legenerálódhassanak. Egy sabloncsomagon belül vannak követelményfájlok, amik leírják a követelményeket, és egy kezdő sablont biztosít minden egyes követelményhez, ami egy belépési pont a függőségi gráfban.



6. ábra. Példa a függőségi gráfra

Az ábrán látható, hogy mennyi mindent kell létrehozni egy-egy entitás esetében. A gyakorlatban ez tovább bonyolódik. Opcionális követelmények esetében, egy-egy sablon nem feltétlen külön fájl, hanem egy meglévő fájl módosítását tartalmazzák. A fenti ábra is ábrázolja a controller több metódussal való kiegészítését. Gyakorlatban ez nem csak a controllernél jelent további bontásokat, hanem külön eset kell arra is, ha valaki nem szeretné listázni az egyes erőforrásokat, és emiatt új létrehozásakor ne ajánljon fel linket az összes mutatására, ugyanis az az oldal nem fog létezni.

A függőségi gráf nem feltétlen feszítőfa. Lehetnek körkörös függőségek, többszörös hivatkozások is. A program szempontjából ezt kezelni kell oly módon, hogy a sablonnak átadott bemeneti adatok és a fájlnév kombinációja alapján a sablon maximum egyszer generálódjon le.

A követelményfájlok megkapnak egy entitást, majd egy függvényben eldöntik, hogy arra az entitásra megvalósítható-e az aktuális követelmény. Ha megvalósítható a követelmény akkor az entitásból nyert adatok alapján (attribútumok, kapcsolatok),

átadják az első sablonnak a sablon bemeneti paramétereit.

A sablonok felépítésileg két részből kell, hogy álljanak: A sablon meta adatai A sablon által generálandó kód A meta adatok dinamikus adatok, vagyis a bemenő paraméterként megkapott adatokból készül. Ezek írják le, hogy a sablont hova, milyen néven, milyen jogosultsággal kell legenerálni, illetve, ha meglévő fájl módosítására használjuk, akkor melyik fájl melyik szekciójába generálja és milyen művelettel (szekció cseréje, szekció előtt, szekció után). A meta adatok tartalmazzák a sablon egyes függőségeit is.

A sablon egy egyszerű szöveg, kiegészítve változókkal vagy a használandó sablon motor által biztosított nyelvi elemekkel.

Ha egy sablonban megadott elérési út nem létezik, akkor a generátornak létre kell hoznia rekurzívan minden mappát és magát a fájlt is. Ha a megadott út módosítás céljából lett odaírva, de nincs olyan fájl, akkor a sabloncsomag függőségei rosszul lettek definiálva.

2.2.2. Lehetséges jelentések különböző modellekre

A struktúrában nagyon sok mindenre lehet következtetni. Ha csak a struktúrát nézzük akkor 6 jól elkülöníthető eset létezik:

- egy az egyhez kapcsolat
- egy a sokhoz kapcsolat
- sok a sokhoz kapcsolat
- Rekurzív egy az egyhez kapcsolat
- Rekurzív egy a sokhoz kapcsolat
- Rekurzív sok a sokhoz kapcsolat

Az egy az egyhez kapcsolat valamilyen adat kiegészítést jelent. Ha van pár attribútum, ami elég bonyolult ahhoz, hogy külön entitásként kezeljük, akkor erre az entitásra alkalmazhatjuk ezt a kapcsolattípust. Egy felhasználó lakcíme jó példával szolgál. A lakcím sok elemből áll, és lehet, hogy a kitöltése nem is kötelező. Ekkor a felhasználó entitást összekapcsolhatjuk egy “egy az egyhez” kapcsolattal a lakcím entitáshoz. (Az Entity Relationship modellben erre külön vannak összetett objektumok, gyakorlatilag a kettő megvalósítás között elhanyagolható különbség van.)

Az egy a sokhoz kapcsolat, a leggyakrabban használt kapcsolati forma. Jó példa erre egy blog rendszer, ahol a bejegyzések alá lehet írni megjegyzéseket. Ezt a kapcsolatot “van sok” (angolul: has many) kapcsolatnak szokás hívni, amiatt, mert az A entitásnak van sok B entitása. Ebben az esetben a bejegyzésnek van sok hozzászólása.

A sok a sokhoz kapcsolatra jó példa lehet a zenék és a stílusok kapcsolata. A modern zenéket már nehéz besorolni egy-egy stílusba, ugyanis több, mint egy igaz rájuk. Ugyanakkor, egy stílusra sem csak egy zene érvényes. Ilyenkor a zenéhez tartozik több stílus, a stílushoz pedig több zene. Elképzelhető egy olyan felület, ahol zenéket és stílusokat kell pároztatni. Talán gyakoribb példa, egy új zene beküldésekor a zenére jellemző stílusok kiválasztása.

A rekurzív kapcsolatoknak talán legritkább változata az egy az egyhez kapcsolat. Az ilyen kapcsolat egy láncolt listához hasonló kapcsolatot ír le. Ha egy adatbázis

tekintetében nézzük, és egy táblára gondolunk, akkor az ilyen tábla elkülöníthető listákat írna le. Habár használatára nehéz példát találni, a megléte nem okoz gondot, viszont szintén kibővíti a lehetséges felületeket.

A rekurzív “egy a sokhoz” kapcsolat egy fa szerkezetet ír le. A legjobb példa erre a részlegek felépítése. Egy nagy cégnél különböző részlegek lehetnek, akár elhelyezés szerint (Berlin, Veszprém, Bécs), de minden egyes részleghez tartozhat több kisebb részleg (fejlesztői, marketing, HR), és ezeken belül is még lehetnek apró részlegek. Ekkor elképzelhetőek különböző felületek:

- Új részleg hozzáadása egy meglévőhöz.
- Részlegek listázása fa szerkezetben.
- Részleg áthelyezése
- stb...

A rekurzív sok a sokhoz kapcsolat gyakorlatilag egy általános gráf szerkezet. Ilyen lehet a szociális hálózat ábrázolása is. Ez alapján a legjobb példa a szociális hálók.

- Kapcsolat hozzáadása.
- Kapcsolat törlése.
- Új felhasználó létrehozása kezdeti ismerősök kiválasztásával.
- stb...

Fontos megemlíteni, hogy a generátor nem korlátozódik relációs adatbázisokra. Minden implementáció a sablonokon múlik. Így, ha célszerű, a sablon használhat akár gráf adatbázisokat is.

Nem csak a struktúrából lehet következtetni az egyes legenerálandó részekre, hanem attribútumok típusaiból, neveiből, valamint entitások neveiből is. Például, ha egy entitás attribútumának a neve created_at és dátum típusú, akkor lehet feltételezni, hogy létrehozáskor ezt a mezőt az aktuális dátummal fel kell tölteni.

Egy másik jó példa a User entitás, aminek megléte jelenthet bejelentkezést, regisztrációt és egyéb legenerálandó részeket is.

Típusoknak bevihettünk nem megszokott típusokat is (pl.: image, audio, video). Ezekből szintén lehet következtetni még sok felületre (pl.: feltöltési lehetőség).

2.2.3. Módosíthatóság

A kódgenerátorok legnagyobb hátránya, hogy generálás után, az absztrakt definíció nem módosítható anélkül, hogy a kódot újra ne kellene generálni. Egy projekt elején nem tudhatjuk az összes követelményt, ezért nem mindent csak egy részét fogjuk legenerálni, amit utána módosítunk. Később ki fognak derülni más-más eltérő igények is, amit szintén generálni szeretnék, majd ismét módosítani.

Szerencsére a git verziókezelő erre a feladatra tökéletes. A git biztosít lehetőséget arra, hogy bizonyos funkcióknak a verzióit egy másik szálon vezéreljük. Ezeket a szálakat ágaknak (angolul: branch) hívják. Az ágakat egymásba lehet egyesíteni, ilyenkor annak az ágnak a módosításait amelyiket éppen beleegyesítünk az aktuális ágba, összefésüli, és az újakat beeszközöli. Ha az egyesítésnél komplikációk lépnek fel, akkor ezeket egyesével meg kell oldani.

A git meglétét nyugodtan lehet feltételezni, hiszen csekély számú kivételtől eltekintve minden fejlesztő, minden projectet már a git segítségével verzió-kezel.

A módszer a módosítás lehetőségére a következő:

A generátornak megadott generálási helyen feltételezzük, hogy a project már egy git repository -ként inicializálva lett. A generálás során, a generátor létrehoz egy másik ágat ERBranch néven.

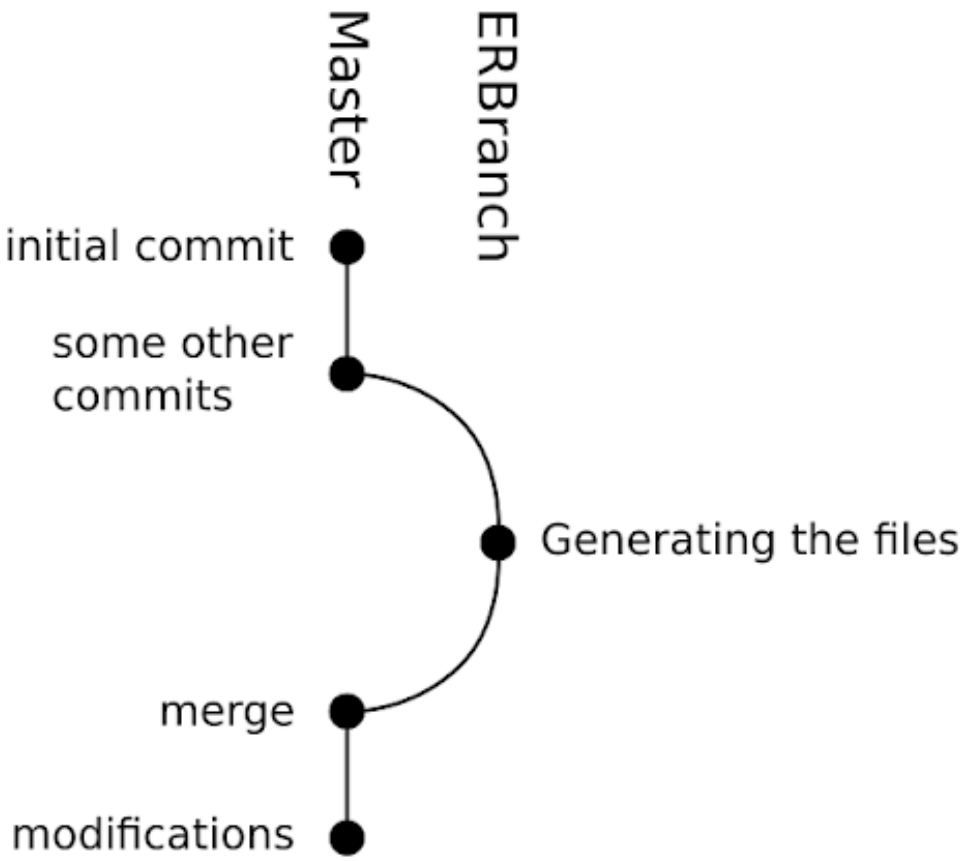
```
1  [generátor ]$ git checkout [-b] ERBranch
```

Ezt az ágat csak a generátor használhatja. Generáláskor a generátor erre az ágra hozza létre a fájlokat. Ahhoz, hogy ez egy visszaállítható állapot legyen, a változásokat elkövetjük.

```
1  [generátor ]$ git commit -a -m ‘Generation is ’succeded
```

A felhasználónak csak annyi dolga van, hogy visszamenjen arra az ágra, amelyikből az ERBranch ág leágazott, és egyesíti az ERBranch ág módosításaival.

```
1  [felhasználó ]$ git checkout master
2  [felhasználó ]$ git merge ERBranch
```



7. ábra. Git ágak az első generálás után

Az egyesítés közben felléphetnek konfliktusok (git conflict), amit az egyesítéshez meg kell oldani. Egyesítés után készen áll a program a módosításra.

A fejlesztés során fény derülhet arra, hogy a modellt ki kell még egészíteni. Nagyon fontos, hogy a lokális repository tiszta legyen, azaz minden hozzáadott módosítást véglegesíteni kell előtte:

```
1  [felhasználó ]$ git commit -a -m ‘some ’message
```

Ezt követően a generátort újra lehet alkalmazni. Az ágak ekkor az alábiak szerint fog alakulni:

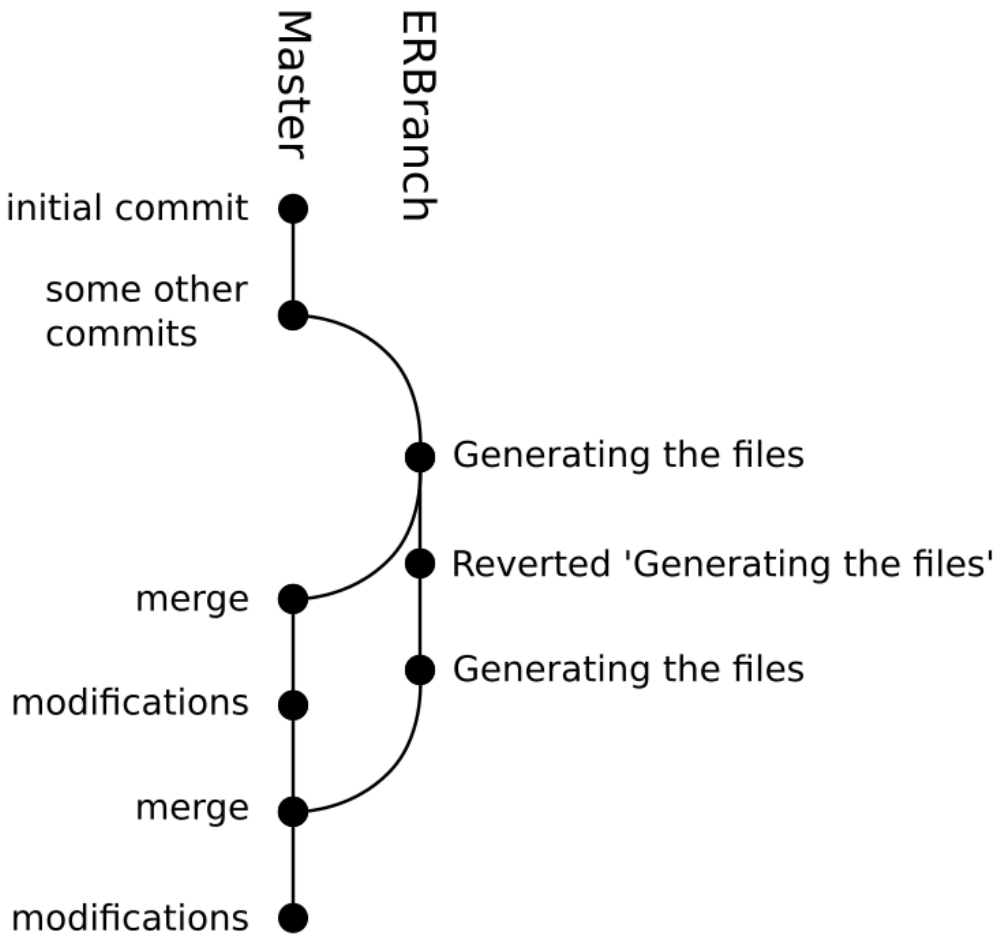
```
1  [generátor ]$ git checkout ERBranch
2  [generátor ]$ git revert THE_LAST_COMMIT
```

Ekkor a generátor újragenerálja az egész projectet, de a git szempontjából ez csak egy változásnak fog tűnni.

```
1  [generátor ]$ git commit -a -m ‘Generation is ’succeded
```

Ismét egyesítjük az előző branch -et az ERBranch -el.

```
1  [felhasználó ]$ git checkout master
2  [felhasználó ]$ git merge ERBranch
```



8. ábra. Git ágak a modell módosítása után

2.2.4. A nagy sablonválaszték elérése

A sablonok a generátor szerves részei. Ha nincsenek sablonok, akkor a generátor használhatatlan. Éppen ezért, az új sablon létrehozása a lehető legkevesebb tanulást igénybevevő megoldásnak kell lennie.

A dinamikusság és a kevés tanulás elérése érdekében a sablonoknak egy jól ismert programozási nyelven kell íródniuk. A stackoverflow 2018 -as felmérése szerint, a legismertebb programozási nyelv a JavaScript (forrás: https://insights.stackoverflow.com/survey). A JavaScript -et nem csak emiatt érdemes használni, hanem mert van hozzá interpeter, és a sablonokat build nélkül is lehet vele használni.

A JavaScript -hez léteznek meglévő sablon motorok, amelyekhez a fejlesztői környezetek biztosítanak szintaktikai megvilágítást.

2.3. Modulok

2.3.1. Irányelvek, konvenciók

Egy szoftver fejlesztése során nagyon fontos betartani bizonyos konvenciókat. A konvenciók segítik a projekt átláthatóságát, és megóvnak az egyes helytelen működésektől. [9]

Egy szoftvernek modulárisnak kell lennie, azaz a szoftvert különböző modulokra kell osztani. Minden modul egymástól független kell, hogy legyen, annak érdekében, hogy az egyes modulok bármikor cserélhetőek legyenek. A komponens cserélhetősége szintén emeli a programkód átláthatóságát és ezzel együtt a minőségét is. Egy üzlet igényei gyakran változhatnak, ezért egy nem modulárisan megírt program költségesebb lehet a későbbiekben. [5]

A funkcionális programozás jól ismert előnye az úgynevezett “immutablility”, ami az adattagok nem-módosíthatósága. Ha egy nagyobb projektnél erre nem figyelünk, akkor könnyen egy nem determinisztikusnak tűnő működést tapasztalhatunk az egyes fájlok szemszögéből. A nem determinisztikusnak tűnő kód szintén egy átláthatósági hiba, ezért figyelnünk kell rá. Teljesen változók nélkül nem mindig sikerül megvalósítani a szoftvert, de törekedni kell ezek használatának minimalizálására.

A Procedurális programozásban a függvények a paraméterükben várják az egyes függőségeit. A Dependency Injection ezt a szemléletet vetíti ki az objektum orientált programozásra. Az osztályok importálása helyett a konstruktor várja a működéshez szükséges függőségeket, így az osztály a nyelvi elemeken kívül csak a konstruktorban átadott elemeket használhatja, ezért a függőségek bármikor lecserélhetőek lesznek, és ezzel együtt a tesztelhetőség is egy újabb szintre lép, mivel a unit tesztek esetében csak arra kell figyelni, hogy hogyan használja a függőségeit. Gondoljunk bele, hogy egy osztály feladata a fájlrendszer módosítása. [4, 8] Ilyenkor az osztály meg fog kapni egy fájlrendszer kezelő objektumot. A unit teszt esetében megoldható, hogy egy olyan fájlrendszer kezelő objektumot adunk át, ami nem kezeli a fájlrendszert. Ezt a megoldást hívjuk a teszteknél mockup-nak.

A Dependency Injection azért injection mert általában egy automatikus rendszer figyeli, hogy a konstruktor milyen típusú függőségeket vár, és ezeket a függőségeket létrehozáskor be is injektálja. Sajnos az automatika nem minden nyelvel lehetséges,

de automatika nélkül is hasznos a konvenció. [10]

Az osztályoknak átadott függőségekből egy gráf szerkezetet lehet felírni. Ha az alkalmazást rétegesen építettük fel, akkor ez a gráf egy fa szerkezet lesz. A fa levelein lehetnek olyan osztályok, amiknek a működése egy állapottól függ (pl.: fájlrendszer), ezért tesztelni csak nehezen lehet. Ezeket az osztályokat érdemes minimális funkcionalitással létrehozni, annak érdekében, hogy kevesebbet kelljen tesztelni. A programozás során az egyik legnagyobb kihívás az állapotok kezelése, mert a sok feltétel miatt nem fog látszódni a kód funkcionalitása. A többi osztályt úgy érdemes megírni, hogy ne legyen állapotuk, azaz olyan legyen, mint egy jól strukturált függvény. Ideálisan, néhány kivételtől eltekintve egy constructor, egy publikus függvény és privát metódusokból áll.

Az ilyen esetekben az osztály elnevezése az osztály struktúrájából adódóan csak egy cselekvés és a tárgy neve lesz, amin a cselekvést végezzük. Például: TemplateRenderer. Ekkor a publikus metódust célszerű a cselekvésről elnevezni. Az előbbi példából kiindulva lehet: render.

A tesztelés nagyon fontos mielőtt a szoftvert kiadhatnánk, de a tesztek nem csak a fejlesztés végén kellenek, hanem a fejlesztés segítésére is. Vannak olyan helyzetek, amikor az egyes komponensek működését nem láthatjuk a szoftverben, mert szükségesek hozzá más komponensek is, vagy túl sok lépés lenne a működését előhozni. Erre a problémára találták ki a Test Based Development nevű fejlesztési módszert, ami azt jelenti, hogy a kódunkat egyből egy unit tesztel próbáljuk ki. A tesztek miatt a szoftver minősége nő, ugyanakkor a fejlesztési idő nem feltétlen több, a fentebb említettek miatt.

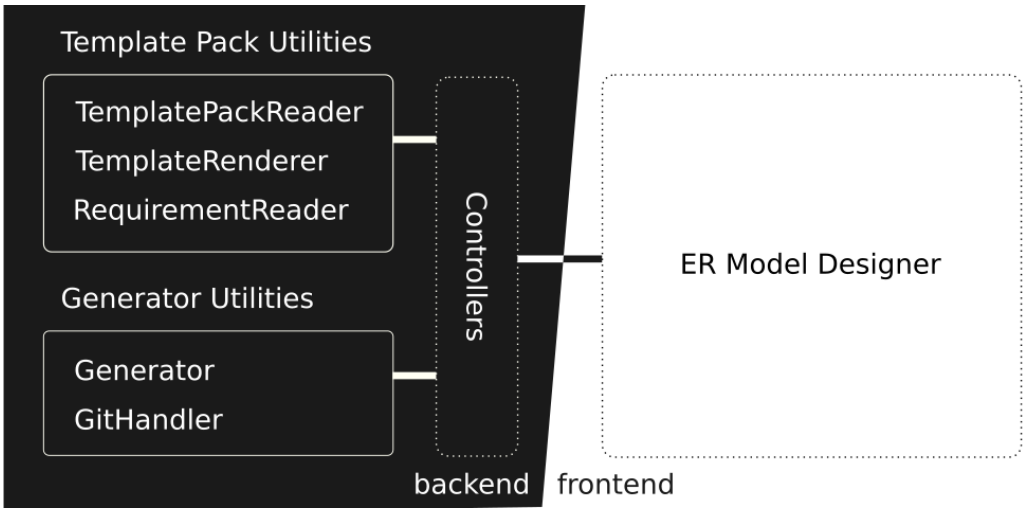
Fontosk, hogy ne írjuk meg többször ugyanazt, inkább legyen általánosítva, mint kétszerezve. Egyrészt kétszer megírni több idő, másrészt, ha később módosítani kell, akkor két helyen kell átírni.

A kód duplikálás elkerülésére jó módszer a réteges fejlesztés. A réteges fejlesztés segít elkülöníteni és csoportosítani a kód egyes funkcióit. A legismertebb réteges architektúra a Model View Controller (MVC). Fontos azonban felismerni, hogy szükség lehet további rétegekre is. A Model feladata az adatokkal kapcsolatos műveletek elvégzése, azonban szeretnénk elfüggetleníteni az üzleti logikát. Tekintsük például a következő logikát: Maximum az 5 perce íródott üzeneteket tekintsük új

üzenetnek. A Controller szintén nem jó réteg erre, mert többször szeretnénk használni az új üzeneteket. Ilyenkor érdemes bevezetni egy másik réteget is a Model és a Controller közé, ami ezeket lekezeli. Általában ezt a réteget hívják Repository rétegnek. Ugyanígy eljárhatunk a többi réteg esetében is.

Az egyedüli igazság forrás (Single source of truth) konvenció, lényege, hogy ne duplikáljunk adatot. Az adatok elérését a lehető legjobban csoportosítjuk, így elkerüljük a konzisztencia fenttartására irányuló problémákat.

2.3.2. A rendszer fontosabb elemei



9. ábra. A rendszer fontosabb elemei

Az előző irányelvek szerint a rendszer fontosabb elemei a fenti ábrán láthatóak. Az ábra két részből áll. A backend és a frontend. A frontend a felhasználói felületet biztosító része a programnak, a backend pedig a felhasználói felülettől jövő kérések feldolgozásáért felelős. A kérést elsőnek egy Router kapja meg, ami a megfelelő Controller -hez irányítja azt. A Controller megvalósítja a kérés feldolgozását az erre specializálódott osztályok segítségével, és visszatér egy válasszal, ami megjelenik a felhasználói felületen.

3. Felhasznált technológiák

3.1. Technológia kiválasztása az eszköz elkészítéséhez

A web alapú alkalmazások minden nagyobb platformon elérhetőek, kompatibilitási problémák nélkül. A web a felhasználói felületre számos lehetőséget kínál, és ezek mindegyikét egyszerűen el lehet érni. Köszönhetően a Konva.js könyvtárnak már a HTMLCanvasElement -re való rajzolás is leegyszerűsödött, így a ER modell rajzolására is megfelelő választás. Nem meglepő tehát, hogy a felhasználó felület webes felületen készült.

A felhasználó felület programozása olyan problémákat vet fel, amikre nem érdemes saját megoldást adni, habár az egyedi implementáció általában gyorsabb programot eredményez. Ezeket a problémákat küszöbölik ki a frontend keretrendszerek. A frontend keretrendszerek közül a legelterjedtebb a facebook által kifejlesztett React. A React a nevéből adódóan reaktív keretrendszer, azaz, ha frissül az állapot, frissül a nézet is automatikusan. A React szemlélete, hogy minden egyes elemnek külön komponenst hozzunk létre. A komponensek tartalmazhatnak több komponenst is. Minden komponensnek lehet egy saját állapota és több paramétere, így látszólag egy react -es komponens ugyanolyan, mint egy HTML DOM elem, így a HTML kontextusában kényelmesen használni.

A HTML DOM egy fa szerkezet. A legtöbb frontend keretrendszer ezt a szerkezetet használja az állapot változás utáni frissítéshez. A DOM strukturálisan nem tud tartalmazni általánosabb gráfot, mint például az ER modellt. Így az speciális megoldást igényel. A Vue és az Angular túl sok mindent rejt el a tényleges implementációból, így azokban ezt a speciális igényt beeszközölni nehéz lett volna. Mivel a React rejt el a legkevesebbet, így az a megfelelő keretrendszer erre a problémakörre.

Az alkalmazás kezelni fogja a fájlrendszert és a git-et, így teljesen web alapú nem lehet. Ahhoz, hogy a frontend -et megjeleníthessük asztali alkalmazásként, szükség van az Electron -ra. Az Electron egy Nodejs csomag, ami tartalmazza a Chromium böngésző egy részét, és gyakorlatilag ez adja a megjelenést. Az Electron miatt szükség van Nodejs -re, így a backend logikája is JavaScript lesz.

A közös nyelv előnye, hogy bizonyos osztályokat egyaránt használhatunk a fel-

használói felületen és a backend -en is. Így egy közös modell réteggel, biztosítani lehet kommunikációs interfészeket a két réteg között.

Mindkét réteg az ECMAScript 2018 -as szabvánnyal íródott. Ezt a szabványt nem támogatja sem a Nodejs, sem a Chromium, ezért szükséges egy fordítóra, ami egyszerű JavaScriptre fordítja a kódot. A fordítást egy még nem annyira elterjedt fordító végzi: A Sucrase. A Sucrase az elterjedt Babel fordítónak egy gyorsabb és egyszerűbb alternatívája. A React esetében célszerű a JSX szabvány is, amit szintén a Sucrase fordít javascriptre. A JSX lehetővé teszi, hogy a react -es komponenseket írhassuk html elemként.

3.2. Felhasznált saját külső könyvtárak

3.2.1. Bevezetés

A fejlesztés során sok olyan könyvtárat meg kellett írni, ami eddig nem volt még elérhető, de érdemes lehet felhasználni egy másik projektben is. Mivel Nodejs - ben készült a szoftver, a Node Package Manager segítségével publikálhatóak saját könyvtárak is.

3.2.2. Vecjs - Két dimenziós vektor számításokért felelős könyvtár

A felhasználói felület vektorgrafikai számításaihoz szükséges egy gyors két dimenziós függvény könyvtár. A könyvtár vecjs néven érhető el (https://www.npmjs.com/package/vecjs). A vecjs a leggyakoribb vektor műveleteket tartalmazza funkcionális paradigmákkal. Az eddig is meglévő vektor műveleteket segítő könyvtárak nem voltak alkalmasak átlátható kód írására. A legtöbb mutable volt, ami azt jelenti, hogy a művelet módosította a változót, és nem újat adott vissza. Pl.:

```
1  a = new Vec2D(10,20)
2  b = new Vec2D(30,40)
3  a.add(b)
```

Ekkor az ‘a‘értéke: Vec2D(40,60). A probléma ezzel az, hogy az ‘a‘eredeti értékét nem érjük el a hozzáadást követően. Matematikai problémáknál jobb megoldás, ha a változók nem módosulnak. Pl.:

```
1  a = new Vec2D(10,20)
2  b = new Vec2D(30,40)
3  c = a.add(b)
```

Ebben az esetben a c kapja meg az ‘a‘+‘b‘értékét, de itt az ‘add‘függvény nem módosítja a bemenet értékét. Az implementáció lehetővé teszi, hogy a megszokott infix jelölésként alkalmazzuk az összeadást. Pl.:

```
1  (a) .add (b) .add (c)
```

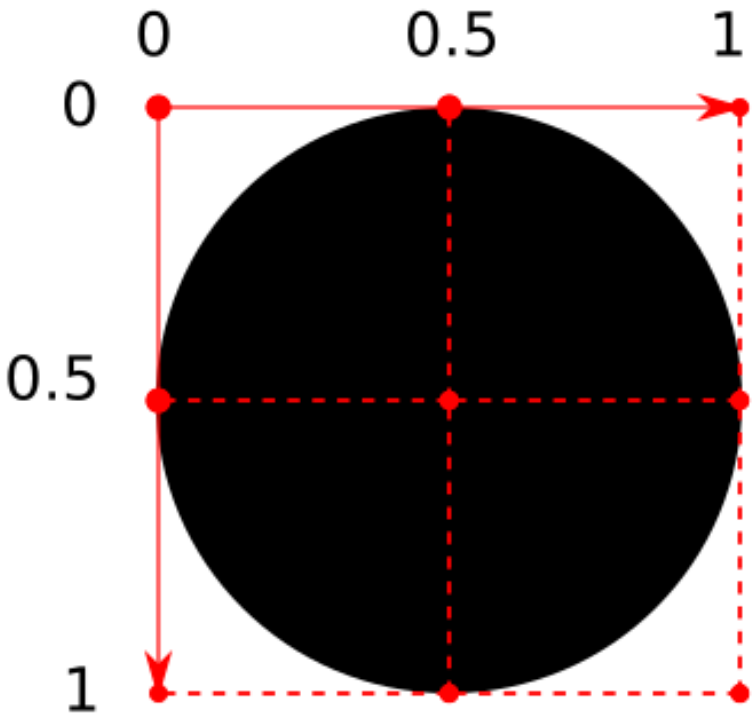
3.2.3. React-konva-anchors - Dinamikus pozícionálást és méretezést segítő könyvtár a react-konva számára

A konva egy canvas -re rajzolást leegyszerűsítő könyvtár, amit ha nagy projektnél használunk, akkor érdemes valamilyen frontend keretrendszerrel egybekötni. A kettő könyvtár közötti kapcsolatot - egy másik - a react-konva biztosítja. A react-konva-anchors segítségével definiálhatunk elemek közti relatív elhelyezést és méretezést, ami frissül, ha a referencia elem frissül.

A cél az volt, hogy a CSS -hez hasonló logikával lehessen megoldani a pozícionálást, de az eredmény egy sokkal általánosabb megoldás lett. Tegyük fel van két elemünk. Az egyik egy téglalap, aminek az origója a téglalap bal felső sarkában van. A másik egy kör, aminek az origója a kör közepén van. Legyen a cél az, hogy a kört a téglalap jobb felső részére szeretnénk tenni. Szeretnénk azt is elérni, hogyha változik a téglalap szélessége akkor a kör pozíciója is frissüljön. A megoldáshoz szükség van megadni a mozgatni kívánt elem origóját relatívan. A relatív origó a következőképpen számolódik:

Így tehát az (0.5,0.5) lesz a kör origója. Az origót ezután átállíthatjuk, ha esetleg nem a közepe szerint szeretnénk mozgatni, de mivel nekünk jó a közepe, ezért marad a (0.5,0.5). A referencia elemnél hasonlóan meg kell adnia az elem eredeti origóját relatívan, ezután ott is átállíthatjuk a kívánt helyre. Jelen esetben a referencia origója (0,0). Mivel mi a jobb felső sarokba szeretnénk mozgatni a kört, ezért ezt az origót átállítjuk (1,0) -ra. A horgony úgy működik, hogy az elemet eltolja a referencia elemhez úgy, hogy a két elem origója megegyezzen, így megvalósítva a százalékos elhelyezést. Szükség lehet egy utólagos nem százalékos eltolásra is (pl.: (10px, -10px)). Erre az esetre itt egy példakód:

```
1  <PositionAnchor
2  element={() => ELEMENT_TO_MOVE}
3  elementOrigin={{x:0.5,y:0.5}}
4  elementDesiredOrigin={{x:0.5, y:0.5}}
5
6  reference={() => REFERENCE_ELEMENT}
7  referenceOrigin={{x:0,y:0}}
8  referenceDesiredOrigin={{x:1, y:0}}
```



10. ábra. Egy kör relatív kordinátarendszere

```
9
10  shift={{x:-25, y:0}}
11  change={(x, y) => this.setState({ POS: { x, y } })} />
```

A PositionAnchor-t lehet használni egy elem saját magának az eltolására is. Például, ha egy téglalap origója a (0,0) -ban van, a téglalap pozíciója szintén (0,0), de mi szeretnénk az origóját megváltoztatni (0.5,0.5) -re azért, hogy a koordinátarendszerben a (0,0) pont legyen a téglalap közepe. Erre azonban túl bonyolultnak tűnhet a PositionAnchor. Ezért van egy specifikusabb változata: a CenterAnchor.

```
1  <CenterAnchor
2  element={() => ELEMENT_TO_CENTER}
3  change={(x, y) => this.setState({ POS: { x, y } })} />
```

Mivel a PositionAnchor elég általános ahhoz, hogy mindenféle pozícionálást

meg lehessen vele oldani, és a CenterAnchor segítségével a középrehelyezés is leegyszerűsödött, ezért a pozícionáláshoz nem kell több horgony elem. A méret kezeléséhez viszont igen. Legsűrűbben a szélességet kell horgonyozni valamilyen referencia elem méretéhez. Például egy téglalap méretét a benne lévő szöveg méretéhez. Erre szolgál a WidthAnchor.

```
<WidthAnchor
reference={() => REF_ELEMENT}
element={() => ELEMENT_TO_RESIZE}
padding={25}
change={width => this.setState({EL_W: width })} />
```

A WidthAnchor a megadott referencia elem méretét leolvassa, és az elemet átméretezi úgy, hogy a padding értékét hozzáadja a mérethez. Minden esetet viszont lehetetlen lenne lefedni speciális horgonyokkal, ezért van a CustomAnchor, ami lehetőséget nyit saját megoldásokra is.

```
<CustomAnchor
reference={ () => EL_TO_WATCH }
element={ () => EL_TO_CHANGE }
result={ (ref) => ({ x: ref.width(), y:0 }) }
isUpdateNeeded={
  (el, res) =>
  el.x() !== res.x || el.y() !== res.y
  }
  change={(res)=> { this.setState({PROP: res }) } } />
```

A CustomAnchor esetében szükséges definiálni azt, hogy mikor frissítse az elemet, ezzel el lehet kerülni azt, hogy egy állandó frissítési ciklus alakuljon ki, amiatt, hogy egy horgony referencia eleme egy horgony módosítandó eleme. Meg kell adni a számítást, amit egy függvényben, kell megvalósítani.

3.2.4. FastEJS - EJS-hez hasonló sablonmotor

Az EJS egy ismert egyszerű sablonmotor a javascript nyelvhez. A FastEJS ennek egy gyorsabb és használhatóbb implementációja. A fő ok, amiért nem volt jó az EJS az, hogy EJS -hez hasonló sablonokat is szükséges lehet generálni, azaz a sablon sablonját is meg kell írni. Az EJS erre lehetőséget nem adott, vagy csak hibásan. Az implementáció egyszerű reguláris kifejezéseket tartalmaz, és egy eval kifejezést. A regexp összeállítja a sablonból a javascriptet, az eval (evaluate) pedig lefuttatja azt.

Mivel a javascript interpretált nyelv, ezért az egész sablon motort szabadon össze lehet állítani szövegesen. Ez lehetővé teszi, hogy az egyes regexp utasítások kikapcsolhatóak legyenek. Példa a motor lehetséges regexp utasításaira:

```
{
  "begin": ".replace(/\\n/g, '\\uffff')",
  "<%=": ".replace(/<%=([^%]+) %>/g, '`; __out += $1 + `')",
  "<%#": ".replace(/<%#([^%]+) %>/g, '')",
  "<% ": ".replace(/<%(\\s|\\uffff)/g, '`; ')",
  " %>": ".replace(/(\\s|\\uffff)%>/g, '; __out +=`')",
  "<%%": ".replace(/<%%/g, '<%')",
  "%%>": ".replace(/%%>/g, '%>')",
  "end": ".replace(/\\uffff/g, '\\n')"
}
```

Mivel ez egy javascript objektum, és a könyvtár engedi a publikus hozzáférést, ezért ezek az elemek az importálás során szabadon törölhetőek. Így akár a legegyszerűbb sablonmotorrá is átalakíthatjuk, ami jelentős sebesség növekedést eredményez. Ugyanakkor lehetőséget ad új kifejezés hozzáadására vagy kifejezés módosítására is. Egy lehetséges példa a kifejezés törlésére:

```
const FastEJS = require('fastejs')
delete FastEJS.tags['<% ']
delete FastEJS.tags['<%#']
delete FastEJS.tags['<%%']
delete FastEJS.tags['%%>']
delete FastEJS.tags[' %>']
FastEJS.parse(`<%= "hello world" %>`)
```

4. Sablonok megvalósítása

4.1. Bevezetés

A sablonok jelentik a lelkét a generátornak. Ha nincs sablon, nincs mit generálnia. A sablon megvalósításának nagyon egyszerű folyamatnak kell lennie. Ideálisan egy fájl sablonja annyira komplex, mint maga a fájl létrehozása. Így elérhető lenne, hogy közvetlen fejlesztés helyett sablonok fejlesztése legyen a cél. Gyakorlatilag ennek elérése nem lehetséges, mert a sablon az általánossága miatt mindig bonyolultabb lesz, de törekedhetünk arra, hogy a sablonok bonyolultsága minimális legyen.

4.2. Nevek formázása

A sablonok bonyolultságának elkerülésére használhatunk segédosztályokat. Ilyen segédosztály a Formatter is, ami a nevek formázását segíti. A programozásban elterjedtek bizonyos elnevezési konvenciók. PL: UpperCamelCase, lowerCamelCase, kebab-case, snake_case. Az elnevezési konvenciók segítik az osztályok, a változók és egyéb típusú nevek megkülönböztetését. A Formatter egy reguláris kifejezés segítségével alakítja át bármelyik formátumról a megadott formátumra.

Nem csak a kis- és nagybetű közti alakítás szükséges, hanem egyes- és többesszám közti átváltást is el kell végezni. Például egy bejegyzés entitásból generált listázás oldalnak a címe valószínűleg bejegyzések lesz. Ezzel szemben, a Formatter csak angol kifejezéseket tud átalakítani, ugyanis a programkódban nincs szükség más nyelvű kifejezésekre. Ha a program más nyelvű lesz, azt lokalizációval szokták megoldani. A több szavas kifejezéseknél, az átalakítás csak az utolsó szóra vonatkozik (PL: UserRight -ból lesz UserRights). Ezt az átalakítást a pluralize nevű NodeJS csomag végzi.

4.3. Sablon csomagok

Minden fejlesztőnek megvan a saját kódstílusa, emellett a beépített sablonok kiegészítése és felülírása kulcsfontosságú, ezért a program biztosít lehetőséget saját sablonok hozzáadására és a meglévők módosítására. Az úgynevezett “Single Source of Truth” konvenció alapján, a program csak a felhasználói sablonokat használja. A beépített sablonokat a program a felhasználói sablonok mappájába másolja generálás előtt, ha az a mappa addig nem létezett.

A felhasználói sablonok Linux
GNU illetve Mac típusú rendszereken a HOME környezeti változóban megadott mappát felhasználva a $HOME/.config/erscaffold-packs mappában érhetőek el.

A sablon csomagok rendelkeznek egyedi tulajdonságokkal, amelyek tárolásához biztosítani kell egy fájlt. Ez a fájl az index.pack.js, ami minden csomag számára kötelező. Ez írja le a csomag nevét és a benne található követelményeket.

```
1  // BASIC_PHOENIX/index.pack.js
2  const pack = {
3    name: 'Basic Phoenix',
4    requirements: [
5      {
6        file: 'resource_new.requirement.js'
7      },
8      // ...
9    ]
10 };
```

A kiolvasását a NodeJS végzi el. Első lépésként a program beolvassa a fájl tartalmát. Mivel a javascript interpretált nyelv, ezért ad lehetőséget arra is, hogy build nélkül le lehessen futtatni a beolvasott kódot. A beolvasott kódhoz hozzáilleszti a kiolvasni kívánt változó nevét, és ezzel együtt lefuttatva, megkaphatjuk annak a változónak az értékét. Ebben az esetben a pack változó lesz a kimeneten.

4.4. Követelményfájlok

Minden sabloncsomag megoldást nyújt bizonyos követelményekre (pl: lehessen erőforrást listázni). Egy követelményen belül lehetnek kisebb követelmények is (pl: legyen új hozzáadása gomb). Ezeket a követelményeket ún. követelményfájlokban tároljuk.

```
1  // BASIC_PHOENIX/resource_edit.requirement.js
2  const requirement = {
3    name: 'Edit resource',
4    template: './template/edit.template.ejs',
5    data(entity) {
6      return entity;
7    },
8    children: [{ file: 'resource_edit_go_back_button.requirement↩
         .js' }]
9  };
```

A követelményfájlok tartalmazzák a követelmény nevét, ami majd megjelenik a kezelőfelületen a követelmények kiválasztásakor. Ezen felül tartalmazniuk kell egy kezdő sablont, ami ezen követelmény teljesítésére szolgál.

Minden követelményfájl tartalmaz egy data nevű függvényt, ami megkap egy entitást a generáláskor. Ezzel a függvénnyel, a követelményfájl kiválaszthatja, hogy milyen adatokat kell megkapnia a beállított kezdősablonnak.

Ezen felül a children nevű tömb tárolja az ezen követelmény alá tartozó követelményeket.

Későbbi továbbfejlesztésekben a gyerek követelmények tárolása helyett, szülő követelmény megadása lesz a kötelező, ezzel modulárisabbá lehet tenni a rendszert. Ezzel a megoldással elérhető lesz, hogy egy sabloncsomag egy másik sabloncsomagot egészíthessen ki. Az, hogy milyen további követelményeket választhatunk ki az függhet az entitástól, ezért későbbiekben lesz egy függvény, ami a követelmény paramétereit fogja visszaadni egy bemeneti entitásból.

4.5. Sablonok

A sablonoknak két fajtája van. Az egyik típus az amelyik létrehoz fájlokat. A másik amelyik kiegészíti a már létrehozottakat.

```
1  <%
2  // BASIC_PHOENIX/view/view.template.ejs
3
4  let app_name = Formatter.snakeCase(APPNAME)
5  let AppName = Formatter.upperCamelCase(APPNAME)
6  let entity_name = Formatter.snakeCase(entity.name)
7  let EntityName = Formatter.upperCamelCase(entity.name)
8  let meta = {
9    creates: {
10     fileName: "lib/" + app_name + "_web/views/" + ↩
           entity_name + "_view.ex"
11   }
12 }
13
14 %>
15 defmodule <%= AppName %>Web.<%= EntityName %>View do
16 use <%= AppName %>Web, :view
17 end
```

Egy példa kimenet a fenti sablonra:

```
1  defmodule BlogWeb.PostView do
2  use BlogWeb, :view
3  end
```

Minden sablon tartalmaz egy meta nevű változót. A meta leírja a sablon generáláshoz szükséges adatokat (pl: a létrehozandó fájl elérési útja). Minden sablon a saját fejlesztésű FastEJS sablon motort használja, amivel könnyedén kilehet nyerni a rendereléskor felhasznált változókat, így a meta objektumot is.

A kiegészítő sablonoknál a meta értéke nem csak a fájl elhelyezkedését kell, hogy tartalmazza.

```
1  <%
2  // BASIC_PHOENIX/controller/action_show.template.ejs
3  let CtxName = Formatter.upperCamelCase(entity.context)
4  let app_name = Formatter.snakeCase(APPNAME)
5  let entity_name = Formatter.snakeCase(entity.name)
6
7  let meta = {
8    extends: {
9      fileName: "lib/"
10     + app_name
11     + "_web/controllers/"
12     + entity_name
13     + "_controller.ex",
14     section: "# methods",
15     place: "after"
16   },
17   depends_on: [
18     { template: "./controller.template.ejs", data: {entity: ↩
         entity} }
19   ]
20 }
21
22 %>
23 def show(conn, %{"id" => id}) do
24 <%= entity_name %> = <%= CtxName %>.get_<%= entity_name %>!(id↩
    )
25 render(conn, "show.html", <%= entity_name %>: <%= entity_name ↩
    %>)
26 end
```

A fenti példasablon egy controller kiegészítése egy show nevű metódussal. A meta tartalmában az extends nevű objektum hívja fel a figyelmét a generátornak, hogy ez a sablon, egy másik kiegészítése. Ezen belül meg kell mondani, hogy melyik sablonfájl által generált fájlt egészítse ki, és hogy hova tegye a generált kódot a fájlon belül. A pozíció kijelölésére két paraméter szolgál. A section és a place. A section lehet egy reguláris kifejezés, illetve egyszerű szöveg típusú is. Ez mondja meg, hogy a kiegészítendő fájlban milyen kifejezést keressen. A place megmondja, hogy a kiegészítés milyen elhelyezésű legyen a megtalált kifejezéshez képest.

Egy példa kimenet a fenti sablonra:

```
1  def show(conn, %{"id" => id}) do
2  post = posts.get_post!(id)
3  render(conn, "show.html", post: post)
4  end
```

A place értékének különböző fajtái lehetnek:

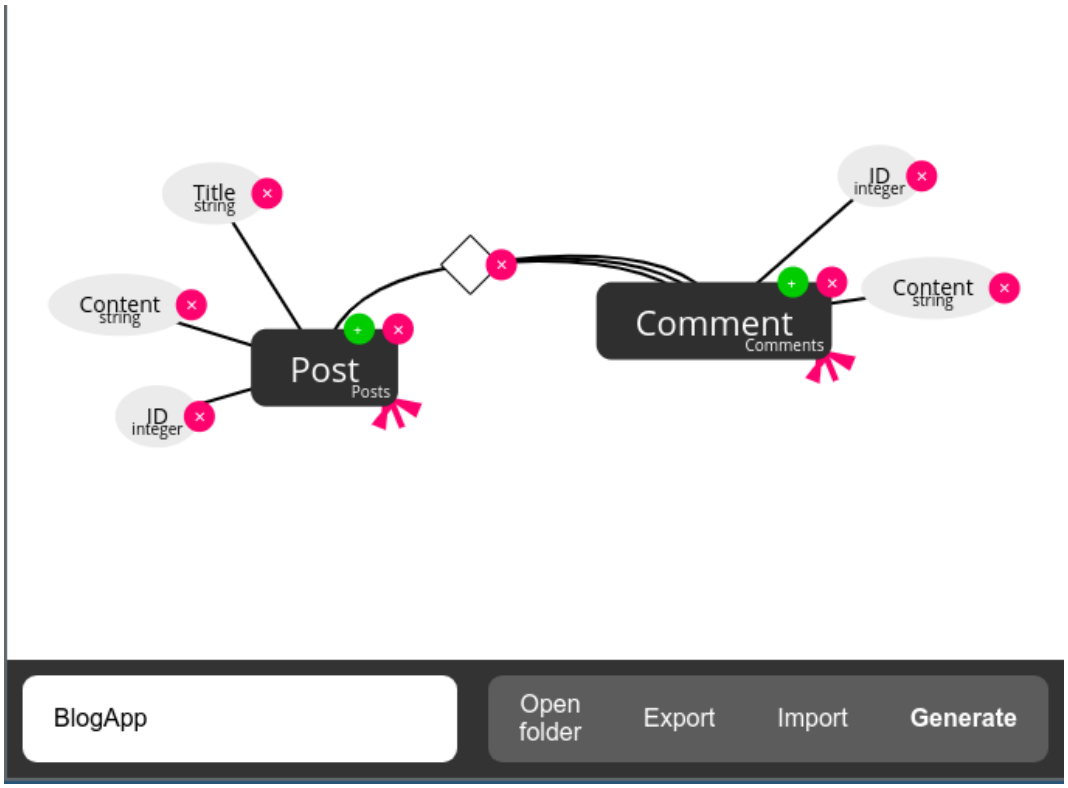
- “before”
- “after”
- “replace”

A “before” esetében a kiegészítés a megtalált kifejezés sora előtti sorba, az “after” esetében a megtalált kifejezés sora utáni sorba fogja helyezni a lerenderelt kiegészítő sablon tartalmát.

Lehetőség van a megtalált kifejezés cseréjére is a “replace” érték által. Ez esetben a megtalált kifejezés helyett a lerenderelt kiegészítő sablon tartalma fog bekerülni.

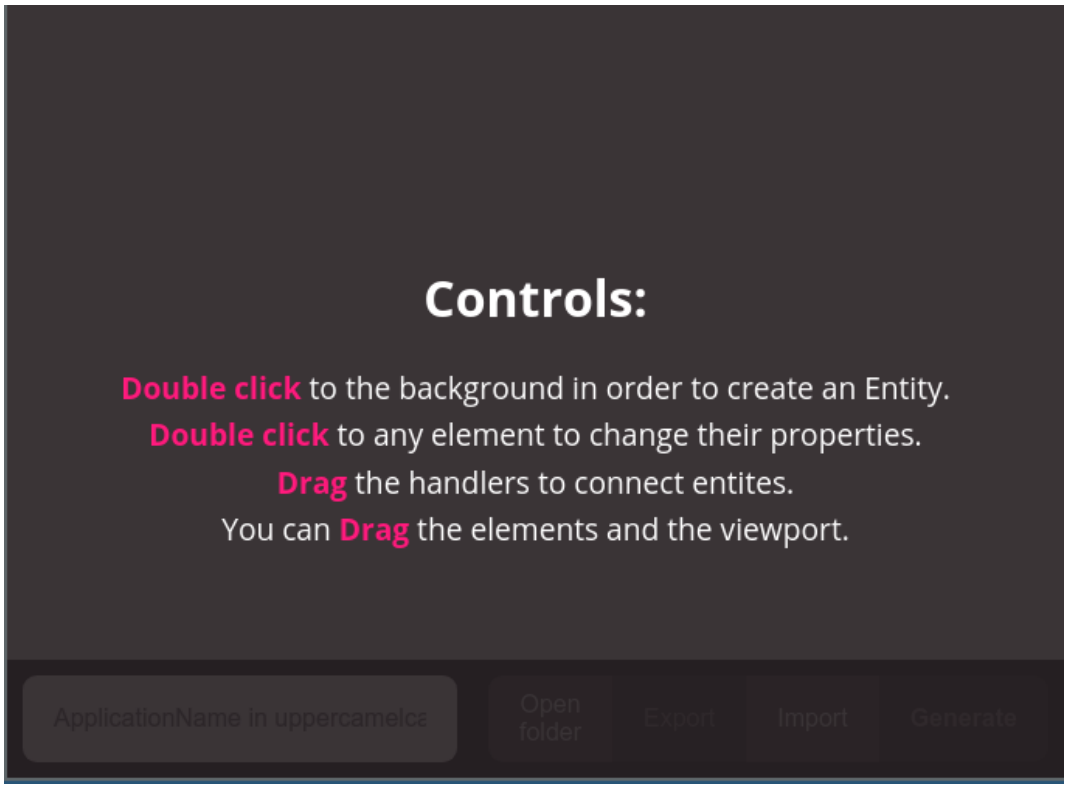
Későbbi továbbfejlesztés során a kiegészítő sablonok át fognak alakulni, ugyanis a megtalált kifejezéstől függhet a kiegészítésnek szánt tartalom és annak elhelyezése is. Jelenleg minden típusú kifejezésre, elhelyezésre külön sablont kell létrehozni, ami növelheti a sabloncsomag átláthatóságát, viszont előfordulhat, hogy ezen sablonok közötti különbség kicsinysége miatt egyszerűbb lenne egy sablonba tömöríteni.

5. A Tervezőfelület megvalósítása



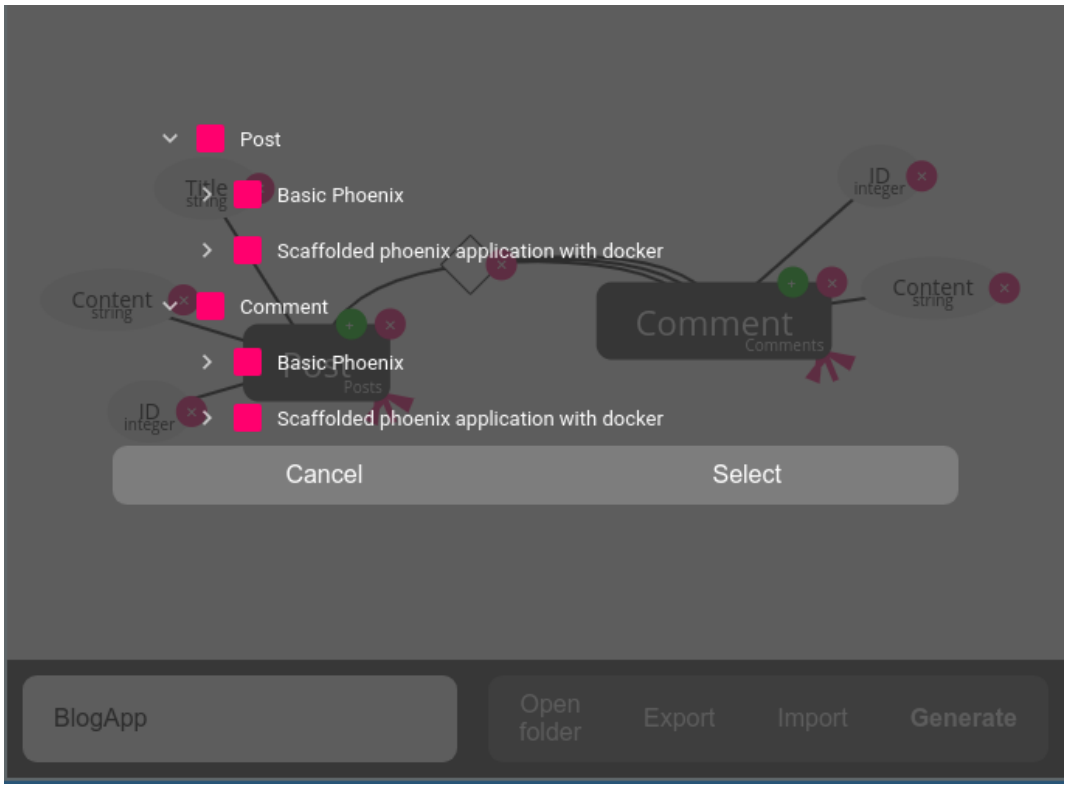
11. ábra. Az ER modell tervezőfelülete

A tervező felületnél figyelembe kellett venni, hogy ne legyen sok panel. A sok panel kitakarja a felhasználók számára érdekelt részt, a modellt. A képernyőn ezért csak alul van egy csík, ami tényleg a lehető legfontosabb elemeket tartalmazza.



12. ábra. Útmutató

A tervező a felhasználó interakciójára várva, előbb egy használati útmutatót ad. Az útmutató eltűnik, amint a felhasználó odahúzza az egér mutatóját. Az útmutató egyértelműsíti a program használatát.



13. ábra. A követelmények kiválasztása

A generálás előtt egy sablon kiválasztási felület ugrik fel. A kiválasztás egy fa szerkezetet mutat a sabloncsomag követelményfájlai alapján.

Jelenleg a generátor működéséből következően minden követelményfájl megkap egy entitást, ezért minden entitásra külön-külön ki kell választani a követelményeket. Későbbi továbbfejlesztési lehetőség, hogy lehetnek majd olyan követelmények is, amelyek nem igénylenek entitásokat. Ilyen követelményeket jelenleg is létre lehet hozni úgy, hogy a követelmény data nevű függvénye minden entitásra ugyanolyan értékkel tér vissza. Ekkor a követelményhez tartozó sablonok csak egyszer generálódnak le, nem pedig minden entitásra külön. Habár használható, nem tűnik jó megoldásnak, és a tervező felületen is okozhat félreértéseket.

6. Hozzájárulás a fejlesztéshez

Egy szoftver fenttartása elengedhetetlen. Előfordulhatnak egyes nem előre látható hibák, amelyek kijavítása szükséges a későbbiekben is. Emellett az integrálható funkcióknak nincs határa, és egy kódgenerátor csak jobb lehet, ha többet tud segíteni. Egy ekkora méretű szoftvert nehéz egyedüli személyként karbantartani és egyben továbbfejleszteni is. Éppen ezért a szoftver nyílt forráskódú, és bárki számára ingyenes. Az alkalmazás teljes állománya megtalálható a https://github.com/fxdave/erscaffold címen.

Irodalomjegyzék

[1] Fogarassyné Vathy Ágnes (2017) *Adatbáziskezelő rendszerek I. - Egyed-kapcsolat modell*

[2] https://www.web-and-development.com/a-framework-or-a-cms-what-is-better-to-choose/ (letöltés dátuma 2019. május 10.) *A Framework or a CMS? What is better to choose? | Web and Development*

[3] https://medium.com/omarelgabrys-blog/database-modeling-entity-relationship-diagram-part-5-352c5a8859e5 (letöltés dátuma 2019. május 10.) *Database — Modeling : Entity Relationship Diagram (ERD) (Part 5)*

[4] http://dillonbuchanan.com/programming/dependency-injection-constructor-vs-property/ (letöltés dátuma 2019. május 10.) *Dependency Injection: Constructor vs Property*

[5] Ivica Crnkovic, Magnus Larsson (2000) *Managing Standard Components in Large Software Systems*

[6] https://www.justinmind.com/blog/rapid-prototyping-isnt-helpful-or-is-it (letöltés dátuma 2019. május 10.) *Rapid prototyping isn't helpful. Or is it? - Justinmind*

[7] https://www.techopedia.com/definition/31002/component-based-development-cbd (letöltés dátuma 2019. május 10.) *What is Component-Based Development (CBD)? - Definition from Techopedia*

[8] https://www.quora.com/What-is-dependency-injection-How-is-it-better-than-procedural-code-if-better-Why-should-I-care (letöltés dátuma 2019. május 10.) *What is dependency injection? How is it better than procedural code (if better)? Why should I care? - Quora*

[9] https://blog.fossasia.org/why-coding-standards-matter/ (letöltés dátuma 2019. május 10.) *Why Coding Standards Matter | blog.fossasia.org*

[10] https://hackernoon.com/you-dont-need-a-dependency-injection-container-10a5d4a5f878 (letöltés dátuma 2019. május 10.) *You Don't Need a Dependency Injection Container - Hacker Noon*

Mellékletek

A szakdolgozat CD mellékletének könyvtárszerkezete:

/szakdolgozat.pdf

/szakdolgozat-forraskod

/rendszer-forraskod

/internetes-hivatkozasok

/A Framework or a CMS What is better to choose Web and Development

/Database Modeling Entity Relationship Diagram ERD Part

/Dependency Injection Constructor vs Property

/Rapid prototyping isnt helpful Or is it Justinmind

/What is ComponentBased Development CBD Definition from Techopedia

/What is dependency injection How is it better than procedural code

/Why Coding Standards Matter blogfossasiaorg

/You Dont Need a Dependency Injection Container Hacker Noon