EÖTVÖS LORÁND TUDOMÁNYEGYETEM

INFORMATIKAI KAR

Programozási Nyelvek és Fordítóprogramok Tanszék

CONTOUR Framework
Fejlesztői keretrendszer JavaScript nyelven

Tejfel Máté

Adjunktus, Informatika PhD

Legéndi Richárd Olivér

Szoftverfejlesztő, programtervező matematikus M.Sc.

Kiss Nándor

programtervező informatikus B.Sc.

Budapest, 2014

Tartalom

1. Bevezető

1.1. Témaválasztás, keretrendszerek

Napjainkban az alkalmazásfejlesztés tervezési lépésének egyik fontos momentuma a technológiák, - és ezzel együtt a keretrendszer - kiválasztása, mivel üzleti elvárás a termék mihamarabbi szállítása, és annak megfelelő rendelkezésre-állása. Egy jól megtervezett, aktív támogatású keretrendszer nem csak kész megoldásaival nyújt tökéletes alapot az alkalmazások írásához, hanem biztosítékot is ad azok helyes működésére, az ismert biztonsági hibák lefedésére. Webes alkalmazások fejlesztéséhez számos keretrendszer közül válogathatunk, de sokszor az újabb technológiák használatához talált keretrendszerek kevésbé kompatibilisek az elképzeléseinkkel, vagy elvárásainkkal. A böngészőben használt népszerű keretrendszerek számomra pontosan ilyenek, mert legtöbbjük kevésbé támogatja az objektum-orientált tervezést, így felmerült bennem egy saját keretrendszer készítése.

1.2. JavaScript

A JavaScript - napjaink egyik legelterjedtebb programozási nyelve – 1995-ben a Sun Microsystems laboratóriumában született a Netscape megbízásából. A terv szerint egy olyan nyelvet kellett létrehozni, mely a HTML kódba ágyazva kis alkalmazások futtatását teszi lehetővé, ezzel növelve a felhasználói élményt. Persze az évek folyamán az elvárások nőttek, így a nyelvnek is fejlődnie kellett. Az üzleti elvárások nyomására hamar születtek más – a JavaScripthez hasonló – nyelvek (JScript[4], ActionScript[5]), melyek újabb dialektusokat hoztak létre. Mielőtt a helyzet kaotikussá vált volna, létre jött az ECMAScript[3] szabvány, mely ezeknek a különböző dialektusoknak nyújt egy nyelvi szabványt.[1]

1.3. Motiváció

A legtöbb fejlesztő számára viszont a JavaScript még mindig csak céleszköz, a HTML elemek animálására szolgáló iskolanyelv. A pehelysúlyú alkalmazások a különféle honlapokon pedig csak erősítik ezeket a sztereotípiákat. A NodeJS megjelenésével viszont elindult egy komolyabb alkalmazási terület, mely még csak

gyerekcipőben jár. Olyan lehetőségek tárháza ez, mely a következő években forradalmasíthatja a webfejlesztést, hiszen olyan nagy cégek is a NodeJS mellett döntenek, mint a PayPal[6].

1.3.1. Webfejlesztés menete, mai módszertanok

Az egyetemi évek alatt többször dolgoztam webfejlesztőként, így gyűjtve tapasztalatokat a szakmában. Nem tagadom, hogy legfőképp a web széleskörű elterjedése miatt kezdtem el ezzel foglalkozni. Manapság a legtöbbet alkalmazott szerkezeti minta az MVC - azaz Model-View-Controller - létrehozott egy kézenfekvő munkamegosztást, szerver- és kliens oldalra tagolva a fejlesztést. Így a sitebuilder és a backend fejlesztő - megfelelő tervezéssel - párhuzamosan dolgozhatnak. Szerver oldalon PHP nyelven kódoltam alkalmazásaimat és eleinte kényelmesnek tűnt a kliens oldal eleresztése. Később viszont úgy éreztem, hogy nagyon sok hiba adódik ezen két oldal egymástól független működéséből. Hiszen a bevett, elegáns szokás szerint a szerver a beérkező kérés válaszaként összeállítja a HTML kódot, majd annak végére fűzi a JavaScript modulok indításához szükséges információkat. Ebben a munkamenetben mindig a szerver állítja elő a HTML kódot, mely nagyon sok műveletet jelent karakterláncokkal. Ezen a ponton elvész az objektum-orientált tervezési lehetőség. Hiszen a HTML sablon fájlokban szétszórt változók és műveletek összességét felváltani osztályelmélettel, majd azt szerializálva előállítani a HTML kódot iszonyú költségeket vonna maga után terhelésben és fejlesztési időben egyaránt. Ez a módszer az ismert karakterlánc műveletek hibáit gerjeszti, kliens oldalon pedig a JavaScript függőségekre kell legfőképp odafigyelni.

1.3.2. Webfejlesztési trendek

Mára egyre többen építik az alkalmazásaikat inkább a kliens oldalra, mintsem a szerver oldalra támaszkodva. Ekkor a backend - néhány szinkron végponton kívül - több aszinkront működtet. Ezek lényeges különbsége, hogy a szinkron végpont válasza HTML kód, vagy egyéb forrás kód, míg az aszinkronoké valamilyen adatstruktúra - többnyire JSON. Ebben az esetben a böngésző már jó néhány alkalmazás logika bonyolítását végzi, de nem szűnik meg teljesen a szerver által előállított HTML kód. Mára a cloud, azaz felhő rendszerek használata is egyre elterjedtebb, melynek

segítségével a felhasználónak nem kell lokálisan tárolnia állományait, hanem egy távoli helyen, biztonságban tudhatja azokat. Ezen felbuzdulva születnek sorra olyan szoftverek, melyek ezen állományok feldolgozását, szerkesztését teszik lehetővé. Így állítva konkurenciát asztali társaiknak, hiszen miért töltse le, szerkessze, majd töltse vissza a felhasználó az adatait, mikor megtudja tenni mindezeket egy helyen, a fel- és letöltést mellőzve.[9]

1.3.3. Contour Framework létjogosultsága

Talán a fentiekből már sejthető, hogy a kliens és szerver közötti kommunikáció inkonzisztenciájára, valamint a böngészőben megjelenő DOM elemek karbantarthatóságára szeretnék megoldást nyújtani. Úgy gondolom, hogy erre most van igény, hiszen a mai trendek is azt követelik, hogy a webes alkalmazások asztali alkalmazásokként működjenek, távoli erőforrásokkal. A Contour Framework ezekre épít, így nem fordít különösebb figyelmet a böngésző támogatottságra, a lapbetöltés gyorsaságára. Viszont ezeket a követelményeket elhagyva is széles körben alkalmazható a keretrendszer, hiszen a böngészők is napról-napra gyorsabbak, és aktív fejlesztésükkel folyamatosan tűnnek el a támogatottsági különbségek az újabb verziókban.

2. Felhasználói dokumentáció

Szakdolgozatom tárgya és terméke egy fejlesztői keretrendszer, így a felhasználói dokumentáció is csak fejlesztőknek szólhat. Egyfajta ajánlást nyújtva a használati módokra.

2.1. Alkalmazási területek

A Contour Framework azoknak a szoftvereknek a fejlesztését segíti, melyek komplexitásukban, szolgáltatásaikban asztali alkalmazásokhoz közelítenek. Ott alkalmazható igazán, ahol nem a lapbetöltés sebességének növelése a cél, hanem inkább a kód karbantarthatósága, a fejlesztési és tesztelési idő minimalizálása. Azoknál az szoftvereknél, ahol az üzleti értékű szolgáltatások rendelkezésre-állást követelnek. Olyan alkalmazások alapjának nyújt alternatívát, melyek akár Qt-ben[7], Swing-ben[8] is megvalósíthatóak, de voltaképp csak egy vékony kliensre lenne szükség, mely az aktuális hardveren semmilyen változtatást nem eredményez. Például webes e-mail kliensek, bug trackerek, adminisztrációs rendszerek, valamint a fentebb említett felhő alapú alkalmazások. Itt jegyezném meg, hogy a szoftver MIT licenccel rendelkezik, így szabadon felhasználható (lásd: 6.1. számú melléklet: MIT licenc).

2.2. Rendszerkövetelmények

Mint ahogyan a legtöbb szoftvernek, ennek is vannak rendszerkövetelményei, melyekre feltétlenül támaszkodik a keretrendszer. Ilyen például a már fentebb is említett NodeJS webszerver, mely elérhető a főbb operációs rendszereken. Valamint a frontend oldal kiszolgáláshoz szükséges egy Nginx webszerver is. Az utóbbi webszerver elhagyható lenne, de statikus fájlok kiszolgálásában ez az egyik leggyorsabb technológia az egyszálúsága, és gyorsítótár technológiái miatt[2]. Ezek mellett a MongoDB[11] adatbázisszerver is szükséges lesz a demonstrációs alkalmazáshoz. A noSQL technológiák is egyre elterjedtebbek azokon a területeken, ahol a rendelkezésre-állás fontosabb, mint a konzisztencia. Éppen ezért a későbbiekben a keretrendszer munkamenet-kezelése is valamilyen hasonló noSQL[10] adatbázisra építkezik majd. Így egyszerű struktúrában villámgyorsan lekérdezhetőek lesznek a felhasználók munkamenetei.

2.3. Demonstrációs alkalmazás telepítése

1. Mindenek előtt rendelkeznünk kell egy Nginx szerverrel. A telepítésről bővebb információk érhetőek el az alábbi linken: http://wiki.nginx.org/Install.

2. Következő lépésben konfigurálnunk kell az előbb telepített webszervert, hogy egy végponton válaszként dobja vissza a kért statikus fájlokat, egy másikon pedig proxy-ként működve nyújtson átjárást a NodeJS szerverhez. Az alábbi képen az általam is használt konfiguráció található, ahol a behelyettesítendő adatok „<<>>” jelek között találhatóak.

```
/home/kipal/node-my-mongo/NodeMongoAdmin/nginx_conf   [-M--]  0 L:[  1+22  23/ 23] *(508 / 508b) <EOF>
server {
<------>listen        80   default;
<------>server_name   node_mongodb;
<------>access_log    /var/log/nginx/node_mongodb.log;
<------>
<------># static files
<------>location /static/ {
<------><------>rewrite /static/(.*) /$1 break;
<------><------>autoindex   on;
<------><------>root        <</static/files/path/>>;
<------>}
<------>
<------>location / {
<------><------>rewrite /(.*) /$1 break;
<------><------>proxy_pass http://<<host>>:<<port>>/;
<------><------>proxy_connect_timeout 30;
<------><------>proxy_send_timeout 30;
<------><------>proxy_read_timeout 30;
<------><------>proxy_set_header Host $host;
<------><------>proxy_set_header X-Forwarded-For $proxy_add_x_forwarded_for;
<------>}
}
```

1. ábra: Nginx webszerver példa konfiguráció

3. Majd telepítsük a NodeJS szervert, az npm - Node Packaged Modules - csomagkezelővel[12] együtt a hivatalos útmutató alapján: https://github.com/joyent/node/wiki/installation.

4. Továbbá szükséges lesz a MongoDB adatbázis szerver telepítése, és indítása: http://docs.mongodb.org/manual/installation/.

5. Ezután következhet a példa alkalmazás letöltése a git repository-kból. A frontend alkalmazás a következő linken érhető el https://github.com/kipal/NodeMongoAdmin/, az adatbázis API pedig https://github.com/kipal/MongoAPI linken. A konfigurációs állomány egy külön

repository-ban helyezkedik el, mivel mind két alkalmazás ugyanezt használja - ezáltal „ismerik” egymást. Célszerű volt ezek almoduljaként felvenni a konfigurációs repository-t. Ebből következik, hogy a repository-k klónozásakor kezdeményezni kell a almodulok rekurzív lekérését, majd a program indítása előtt át kell írni a konfigurációt a saját beállításokra. Tehát a két parancs a következőképp néz ki:

- `git clone git@github.com:kipal/NodeMongoAdmin.git --recursive`
- `git clone git@github.com:kipal/MongoAPI.git --recursive`

6. Természetesen ezektől teljesen független repository-ban található a keretrendszer, de az *npm* csomagkezelő segítségével letöltődik, és a helyes konfigurációban áll a rendelkezésünkre. Ezt a folyamatot az `npm update` paranccsal indíthatjuk el. A két alkalmazásnak különböző függőségei is vannak, így mind a két helyen ki kell adnunk ezt a parancsot.

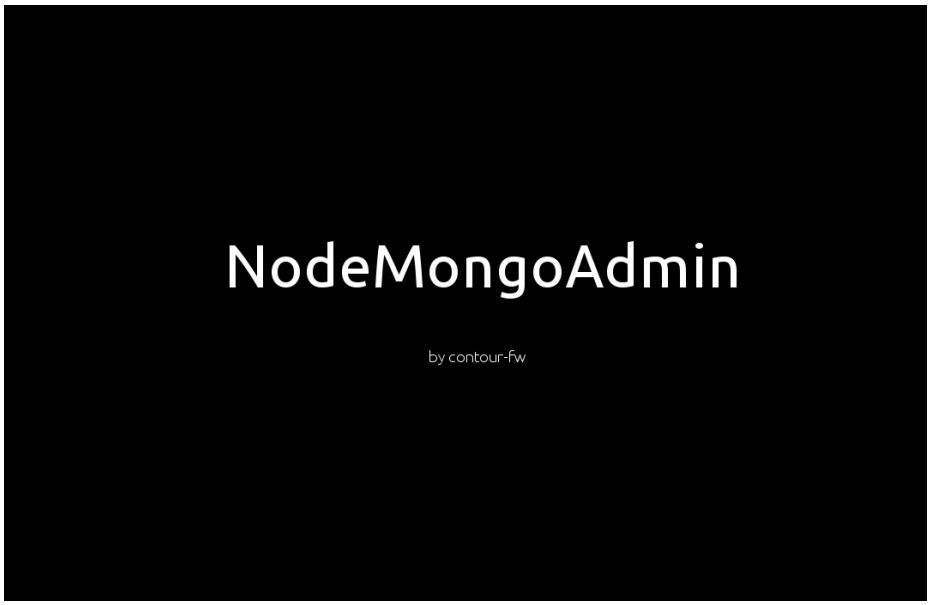
7. Indításuk pedig a `node start-frontend.js` és a `node start-api.js` parancsokkal történik.

   Ha minden sikeresen lezajlott, akkor a terminálablakban a betöltött modulok jelennek meg, ahogy az a következő képen is látható.

2. ábra: Alkalmazás indítási üzenetei

2.4. Fejlesztés bemutatása a demonstrációs alkalmazás segítségével

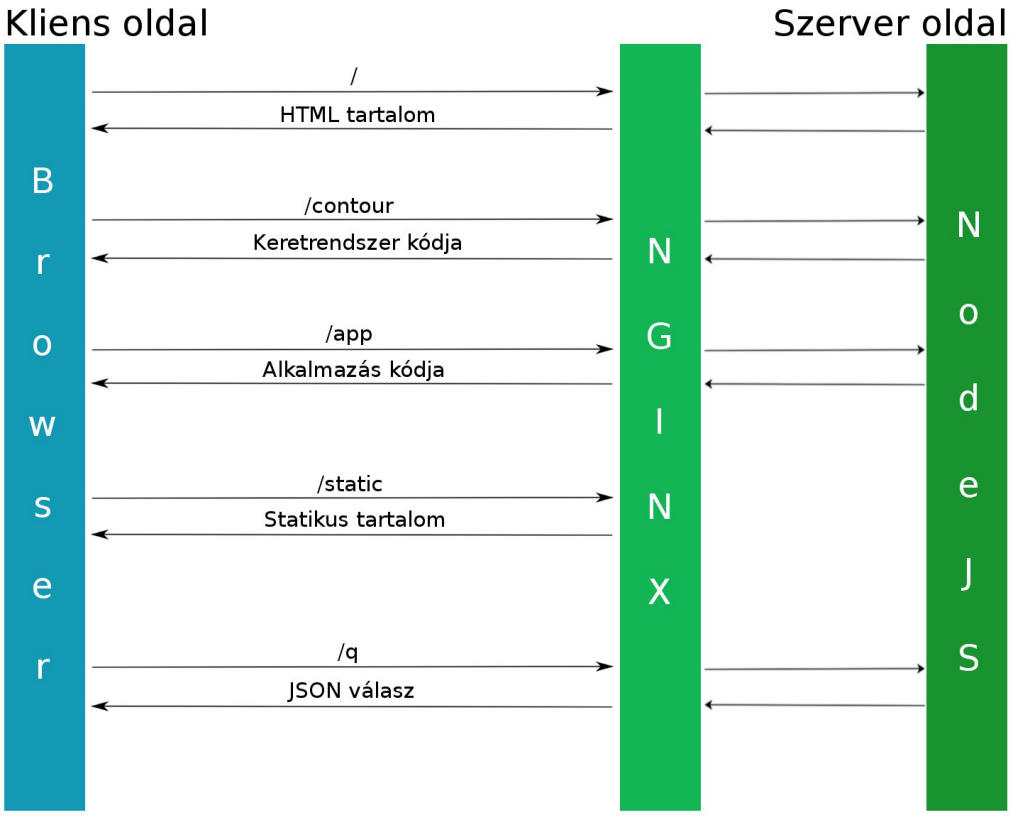
Ha sikeresen elindult a demonstrációs alkalmazás, akkor a „http://localhost”-ot begépelve a böngésző URL sorába a következő üdvözlő képernyő jelenik meg(lásd: 3. ábra).



3. ábra: Alkalmazás üdvözlőképernyője

2.4.1. A lapbetöltés folyamatai

A lenti ábrán (lásd: 4. ábra) jól látszik, hogy a lapbetöltés valamelyest eltér a megszokottól. A bevett szokás szerint a böngésző egy hoszttól URI-k alapján szinkron hívást indít, melynek a végeredménye egy HTML kód, ami linkek formájában tartalmazza a különböző URI-kat. Ezzel persze jöhetnek még továbbá JavaScript kódok is, melyek aszinkron hívásokat is indíthatnak. A Contour Frameworkben viszont a lapbetöltés már az alkalmazás részének számít, így az alábbi módon alakul.



4. ábra: Lapbetöltődés folyamatai

Szinkron hívások

Sajnos a böngésző, csak úgy képes felépíteni egy HTML oldalt, ha a szinkron kérésére, a szerver válasza is HTML formátumú. Így megkerülhetetlen, hogy legalább egy szinkronhívást kezdeményezzen a kliens, mely esetünkben a „/” jelenti. Ekkor a szerver az 5. ábrán látható választ adja.

```
<!DOCTYPE html>
<html>
        <head>
                <script src="/contour"></script>
                <script src="/app"></script>
        </head>
        <body>
        </body>
</html>
```

5.ábra: Szerver HTML válasza a „/” útvonalú kérésre

Ez egy jól áttekinthető, minimális hosszúságú kód, mely két JavaScript forrást kér, a keretrendszer és az alkalmazás kódját. Ezek további két szinkronhívást jelentenek, melyeket csak a szeparáltság kedvéért válogattam szét. Természetesen szerepelhetnének a kódba ágyazva is.

Kliensoldali kódok generálása, küldése

A rendszer ezen a ponton viselkedik teljesen különböző módon, mint eddigi társai. A jól bevált módszertanok a JavaScript forrásokat statikus, előre tömörített tartalmakként kezelték, és küldték ki a kliensekhez. A Contour Frameworkben viszont ezeket a kódokat a saját forrásaiból állítja elő. Megértéséhez vegyünk egy komplexebb példát (lásd: 6. ábra).

```
module.exports = new Service.ClientScript(
    function (BaseBodyWidget, MenuWidget, IntroWidget, CenterWidget) {

        function BodyWidget() {


            this.run = function () {

                new IntroWidget(this.getView().appendNode("div", false), this).run();

                var menuView    = this.getView().appendNode("ul");
                var center      = new CenterWidget(this.getView().appendNode("div"), this);

                var menu = new MenuWidget(menuView, this, center);

                menu.run();
            };

            BaseBodyWidget.call(this);
        }

        BodyWidget.prototype            = BaseBodyWidget.prototype;
        BodyWidget.prototype.constructor = BodyWidget;

        return BodyWidget;
    }
).signUp({
    "name" : "Frontend.MVC.BodyWidget",
    "dep"  : [
        "Contour.Frontend.MVC.BodyWidget",
        "Service.Frontend.MVC.Menu.MenuWidget",
        "Service.Frontend.MVC.Intro.IntroWidget",
        "Service.Frontend.MVC.CenterWidget"
    ]
});
```

6. ábra: Service.Frontend.MVC.BodyWidget kliens oldali modul

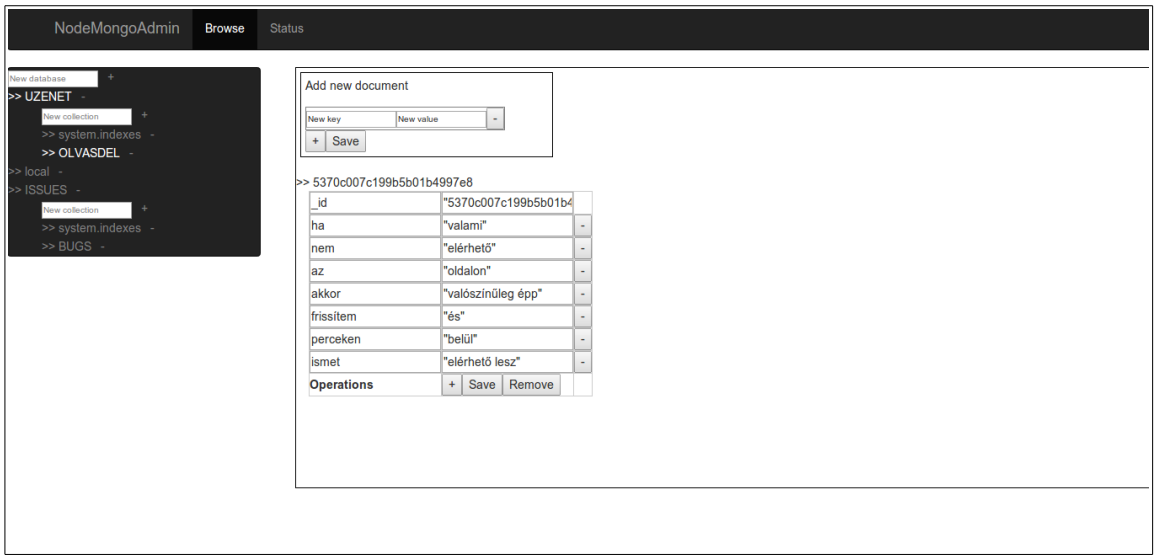
A konstruktor függvény bemutatásával kezdeném a felépítés jellemzését. Ennek a neve, és elérési útvonala esetünkben `Service.Frontend.MVC.BodyWidget`, ahogy az a mintán is látszik. Fontos megemlíteni, hogy a modul végén ez lesz a visszatérési érték is. A 19-23. sorokban látható utasításokkal származtatja magát a `Contour.Frontend.MVC.BodyWidget`-ből. Lentebb a `signUp` metódusban lehet felíratni a modult, hogy kikerüljön kliens oldalra. Ez a metódus lehetőséget nyújt a modul meta adatainak feltöltésére, valamint manipulálására is. A `name` kötelezően feltöltendő attribútum, melyben értelemszerűen meg kell adni a modul nevét. A `dep` kulcs a dependencies kifejezéstt rövidíti, mely alatt sorakoznak a modul által használni kívánt egyéb modulok. Itt jegyezném meg, hogy a modulokra hivatkozó karakterláncok, mindig pontokkal elválasztva szerepelnek. Prefixként az elérési utat kell megadni, és a modul neve kötelezően megegyezik a fájl nevével, akár csak Java-ban az *import* parancs esetén. A `name` attribútumban viszont a gyökér pontot nem kell megadni, hiszen csak ahhoz tartozhat.

Aszinkron hívások

Miután betöltődött az alkalmazás, már csak aszinkron hívások indulnak a szerver felé, méghozzá egyetlen végpontra. Ez a végpont csak JSON kéréseket tud fogadni, mert azt megpróbálja `Request` objektummá alakítani, ellenkező esetben a kérés hibával tér vissza. Persze a hívás lebonyolítására készült egy `RequestHandler` nevű modul, mely csak a megfelelő formában tud kommunikálni, elkerülve a hibákat.

2.4.2. Alkalmazás funkcionalitása

Demonstrációs alkalmazásnak egy adatbázis kezelőt választottam, amelyet a jól ismert *phpMyAdmin*[13] inspirált, így lett a neve *NodeMongoAdmin*. Azért is esett erre a választás, mert a MongoDB adatbázishoz nem találtam NodeJS-ben írt grafikus, webes kezelő szoftvert. Mivel a MongoDB funkcionalitásai széleskörűek, így kezdetben csak egy olyan szoftvert hoztam létre, mely az alapvető, mindennapi adatbázis funkciókat látja el.



7. ábra: Browse menüpont képernyőképe

Jelenleg két fő menüpont érhető el: a „Browse” (lásd: 7.ábra) amely lehetőséget kínál az adatbázisok megtekintéséhez, és módosításához, valamint a „Status”, amely megjeleníti az adatbázis szerver aktuális állapotát.

Adatbázis műveletek

A baloldali dobozban egy listában szerepelnek a szerveren fellelhető adatbázisok. Elütő színnel szerepelnek a *capped* adatbázisok, ezzel is jelezve, hogy belőlük nem lehet dokumentumokat törölni. Egy elemre kattintva betöltődnek az aktuális kollekciók. Az elemek mellett pedig szerepel egy „-” gomb, melyre duplán kattintva törlődik az adatbázis. Új adatbázis felviteléhez pedig a lista legelső elemében szerepel egy input mező, mely az adatbázis nevét várja. A mellette látható „+” gombra kattintva létrejön az adatbázis.

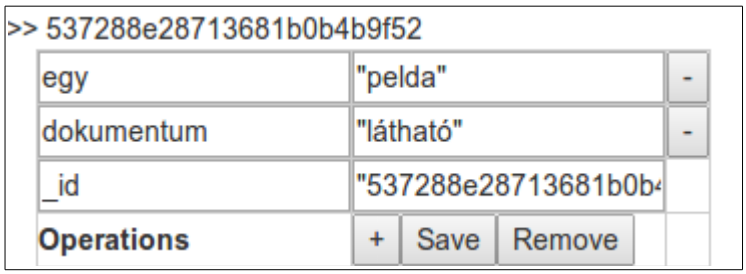
Kollekció műveletek

Ahogy már fentebb említettem, egy adatbázisra kattintva megjelennek annak kollekciói. Egy elemre kattintva betöltődnek a jobb oldalon elhelyezkedő munkaterületre a benne található dokumentumok. Minden kollekció mellett szerepel egy „-” gomb – akárcsak az adatbázisoknál –, amelyre rákattintva törlődik az aktuális kollekció. A lista utolsó eleme pedig egy input mező, mely ebben az esetben is

hasonlóképp működik, mint az adatbázisoknál, s a „+” gombra kattintva felvételre kerül a megfelelő adatbázis alá az új kollekció.

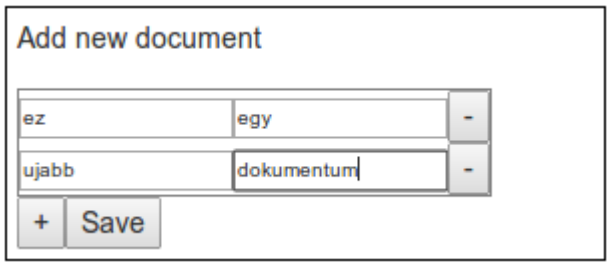
Dokumentum műveletek

A jobboldali munkaterületen az aktuálisan kiválasztott kollekció dokumentumainak listája található. A MongoDB adatbázisban egy dokumentum felvételekor automatikusan bekerült az elsődleges kulcs az *_id* attribútum alá, aminek az értéke egy generált hash. Ezek kizárólagosan azonosítják az egyes dokumentumokat így a listaelemek csukott állapotában csak ezek jelennek meg, majd rájuk kattintva nyílik le egy doboz a tartalmukkal. Lenyitott állapotban a tartalom táblázatba rendezve jelenik meg (lásd: 8. ábra). Az első oszlopban a kulcsok, a másodikban pedig az értékek szerepelnek. Ezek mind input mezők, így szerkeszthetőek is, kivéve az *_id* attribútum, mert ez a konzisztencia fenntartása végett nem módosítható. A módosítható elemek mellett viszont szerepel még egy „-” gomb is, amelyre kattintva eltűnik az adott sor. A táblázat utolsó sorában szerepelnek az „Operations”, azaz a dokumentummal kapcsolatos műveletek. A következő cellában három gomb látható. Az első egy „+” feliratú, amelyre kattintva felkerül egy újabb sor a táblázatba. Az azt követő pedig a „Save”, vagyis a mentést kezdeményezi. Itt megemlíteném, hogy a dokumentum változásai addig nem történnek meg szerver oldalon, míg erre a gombra nem kattintott a felhasználó. Az utolsó pedig a „Remove”, azaz a törlés gomb. Erre kattintva rögtön eltűnik az elem a listából. Ez az egyetlen művelet, ami után nem kell mentést indítani.



8. ábra: Egy dokumentum megjelenése a felületen

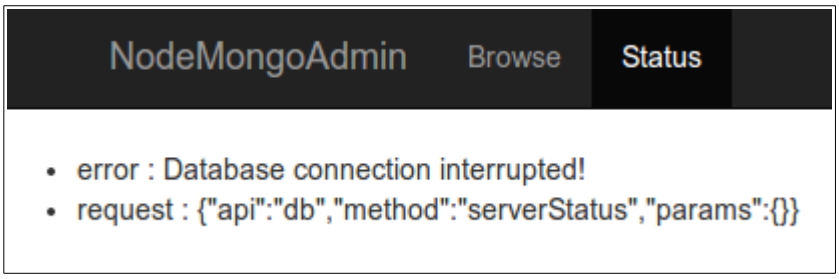
A dokumentum lista felett található egy doboz „Add new document” címmel (lásd: 9. ábra). Ahogy a felvételkor, itt is táblázatba vannak rendezve a kitölthető inputmezők. A „+” gombra kattintva újabb sort ad a táblázatunkhoz, hogy újabb jellemzőket adhassunk meg a dokumentumnak. Itt is minden sor végén található egy „-” gomb, mellyel törölhetjük az aktuális sort. Legalább egy sornak azonban mindig maradnia kell, így egy sor esetén nem fog törölni. A „Save” gomb pedig jelen esetben is a mentésre szolgál. Sikeres mentés esetén újratöltődik a dokumentum lista, amelyben már szerepelni fog a legutóbb felvett elem is.



9. ábra: Új dokumentum felvétele

Szerver státusz

A menüsoron a „Status” gombra kattintva jelennek meg a munkaterületen a szerver aktuális mérőszámai (lásd: 10. ábra). Itt egy faszerkezetű listába rendezve jelennek meg az adatbázis szerver által szolgáltatott adatok. Ha az adatbázis nem elérhető, akkor azt is tudatja velünk.



10. ábra: Adatbázis kapcsolati hiba közlése

2.4.3. Megvalósítás a keretrendszer segítségével

Frontend szolgáltatás – szerver oldal

Az alkalmazás belépési pontja a `start-frontend.js` fájl. Itt elsősorban betöltődik a keretrendszert, majd ennek segítségével az aktuális szerviz moduljai. Itt felhívnám a figyelmet, hogy a `Service.Core.Bootstrap` modul abszolút eléréssel szerepel a kódban, mivel ezen a ponton még nem ismeretesek a függőséget feloldó eszközök. Utolsó lépésében elindítja a `Bootstrap`-et, átadva neki a szerver konfigurációs állományát. Ez a keretrendszert használó alkalmazás indítására szolgál, tehát ez indítja el HTTP szervert feltöltve azt minden hozzá szükséges modullal.

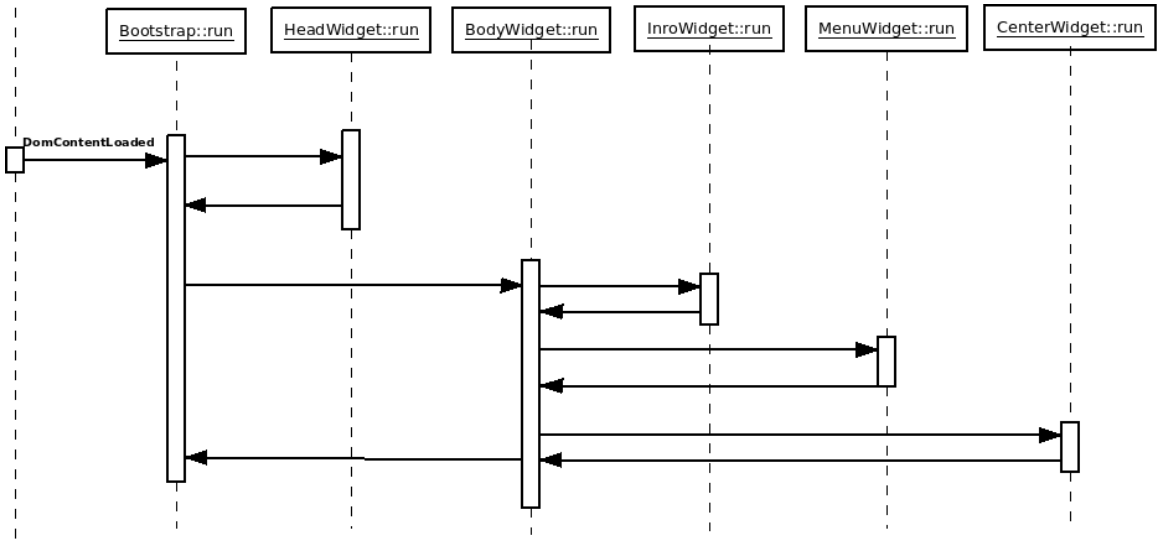
A demonstrációs alkalmazás a `Contour.Frontend.Http.Server` modult használja a HTTP szerver kezelésére, mely egy egyszerű gyerekmodulja a `Contour.Core.Http.AbstractServer`-nek. Ennek csupán egy felelőssége van, hogy a paraméterként megadott porton elindítsa magát, valamint a bejövő kéréseket továbbítsa a szintén paraméterként átadott `ResponseHandler`-nek. Esetünkben ez a `Contour.Frontend.Http.ResponseHandler`, ami a következő - a 2.4.1-es részben már említett három szinkron és egy aszinkron – végpontokat tartja fenn:

- „/” útvonalú kérésre visszaküldi a HTML kódot, benne a két JavaScript forrás hivatkozást.
- „/contour” útvonalú kéréskor a keretrendszer kód regiszterétől kéri a kliensoldali kódok generálását, hogy azt küldhesse válaszul.
- „/app” útvonalú kérés hasonlóképp zajlik, mint az előző, itt viszont az alkalmazás kód regiszterét hívja meg.

- „/q” útvonalú kérések viszont már tartalmaznak POST metódussal küldött paramétereket is. Ez az útvonal a futó alkalmazás aszinkron hívásaira van fenntartva, és a kérés paramétereit a `Contour.Frontend.Http.Router`-nek adja át további feldolgozásra. Ha az `api` kulcs alatt nem a `frontend` érték szerepel, akkor proxy-ként viselkedve továbbítja a megfelelő API-nak a kérést, majd annak a válaszát küldi vissza klienshez.

Frontend szolgáltatás - kliens oldal

A kliens oldali alkalmazás kódja az előző bekezdésben leírt módon betöltődik a böngészőbe - a függőségek szerint helyes sorrendben rendezve a modulokat. Majd itt is a `Service.Core.Bootstrap` indítja el a szoftvert, mely a *DOMContentLoaded* eseményre regisztrálja a `run` metódusát. Ez az esemény indul el, ha már betöltődött a teljes HTML kód a böngészőbe.



11. ábra: A lapbetöltés folyamatának szemléltetése szekvencia diagrammal

Így hát a `Bootstrap` is két widgetet példányosít, a `HeadWidget`-et, és a `BodyWidget`-et (lásd: 11. ábra). Értelemszerűen e két widget hatáskörébe tartoznak a *head* és a *body* DOM elemek. A `BodyWidget` létrehoz egy `IntroWidget`-et, valamint egy `Center`- és egy `MenuWidget`-et. Az `IntroWidget`-hez tartozó `IntroView` úgy manipulálja saját DOM elemét, hogy az töltse ki az egész képernyőt, és középre illessze be az üdvözlőképet. Létrejöttekor a widget beállítja az *onclick*

eseményét, mely bekövetkezésére elhalványul, majd el is tűnik az üdvözlőképernyő. Ekkor láthatóvá válnak az előbb létrehozott `Menu` - és `CenterWidget`-ek nézetei. A menüben betöltődés után aktivizálódik az első - „Browse” - menüpont. A menü példányosításakor megkapja a `CenterWidget` nézetét, ezáltal tudja beleilleszteni az általa létrehozott widgeteket.

Így történik ez a „Browse” menüpont alatt is, ami maga is egy widget, melynek neve `BrowseMenuWidget`. Ez aktiválódásakor létrehoz a `CenterWidget`-ben két újabb widgetet. A bal oldalon látható listadobozt, mely neve `LeftMenuWidget`, és a jobb oldalon található `WorkAreaWidget`-et. A listadoboz létrejöttekor megkapja paraméterként a jobb oldali widgetet, hogy később annak nézetét tudja manipulálni. A listadoboz létrehoz előbb egy inputmezőt, és „+” gombot, amelynek *onclick* eseményére az adatbázis hozzáadás műveletét köti.

Ezek után pedig felsorolásra kerülnek az egyes adatbázisok, amelyek egy-egy listaelemet testesítenek meg a `LeftMenuElementWidget` segítségével. Ennek a `LeftMenuElementView` a saját nézet modulja, amely kiírja az elem nevét, majd mellé rendel egy „-” gombot. A widget pedig a gomb *onclick* eseményére az adatbázis törlés metódusát köti, az elem nevének *onclick* eseménye pedig a kollekció lista nyitását eredményezi. Ekkor a kollekciólista minden eleme a `LeftSubMenuElementWidget` egy példánya lett, amelynek szintén a `LeftMenuElementView` a nézet modulja, így megjelenésében tökéletes mása az adatbáziselemeknek. A „-” gombjának *onclick* eseménye az aktuális kollekciót törli, a kollekció nevének hasonló eseménye pedig betölti a munkaterületre a kollekció összes dokumentumát. Ezek a metódusok természetesen a `LeftSubMenuElementWidget`-ben vannak definiálva. A kollekciók felett szereplő inputmezővel pedig újabb kollekciót lehet felvinni - hasonlóképp az adatbázislistánál –, mely műveletet a `LeftMenuElementWidget` végez.

Ekkor a munkaterületen megjelent dokumentumlista a `DocListWidget` egy példánya. Ez betölti a listáját, melynek minden eleme egy `DocWidget`, majd példányosít egy `DocSaverWidget`-et is, mely egy újabb dokumentum felvételére ad lehetőséget. A `DocWidget` a paraméterként kapott dokumentumot átadja a `DocView`

nézetének, mely megjeleníti annak az *_id* kulcs alatti hash értékét, és alá táblázatba rendezve létrehozza a dokumentum kezelési felületét csukott állapotban. A kulcs *onclick* metódusa lenyitja az alatta elhelyezkedő táblázatot. Az utolsó sorban szereplő „Save" és „Remove" gombok pedig egy `DocUpdaterWidget` és egy `DocRemoverWidget` DOM elemei. A kattintásra a `DocUpdaterWidget` kéri le a táblázat input mezőinek aktuális tartalmát, majd küldi tovább a szervernek mentésre. A `DocRemoverWidget`-nél sincs ez másképp, hiszen kattintásra kezdeményezi a szerveren az aktuális dokumentum törlését. A dokumentum táblázatban szerepelő „-" és „+" gombok csak a DOM manipulálására szolgálnak. A dokumentum lista felett található `DocSaverWidget`-nek a saját nézet modulja a `DocSaverView`, mely kirajzol egy táblázatot, inputmezőkkel. Ahogy a `DocView`-nál, itt is a „+", „-" gombok működtetése a nézetre tartozó logika. A widget csak a „Save" gomb kattintására kötött eseményt kezeli, mely ekkor lekéri az aktuális elemeket a táblázatból és küldi azt mentésre a szervernek.

A menüsoron szereplő második widget - `StatusMenuWidget` - pedig aktiválásakor lekérdezi a szervertől az aktuális státusz adatait. A válaszban többek között szerepelnek az aktuális kapcsolatok számai, a kiszolgáló hoszt címe, a mongodb verziója. A visszakapott adat egy hatalmas JSON objektum, amelyet a `StatusView` - rekurzív feldolgozással – rajzol ki egy faszerkezetű listába.

Az MVC mintából nem hiányozhatnak a modell elemek sem. A kliens oldali modellek pehelysúlyú modulok, mivel metódusaik jellemzően csak egy kérést indítanak a szerver felé, ezzel kezdeményezve az erőforrás valamilyen módosítását. A példaalkalmazásban – nem megszokott módon – csak egy modell létezik, ami magát az adatbázist testesíti meg. Jobban körüljárva a témát ez egy kézenfekvő döntésnek bizonyul, hiszen itt minden adat az adatbázisnak valamilyen jellemzője. Persze lehetett volna még szofisztikáltabbá tenni a modellek listáját (pl.: „Database", „Collection", „Document"), de ezeknek meglehetősen kevés saját jellemzőjük van, melyeket a widgetek már amúgy is ismernek. Mindenesetre ezek szeparált kezelése egy továbbfejlesztési lehetőség.

API szolgáltatás

A szolgáltatást a `start-api.js` állomány indítja, hasonlóképp a frontend alkalmazáshoz, hiszen mindkettő a keretrendszert használja. Így első lépésben betölti azt, majd példányosítja a `Service.Core.Bootstrap`-et a megadott konfigurációval. A bootstrap pedig elindítja a HTTP szervert a konfiguráció szerint. Az alkalmazás saját szerver modult használ, amelyet a keretrendszer moduljából származtat. Ez példányosításakor megpróbál kapcsolatot létesíteni a konfigurációban megadott MongoDB szerverrel. Ha a szerver nem tud adatbázis kapcsolatot létesíteni, vagy ha megszakad az, akkor rögtön hibát dob vissza válaszként, valamint megpróbál újrakapcsolódni. Ha a kapcsolódás sikerrel jár, akkor a kapcsolatot elmenti, majd átadja a saját `Service.DB.Http.ResponseHandler` példányának. A válaszkezelő útvonalak szerint nem különbözteti meg a kéréseket, csak a POST metódussal érkező paramétereket értelmezi, majd aszerint hajt végre lekérdezéseket/módosításokat az adatbázison. Ehhez a válaszkezelő a *mongodb*[33] csomagot használja. Ez a csomag szintén aszinkron függvényekkel kezeli az adatbázist.

2.4.4. Alkalmazás továbbfejlesztési lehetőségei

Mint ahogy azt láthatjuk, az alkalmazás funkcionalitásai nem teljes körűek, így leginkább csak kisebb adatbázisok karbantartására szolgálhatna. A fontos felhasználói követelmény lehetne a későbbiekben a dokumentumok közötti keresés megvalósítása. Ha viszont a *phpMyAdmin* funkcionalitása a mérvadó, akkor majdhogynem teljes lefedettséget kellene biztosítani a MongoDB adatbázisok üzemeltetésére. Ezeket a lehetőségeket a következő listában szedtem össze:

- Felhasználó és jogosultság kezelés
- Replikák kezelése, terhelés elosztása
- Kurzorműveletek támogatása
- Lekérdezések írása

3. Fejlesztői dokumentáció

3.1. Vízió a kapcsolódó technológiák tükrében

Kliens és szerver oldal inkonzisztenciája a mai módszertanokban

Webfejlesztői munkáim során úgy vettem észre, hogy a böngészőben futó JavaScript kód és a backenden futó alkalmazás kooperációja kissé nehézkes, véleményem szerint komplikált. Hiszen a szerver küldi ki a HTML kódot, melyet később a JavaScript programnak kell manipulálnia. Ez máris egy - a JavaScript kódtól függetlenül előállított - forrás, melynek tartalma akár dinamikusan generált is lehet. Így elő is állt az első szembetűnő probléma: a kliens oldali kód nem tudja beazonosítani azokat a DOM elemeket, amelyek szükségesek a futásához. Ennek feloldására a széles körben alkalmazott megoldás a következőképp alakul.

```
▼<script type="text/javascript">
   /*<![CDATA[*/
   jQuery(function($) {
   window.can.init('#id1', '#id2', '#id3', '.class1');
   /*]]>*/
  </script>
```

12. ábra: Példa a szerver által generált JavaScript inicializáló blokkra.

A szerveren előállított HTML kódban – mely itt még csak karakterlánc formájában szerepel – az egyes DOM elemek *id* illetve *class* attribútumait egy-egy megkülönböztető karakterlánccal jelöljük, majd a kliens oldali kódban ezekre hivatkozunk. Valamivel szofisztikáltabb megoldás, ha a szerver a HTML kódba ágyazott „<script>” - „</script>” *tag*-ek közé JavaScript kódot generál, melyben ismerteti a DOM elemek azonosítóit (lásd: 12. ábra). Itt említeném meg a következő problémát: a hivatkozott DOM elemek a kliens oldali kód futásakor lehet, hogy még nem léteznek, hiszen az oldal dinamikusan generált, a JavaScript kódok viszont statikus tartalomként kezeltek. Erre részmegoldás, ha a JavaScript kódok a *DOMContentLoaded* eseményre töltődnek be, mert akkor a böngészőben aktuálisan található HTML kód DOM elemei biztosan léteznek. Viszont egyáltalán nem biztos, hogy a JavaScript kódban függőségként szereplő egyéb DOM elemek is léteznek. Ezekre a problémákat a frontend fejlesztőnek folyamatosan figyelembe kell vennie.

Megoldás a kliens oldali kód előállítására, karbantartására

Ezen problémák ismeretében sikerült egy megoldást kidolgozni: a HTML kódot teljes egészében csak a böngészőben futó JavaScript kód állítsa elő. Ezzel elérhetővé válik az összes DOM elem ismerete a JavaScript kód számára, valamint eltűnnek a szerver által előállított HTML forrás függőségei. Persze ez nem mindenhol alkalmazható eljárás, mert sokszor üzleti elvárás a minél gyorsabb lapbetöltési sebesség. Valamint a fejlesztést megnehezíti a böngészők közötti különbségek lefedése. Ezek többnyire azok az oldalak, amelyeket a hétköznapi értelemben (bemutatkozó) honlapoknak hívunk. Ebből is látszik, hogy inkább azokon a weboldalakon alkalmazható ez az eljárás, ahol alkalmazás építésről beszélünk, ahol üzleti elvárás a kliens oldali alkalmazás logika építése, és ahol elvárható a felhasználóktól, hogy csak a támogatott böngészőket használják.

NodeJS lehetőségei és korlátai

Kutatásaim alatt megismertem újabbnál újabb technológiákat, mint például a NodeJS webszervert és a villámgyors noSQL adatbázisokat. A NodeJS webszerver JavaScriptet futtat, így felmerült bennem, hogy rövidítené a fejlesztési időt, ha ugyanaz a kód használható lenne mind szerver, mind kliens oldalon. Megfelelő absztrahálással ez a lehetőség sok fejlesztési időt takaríthat meg új funkció fejlesztésekor és karbantartáskor is.

A NodeJS megismerése során különös figyelmet szenteltem a hibáinak és korlátainak megismerésére. Egy „NodeJS Budapest meetup”[15] nevű rendezvény egyik előadásán merült fel a probléma az oldalra skálázással kapcsolatban. A webszerver ugyan támogatja a websocketek[16] használatát, amihez szokás szerint Nginx szerver továbbítja a kéréseket. Horizontális skálázás esetén az Nginx szerver *ip_hash* direktívájával képes egy klienst, mindig ugyanahhoz rendszer példányhoz irányítani[14]. Ez fontos, hiszen az egyszálú NodeJS szerver túl sok konkurens websocket kapcsolat esetén túlterhelődhet[37]. Erre a problémára a következő megoldást dolgoztam ki: a keretrendszer beépítve támogatja az API-s felépítést, így elosztva a terheléseket, valamint az API-k konfigurációs állományát megosztva kezeli. Így minden példány ismeri a többi elérhető példányt. Ezzel a megoldással lefedhető az egyes példányok kiesése a hálózatból. A demonstrációs alkalmazás viszont nem

támogatja a websocketek használatát, így ez a lehetőség csak továbbfejlesztések során használható ki.

3.2. Tervezés és megvalósítás

A termékvízió által meghatározott követelmények között két problémát tartottam elsődlegesnek, melyek megoldásai a keretrendszer stabil alapját képzik. Elsőként a kliens oldali kódok előállításának absztrahálását, automatizálását. Ebben akadályt jelentett a NodeJS modul kezelése, melynek feloldása továbbvitt egy saját modulrendszer építéséig. A második probléma pedig az API felépítés támogatása volt, melynek nehézsége az absztrahálásban rejlik.

3.2.1. Modulrendszer felépítése

Eddig még nem esett szó a JavaScript kódok külön fájlokba rendezéséről. Erre a problémára a böngészők nem nyújtanak beépített megoldást. Természetesen a fejlesztés során külön fájlokba szokás rendezni a kódokat, majd azokat *Closure Compiler*-rel, vagy egyéb eszközzel összefűzik, tömörítik. Ezek után már statikus tartalomként kezelhetőek. Ebben az esetben - a konzisztencia megtartása végett -, minden fájl egy modult jelent, mely implementálását a nyelv funkcionális paradigmával, és lambda kalkulus alkalmazhatóságával támogatja. Összefűzéskor fontos tudni, hogy egy modul csak az előtte szereplő modulokat ismeri, tehát kötelező ezek helyes sorrendjének felállítása.

NodeJS module kezelése

A NodeJS a CommonJS szabványban leírt szinkron modulkezelést implementálja[17]. A globális *require* függvénnyel lehet egy másik fájlban lévő kódot kinyerni. Ekkor az betöltődő fájlban definiálódik a *module* változó, mely az aktuális állomány tulajdonságait írja le. A *module.exports*[32] attribútum pedig a publikált értékek elérésére szolgál. Tehát a *require* függvény eredménye a *module.exports* alatt szereplő érték lesz. Így viszont sajnos nincsen lehetőség automatikus modul betöltésre.

Saját modul kezelés

Nem volt kérdéses, hogy a modulok betöltésére megoldást kell találni. A témával való ismerkedés, kutatás során futottam bele a *require-node-directory* csomagba az *npm*

csomagkezelőben[18]. Ez voltaképp csak egy megadott gyökérpont alatti könyvtárakat jár be rekurzívan, és betölti az összes talált modult a megadott gyökérpont alá. Viszont nem oldotta fel a modulok közötti függőséget, ezért elágaztattam a repository-t *contour-require-directory* néven. Ennek a repository-nak a tartalma elérhető az *npm* csomagkezelőben[35].

Szerver oldal

Ebben a kódbázisban definiáltam a `Module` és `Require` modulokat, melyek egy-egy globálisan elérhető konstruktor függvényt testesítenek meg.

```
module.exports = new Module(
    function (Response) {
        'use strict';

        function ResponseHandler() {

            this.getResponse = function () {
                throw 'Contour.Core.Http.ResponseHandler::getResponse is abstract method!';
            };

            this.createResponse = function (body, error) {
                var tmp = new Response();
                tmp.setBody(body, error);

                return tmp.toJSON();
            };
        }

        return ResponseHandler;
    }
).dep("Contour.Core.Http.Response");
```

13. ábra: Példa egy szerver oldali modulra.

A `Module` példányosításkor egy függvényt vár paraméternek, mely egy modul mintát testesít meg. A fenti ábrán (lásd: 13. ábra) megfigyelhető, hogy a *module.exports* veszi fel a `Module` példányt, ezzel biztosítva annak publikálását. A létrejött objektumnak a `dep` függvénnyel lehetőségünk nyílik a függőségeinek megadására. Itt pontokkal elválasztott karakterláncokban adhatjuk meg az egyes modulok elérésének útvonalát és nevét. Ezeket a függőségeket a `Require` modul statikus függvényei ellenőrzik, majd betöltik és adják át az aktuális modulnak.

Kliens oldal

```
module.exports = new Contour.ClientScript.Module(
    function (BaseWidget) {
        function Widget() {

            this.setTitle = function (title) {
                document.title = title;
            };

            this.addBootstrapCss = function () {
                var css = this.getView().appendNode("link");
                css.rel = "stylesheet";
                css.type = "text/css";
                css.href = "static/bootstrap.css";
            };

            this.run = function () {
                this.addBootstrapCss();
            };
            BaseWidget.call(this, document.getElementsByTagName("head")[0]);
        }

        Widget.prototype = BaseWidget.prototype;
        Widget.prototype.constructor = BaseWidget;

        return Widget;
})
.signUp({
    name : "Frontend.MVC.HeadWidget",
    dep  : ["Contour.Core.MVC.Widget"]
});
```

14. ábra: Példa egy kliens oldali modul létrehozására (HeadWidget)

Így a szerver oldali modulok betöltése megoldódott, de nem tisztázott a kliens oldali kódok kezelése. Mivel a böngészőbe kerülő kódokat is modul mintában akartam kezelni a szeparáltság miatt, ezért kézenfekvő volt a `Module` konstruktorfüggvényből kiindulni. Ezen gondolatmenet eredménye lett a `Contour.Core.ClientScript` csomag, mely - a nevéből is láthatóan - már a keretrendszer része. A csomagban található `Contour.Core.ClientScript.Module` konstruktorfüggvény az eredeti `Module`-ból származik, így a használata hasonlóképp zajlik, de működésekor a jóval több lépésen esik át (lásd: 14. ábra). A példány `signUp` metódusa egy objektumot vár, melynek kulcsai az következő táblázatban(lásd 1. táblázat) felsorolva találhatóak.

Meta adat kulcsa	Meta adat típusa	Meta adat leírása
`name`	karakterlánc	Kötelezően meg kell adnunk az alkalmazás nevét, pontokkal elválasztva, a gyökérpontot elhagyva.
`dep`	tömb, értékei karakterláncok	A függőségek listája a megszokott módon, pontokkal elválasztva. Itt meg kell adni a gyökérpontot.
`visibility`	karakterlánc	Értéke „public”, vagy „private” lehet. Alapértelmezetten „public”. A regiszter ennek megfelelően dönti el, hogy csak a gyökérponton belül, vagy kívülről is elérhető-e a modul.
`callback`	Függvény, paraméterként a modul kódját kapja meg karakterlánc formában.	Ezzel a lehetőséggel lehet tovább manipulálni a kódot a küldés előtt. Segítségével például statikus adatokat is publikálhatunk változók értékeként.

1. táblázat: Kliens oldali modul meta adatainak összefoglalása

Ezek után - az előzőleg példányosított - `Contour.Core.ClientScript.Register`-nek adja át az aktuális modult, ezzel regisztrálva azt a kliens oldali kódok sorába. Minden „/contour” és „/app” kérésnél a válaszkezelő egy-egy ilyen `Register` példánytól kéri el a kliensek összefűzött kódját. Természetesen a regiszter csak az első kéréskor gyártja le a kódot, majd gyorsítótárazza azt. A generáláskor a feliratkozott modulokon végig iterálva a `Contour.Core.ClientScript.DepChecker` egy példánya sorba rendezi azokat. Majd a `Contour.Core.ClientScript.Parser` példány segítségével hozzáfűzi a kért függőségek elérését láthatóságukat figyelembe véve. Továbbá hatáskörébe tartozik még a „<private>”, és „<publish>” tokenekkel jelölt részek kivágása, illetve beillesztése is.

```
this.Frontend.MVC.HeadWidget = (function (BaseWidget) {
    function Widget() {

        this.setTitle = function (title) {
            document.title = title;
        };

        this.addBootstrapCss = function () {
            var css = this.getView().appendNode("link");
            css.rel  = "stylesheet";
            css.type = "text/css";
            css.href = "static/bootstrap.css";
        };

        this.run = function () {
            this.addBootstrapCss();
        };
        BaseWidget.call(this, document.getElementsByTagName("head")[0]);
    }

    Widget.prototype             = BaseWidget.prototype;
    Widget.prototype.constructor = BaseWidget;

    return Widget;
}(this.Core.MVC.Widget));
```

15. ábra: Példa a böngészőben megjelenő modulra.

3.2.2. API felépítés

Ahogy azt már a 3.1. fejezetben kifejtettem, a websocketen kommunikáló szerverek oldalra skálázás miatt különösen fontos az API-kra bonthatóság, hiszen a legtöbben a websocket technológia támogatása által jutnak el a NodeJS-hez. De mára már az API-kra bontás - főleg a nagyobb rendszerek tervezésekor - a legkézenfekvőbb terhelésmegosztási lehetőség. Hiszen így egyszerűbben lehet a hardvereket optimalizálni és vertikálisan skálázni.

Keretrendszer támogatása az API-s felépítéshez

Ez az architektúra - végtelenül leegyszerűsítve - egy-egy HTTP szerver mögötti alkalmazás logikát takar. Így máris látszik, hogy a HTTP szerver kezelését, - beleértve a kommunikáció formáját - kell absztrahálni, megkötni. Ekkor egy újabb API létrehozásának folyamata meglehetősen lerövidül, hiszen csak a `Http` csomagok ősmoduljaiból kell létrehozni az újabb, speciálisabb modulokat.

Működés, konzisztencia, korlátok

Egy szolgáltatás felépítése - akár frontend alkalmazás, akár API – rendelkezik egy szerver modullal, ami a `Contour.Core.Http.Server` ősmodulból származik.

Ennek felelőssége kimerül a szerver elindításában a megadott porton, hogy későbbiekben tudja fogadni a kéréseket. A válaszok előállítása viszont már a válaszkezelő hatásköre. Ennek mindig a `Contour.Core.Http.ResponseHandler`-ből kell származnia, és megkapja a kérés tartalmát, valamint a választ küldő függvényt. Ez először különös megoldásnak tűnhet, viszont tudni kell, hogy a NodeJS webszerver aszinkron működik. Így fenn áll a veszély, hogy egy újabb aszinkron függvény indításakor nem fogja megvárni annak lefolyását, és üres válasszal tér vissza a szerver. Tökéletes példa erre a frontend által indított hívás az API-k felé. Mivel ez a metódus is aszinkron fut le, nem tudhatjuk mikor fejezte be futását.

Maga a kérés és a válasz tartalma a `Contour.Core.Http.Request` és a `Contour.Core.Http.Response` modulokban van definiálva. Ez egy fontos megkötés, hiszen ezek struktúrája bármi más lehetne, amikre külön fel kellene készíteni minden kommunikációs félt. A kérés az alábbi táblázatban (lásd: 2. táblázat) részletezett három kulcsot hordozza magával.

Kulcs	**Típus**	**Tulajdonság**
`api`	karakterlánc	Értéke – értelemszerűen – annak az API-nak az azonosítója, melyhez el kell juttatni a kérést.
`method`	karakterlánc	Az futtatandó metódust azonosítja.
`params`	dinamikus típus	A metódus szükséges paraméterei.

2. táblázat: Kérés objektum szerkezete

Itt viszont felmerülhet egy biztonság-technikai kérdés: Biztonságos-e kiküldeni a kliensekhez az háttérszolgáltatások azonosítóit? Sajnos ez kevésbé biztonságos megoldás, viszont a továbbfejlesztési tervek között szerepel ennek megoldása is, amely már generált azonosítókat használ erre a célra.

A válasz is szabványosításra került, amely szerkezete a következő táblázatban található (lásd: 3. táblázat).

Kulcs	Típus	Tulajdonság
`data`	dinamikus típus	A kérésben megnevezett metódus futásának eredménye, visszatérési értéke.
`error`	dinamikus típus	Csak akkor kap értéket, ha futás közben hiba lépett fel. Ekkor jellemzően üres, vagy nem értelmezhető adat szerepel a `data` kulcs alatt.

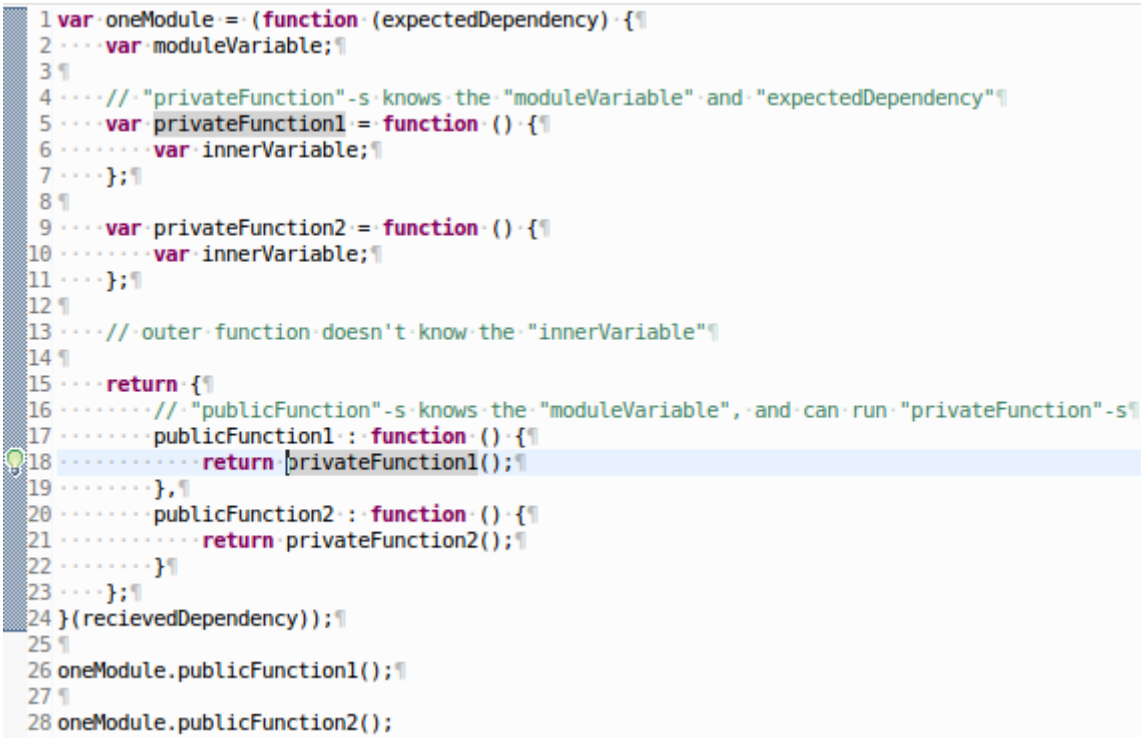
3. táblázat: Válasz objektum szerkezete

A legtöbb API - melyek JSON objektumokban kommunikálnak - válaszában megtalálhatóak ezek a kulcsok. Ugyanis meg kell tudni különböztetni a hibás választ, a futási hibától, hiszen kezelésük a kliens oldalon eltérő lehet. Például, ha egy szoftvercsomag egyik API-ja leáll, akkor a kliensek megpróbálhatnak egy másikhoz csatlakozni, viszont ha az API egyik metódusa során hiba keletkezik az lehet a kliens oldalon tévesen megadott, - vagy rosszindulatúan manipulált - paraméterek következménye is.

3.3. Modulok ismertetése

Nyelvi támogatások kihasználása

A JavaScript másképp kezeli a lokális változók hatáskörét, mint a C alapú nyelvek, amikben egy blokk egy hatáskört jelent. A JavaScript viszont függvény alapú hatásköröket használ, így egy lokális változó – deklarálása után – a függvényen kívül nem látható, azon belül viszont bármely pontján elérhető.



```
var oneModule = (function (expectedDependency) {
    var moduleVariable;

    // "privateFunction"-s knows the "moduleVariable" and "expectedDependency"
    var privateFunction1 = function () {
        var innerVariable;
    };

    var privateFunction2 = function () {
        var innerVariable;
    };

    // outer function doesn't know the "innerVariable"

    return {
        // "publicFunction"-s knows the "moduleVariable", and can run "privateFunction"-s
        publicFunction1 : function () {
            return privateFunction1();
        },
        publicFunction2 : function () {
            return privateFunction2();
        }
    };
}(recievedDependency));

oneModule.publicFunction1();

oneModule.publicFunction2();
```

16. ábra: Példakód a modul minta szemléltetéséhez

Ezekből következik, hogy egy függvényen belül definiált újabb függvény is eléri a külső függvény változóit, akár a külső függvény lefutása után is. Ezeket *closure*-knek hívjuk, melyekre példa a fenti ábrán látható (lásd: 16. ábra) bármelyik belső függvény. Ezek tehát létrejöttükkor átveszik a külső függvény környezetét, úgy, hogy későbbi futásukkor is elérik annak aktuális állapotát. A modul minta is ezekre támaszkodik[19]. Hiszen ha egy változó értékének egy anonim függvény visszatérési értékét állítjuk be, akkor ezen a változón keresztül a visszatérési értéken kívül mást nem érünk el. Így a változó megtestesít egy modult, melynek egyetlen belépési pontja az anonim függvény visszatérési értéke.

3.3.1. Alapstruktúra felépítése, csomagok

Egy fő szempont volt számomra a követelmények létrehozásakor, hogy minél inkább hasonlítsanak a kódok az osztályalapú tervezéshez. Viszont szándékosan nem szeretném az egyes modulokat osztályoknak hívni, mert a modul kilép az osztály korlátai közül, a benne szereplő konstruktor függvény/függvények pedig alulmúlják az osztály lehetőségeit.

Technológia függő saját modulrendszer

A NodeJS a CommonJS által lefektetett modulrendszeren alapszik. Egy fájlt egy modulnak tekinthetünk, melynek az egyetlen behatolási pontja a *module.exports* alatt található (lásd: 17. ábra).

17. ábra: Példa egy NodeJS module betöltésére, használatára

Létrehoztam egy absztrakt `Module` konstruktor függvényt, mely paraméterként egy függvényt vár. Ez nem csak egyszerűen egy függvény, hiszen a lentebb megadott függőségekkel modul mintaként indítja azt. Ezáltal megoldhatóvá vált, hogy az alkalmazás indításakor az összes fájl betöltődjön, és megkapja azokat a modulokat, melyektől függ. Ez jól szeparálttá teszi a kódot, és nem kell használni sehol sem a *require* parancsot. Persze a `Module` a paraméterként megkapott modul függvényt nem tudja továbbküldeni a kliensoldalra. Ezért létrehoztam ennek egy gyermek konstruktor függvényét, melyet `ClientScript`-nek hívok. Ez már kibővített funkcionalitásaival felkínálja a modult szerver és kliens oldali használatra egyaránt.

Csomagok

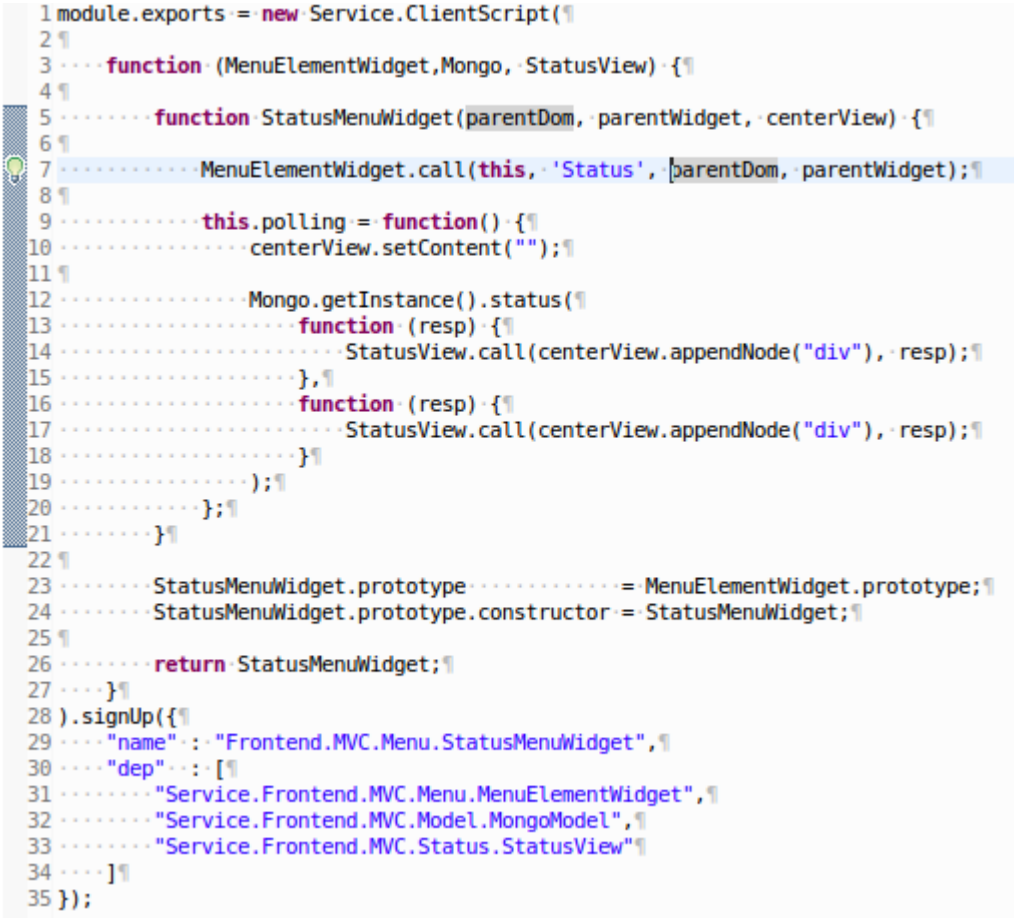
Ahogy azt már a felhasználói dokumentációban is említettem, a szükséges modulok betöltése nagyon hasonlít a Java[20] programnyelvben megszokotthoz. A függőségi modulokat karakterláncokban kell megadni, melyek pontokkal elválasztva az elérési utat testesítik meg, posztfixként pedig maga a modul neve szerepel. Annyi különbséggel, hogy itt előre kell definiálni a gyökérpontot – ami egy globális változó –, és annak könyvtárát. Így függőség feloldásakor visszakereshető a modulok útvonala, majd a betöltött modul hozzácsatolódik a gyökérponthoz. Ezekkel az eszközökkel meg tudtam valósítani a csomagokat, amik valójában egy-egy könyvtárat jelentenek.

Szemléltetésként összehasonlítanék egy – a keretrendszerben megtalálható – modult, és annak egy elképzelt Java változatát.

```
package service.frontend.mvc.menu;

import service.frontend.mvc.menu.MenuElementWidget;
import service.frontend.mvc.model.MongoModel;
import service.frontend.mvc.status.StatusView;

import contour.frontend.mvc.CommonWidget;
import contour.frontend.mvc.View;

public class StatusMenuWidget extends MenuElementWidget 
{
    private View centerView;

    public StatusMenuWidget(Node parentDom, CommonWidget parentWidget, View centerView)
    {
        super("Status", parentDom, parentWidget);
        this.centerView = centerView;
    }

    public void polling()
    {
        centerView.setContent(new StatusView(MongoModel.getInstance().status()));
    }
}

```

18. ábra: StatusMenuWidget kódja Java-ra átültetve

A példán jól látható (lásd: 18. ábra), hogy az osztályunk neve `StatusMenuWidget`, ami a `service.frontend.mvc.menu` csomagban található és a `MenuElementWidget`-ből származik. Működéséhez importál öt külső osztályt. Konstruktorában három paramétert vár a megfelelő típusokkal, majd meghívja az ősosztály konstruktorát inicializálásképp, valamint elmenti a paraméterül kapott `centerView`-t a saját, privát változójába, amit hasonlóképp hívnak. Ezenkívül csak egy függvénnyel rendelkezik, a `polling`-gal, mely a `centerView` tartalmának beállít egy `StatusView` típusú objektumot, amelyet az adatbázis státusz adataival tölt fel. Természetesen a megfelelő szemléltetés miatt, nem használom ki a nyelv adottságait, valamint a Java 1.7-es verziójára[20] alapozva a `MongoModel::status` nem egy *callback* metódust vár paraméterül, hanem visszatérési értékkel rendelkezik.



```
module.exports = new Service.ClientScript(
    function (MenuElementWidget,Mongo, StatusView) {
        function StatusMenuWidget(parentDom, parentWidget, centerView) {
            MenuElementWidget.call(this, 'Status', parentDom, parentWidget);
            this.polling = function() {
                centerView.setContent("");
                Mongo.getInstance().status(
                    function (resp) {
                        StatusView.call(centerView.appendNode("div"), resp);
                    },
                    function (resp) {
                        StatusView.call(centerView.appendNode("div"), resp);
                    }
                );
            };
        }
        StatusMenuWidget.prototype = MenuElementWidget.prototype;
        StatusMenuWidget.prototype.constructor = StatusMenuWidget;
        return StatusMenuWidget;
    }
).signUp({
    "name" : "Frontend.MVC.Menu.StatusMenuWidget",
    "dep" : [
        "Service.Frontend.MVC.Menu.MenuElementWidget",
        "Service.Frontend.MVC.Model.MongoModel",
        "Service.Frontend.MVC.Status.StatusView"
    ]
});
```

19. ábra: A demonstrációs alkalmazás StatusMenuWidget modulja

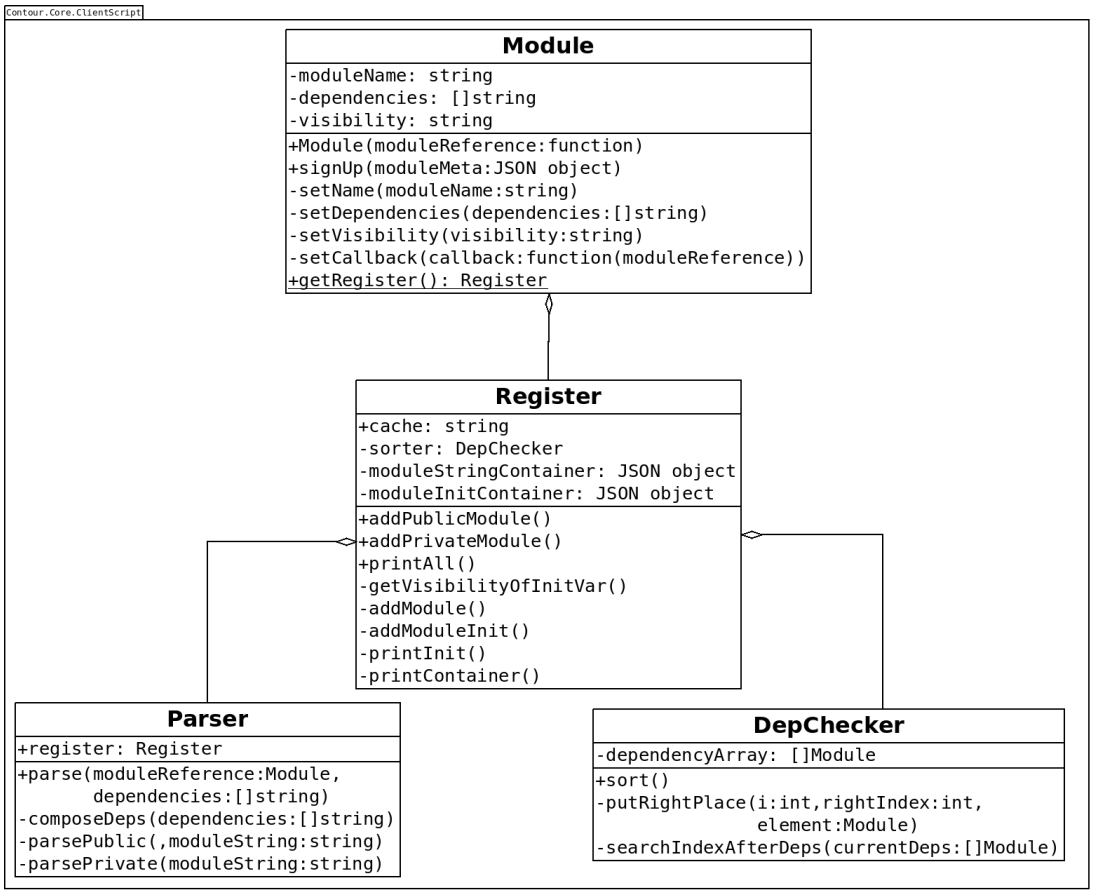
A 19. ábrán pedig már az általam használt `StatusMenuWidget` modul látható, amely tartalmaz egy hasonló nevű konstruktor függvényt. Ez hasonlóképp működik mint a Java-ban definiált osztályok. Ahogy már azt említettem, a NodeJS modulok egyetlen publikálási pontja a *module.exports* változó, így hát ez kapja meg értékként az itt létrehozott modult. A `Service.ClientScript` konstruktor függvény egy példánya a `Contour.Core.ClientScript.Module`-nak, mely megkapja bemenetként az aktuálisan definiált modult. A kód alján meghívott `signUp` metódus a `Service.ClientScript` példány regiszteréhez adja hozzá az aktuális modult. Megfigyelhető, hogy „name” attribútum értéke tartalmazza az útvonalat, viszont a gyökérpontot nem. Ennek oka, hogy a `Service.ClientScript` példány csak a saját gyökérpontjához tud hozzáadni modult, így az elhagyható az útvonalban. Itt jegyezném meg, hogy további fejlesztési lehetőség ennek az attribútumnak az

elhagyása. Az `dep` kulcs alatt találhatóak a függőségek felsorolva, amely megfelelője a Java-ban használt *import* listának. Ezeket a függőségeket a modul paraméterlistáján át kapja meg, ami a példakód harmadik sorában található.

A leszármaztatás máshogy történik, mint az osztály alapú Java-ban. A JavaScript prototípus alapú öröklést valósít meg, tehát a `StatusMenuWidget` *prototype* attribútuma veszi fel a `MenuElementWidget` prototípusát, így örökölve annak értékeit. Valamint az örökléshez tartozik még a `MenuElementWidget::call` metódus meghívása. Ekkor ugyanis az aktuális példány felveszi az ős konstruktorfüggvényében definiált értékeket. Ebből látszik, hogy ez keveréke az öröklésnek és a *super* metódus meghívásának. A `polling` metódus pedig szintén meghívja a `MongoModel::status` metódusát, de itt az paraméterként egy *callback* függvényeket kap, melyek a bejövő választ jelenítik meg a `StatusView` segítségével.

3.3.2. Kliens oldali szkriptek előállítására szolgáló csomag

A csomag a `Contour.Core.ClientScript` útvonalon érhető el, és azzal az egy felelősséggel rendelkezik, hogy szintaktikailag helyesen fűzze össze a felregisztrált modulokat a függőségeikkel csatolva. Ezen feladatokat a csomag négy modulja végzi (lásd: 20. ábra):



20. ábra: Contour.Core.ClientScript csomag UML diagramja

- A - fentebb már említett - `Module`, a globális `Module`-ból származva tud fogadni paraméterként egy függvényt, és szerveroldalon hozzá is tudja csatolni a függőségeit. Ez viszont ennél speciálisabb, okosabb modul, mivel ezen túlmenőleg a `signUp` metódusával fel tudja regisztrálni az aktuális modulfüggvényt kliens oldali megjelenésre. A `Register` egy példányát a modulban tartja számon, melyet a `getRegister` statikus metódussal ér el. A regisztráláskor a modul láthatóságától függően hívja meg a regiszter `addPrivateModule` vagy `addPublicModule` metódusát.

- A regisztrálást a `Register` modul végzi, mely elsősorban képes publikus és privát modulok megkülönböztetésére. A privát modulok csak a gyökérponton belül definiált modulokban lesznek elérhetőek, míg a publikusan azon kívül is. A modulok neveit külön is számon tartja a regiszter, mivel az összefűzéskor

ezeket előre deklarálni kell. Például, ha van egy `Service.Frontend.Http.ResponseHandler` publikus modulunk, akkor - a `Service` gyökérpontot levágva - definiálni kell előbb a `this.Frontend`, majd a `this.Frontend.Http` adattagokat, hogy létre lehessen hozni a `this.Frontend.Http.ResponseHandler`-t. A privát modulok esetén ez annyiban módosul, hogy a `this.Frontend` helyett `var Frontend`-et kell definiálni, majd ennek példájára a továbbiakkal is hasonlóképp kell eljárni. Továbbá a regiszter feladata a gyorsítótárazás, mely esetében a `cache` publikus adattag veszi fel az egyszer már legyártott szkriptet.

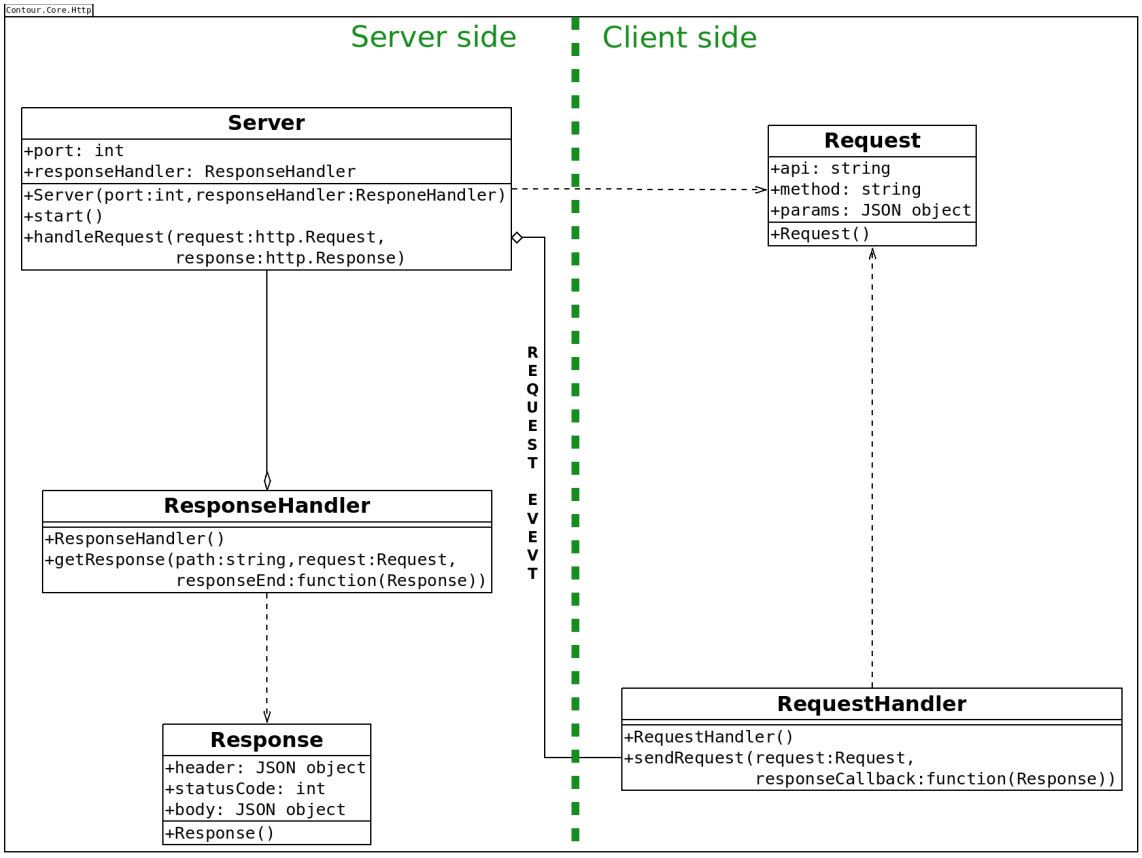
- Az összefűzéskor a regiszter meghívja a `DepChecker` modult, hogy a modulokat sorba rendezze függőségeik szerint. Ez egy kötelezően végrehajtandó metódus, mert egy modul csak az előtte definiált modulokat ismeri. Ennek működését a `sortDeps` metódus végzi, mely addig iterál a modulokat tartalmazó tömbön, míg minden modul a függőségei mögé nem kerül. Ehhez egy kereséssel (`searchIndexAfterDeps`) deríti ki, hogy az aktuális modulnak melyik függősége van leghátrébb, majd azután szúrja be (`putRightPlace`) az aktuális modult. Természetesen ezeket a függőségeket egy fába fel lehetne rajzolni, így tudjuk azt is, hogy ha kört talál a függőségek között, akkor nem tudja végrehajtani ezt a metódust. Ezt külön nem vizsgálja a `DepChecker` modul, és ekkor végtelen ciklusba keveredik. A NodeJS *require* metódusa viszont le tudja kezelni a körkörös hivatkozás hibáját, tehát további fejlesztési lehetőség ennek kihasználása.

- A regiszter összefűzéskor az összes modul kódját, és függőségeit átadja a `Parser`-nek, amely előbb kitörli a „<private>” - „</private>” tokenek között szereplő sorokat (`parsePrivate`), majd a „<publish>” és „</publish>” tokeneket törli (`parsePublish`). Végső soron pedig hozzáfűzi a függőségeket a feldolgozott modul kódhoz(`composeDeps`). Ehhez viszont szüksége van azok láthatósági adataira, melyeket a regisztertől kaphat meg a `getVisibilityOfInitVar` metódus segítségével.

Ezek példányosítására a `Contour` könyvtár gyökerében, és az alkalmazások gyökerében lévő `index.js` fájl szolgál példaként.

3.3.3. Kommunikációs csomagok

A kommunikációért felelős modulok, a `Http` csomagokban szerepelnek. A keretrendszer lehetőséget kínál frontend (`Contour.Frontend.Http`) és adatbázis api-k (`Contour.DB.Http`) különböző kommunikációjára, melyek ősmoduljai a `Contour.Core.Http` csomagban találhatóak. A kommunikáció absztrahálása, szabványosítása az API-s rendszer felépítés miatt elkerülhetetlenné vált. Ezeknek a modulnak a feladatai, a szerver létrehozása, elindítása, kérések fogadása, válaszok küldése.



21. ábra: Contour.Core.Http csomag UML diagramja.

- A `Contour.Core.Server` modul képes elindítani egy HTTP szervert a paraméterként megkapott porton a `start` függvényével, továbbá módjában áll

a kéréseket fogadni a `handleRequest` metódusával. Itt összegyűjti a POST metódusú kérések törzsét, majd átadja azt a válaszkezelőjének, mely a `Contour.Core.Http.ResponseHandler` egy példánya. Ahogy az az UML diagramon is jól látszik, a szerver `handleRequest` metódusa a kliens oldalra küldött `RequestHandler` példány által indított kéréseket fogadja.

- A `Contour.Core.Http.ResponseHandler` felelőssége, hogy a `getResponse` metódusával választ adjon bejövő kérésekre. Ez a modul nem használható példányosításra, mert túl absztrakt ahhoz. Sajnos az absztrakt függvényeket nem támogatja a JavaScript így a `getResponse` függvény absztrakciója kimerül egy hiba dobásában, melynek szövege közli velünk ezt. Ez a metódus megkapja paraméterként a kérés útvonalát, a kérés tartalmát, valamint a válaszadás jogát a *response.end* függvényt[21]. Ez a NodeJS szerver aszinkronitásából ered, hiszen egy újabb aszinkron függvény indításakor megszakad az egyszálú program menete, és a *response.end* függvény nem tudja megvárni annak lefolyását, és üres válasszal tér vissza. Ebből következik, hogy ha az alkalmazás logika úgy dönt, hogy aszinkront függvényt indít, akkor annak az aszinkron függvénynek ismernie kell a *response.end* függvényt, hogy végül választ kaphassunk.

- A szerverekhez a kliensektől a kéréskezelőkön át jut el az aktuális kérés, mely ősmodulja a `Contour.Core.Http.RequestHandler`. Ahogy a fentebb említett válaszkezelő, úgy ez is absztrakt modul, melynek felelőssége, hogy a kérések megérkezzenek a megfelelő célponthoz. Mivel a HTTP protokoll csak szöveges kommunikációra képes, ezért küldéskor a kérés objektumokat JSON formájú karakterláncokká alakítja, melyek a fogadó oldalon visszaalakításra kerülnek. Itt hívnám fel a figyelmet a kommunikációs forma szabványosításának megoldására. A keretrendszer tervezésénél elsődleges szempont volt az API-s felépítés támogatása, tehát a kliens ugyanúgy kommunikál a frontend szerverrel, mint a frontend szerver az API-val. Mindez a `Contour.Frontend.Http.RequestHandler` modul segítségével történik, mely ékes példája ugyanazon modul kliens és szerver oldali hasznosításának. A kódban jól látszik a kommentelt rész a „<publish>” -

„</publish>” tokenek között, mely - ahogy már fentebb említettem - kikerül a kliens oldalra. Viszont a „<private>” - „</private>” tokenek között lévő kód csak a szerver oldalon elérhető. Ebben a két elkülönített részben ugyanaz a metódus van definiálva, csak a két oldal szerint eltérő módon, hiszen a kliensek az *XMLHttpRequest*[22] típusú objektumokkal képesek aszinkron hívásokat indítani, a NodeJS pedig a *http.request* metódusával biztosítja ugyanezt.

- A válasz-, és kéréskezelők szabványos formában kommunikálnak, melyet már fentebb említettem a Működés, konzisztencia, korlátok című alpontban (lásd: 3.2.2. fejezet), az API-król szóló leírásban. Ezeket a szabványokat valósítják meg a `Contour.Core.Http.Request` és `Contour.Core.Http.Response` modulok. A `Request` modul lehetőséget nyújt egy karakterlánc visszaalakításra a `parse` statikus metódusával, melynek visszatérési értéke szintén `Request` típusú lesz. A válasz modul pedig valamivel többet testesít meg a szabványnál garantálásánál. Hiszen ha jobban körüljárjuk a témát, akkor észrevehetjük, hogy egy kéréskor csak a tartalmára vagyunk kíváncsiak, hiszen a fejléc adatait a kliens változtathatja, ezért ezekre alapozni nem lenne biztonságos. A válaszokat, viszont a szerver adja, így azok fejléc adatainak módosítására is lehetőséget ad a `Response` modul. Erre szolgálnak a `statusCode` és `header` adattagok. A státusz kódja valójában csak egy szám, melyek jelentését a HTTP protokoll szabványban fektették le. A `header` pedig egy JSON objektum literál, melynek a kulcsa fejléc tulajdonságok nevei, az értéke meg a választól függ, legtöbbször „application/json”, mivel legtöbb helyen JSON-ban kommunikálnak a rendszerek.

  A keretrendszer előre felkészül a frontend alkalmazások és adatbázis API-k építésére a `Contour.Frontend` és `Contour.DB` csomagokkal, melyekben megtalálhatóak a kommunikációs modulok speciálisabb leszármazottai, de felelősségi köreik nem változnak, így itt külön nem is kerülnek felsorolásra.

3.3.4. Logikai csomagok

A keretrendszer az MVC mintájú alkalmazások építését támogatja hasonló nevű csomagjaival. Az ősmodulok között - a `Contour.Core.MVC` csomagban - csak a `Controller` modul található, mivel a demonstrációs alkalmazás fejlesztése során további absztrakcióra nem volt szükség ezen a téren. A frontend szolgáltatás nem végez semmilyen alkalmazás logikát, de ha szükséges lenne, akkor a kérést egy kontroller példány kapná meg. Majd a kliens a kontroller megfelelő metódusának eredményét egy `Response` objektum `data` értékében kapná meg. Ebből látszik, hogy nézet osztályt - ezen a szinten - felesleges létrehozni.

A `Contour.Frontend.MVC` csomag már bővebb funkcionalitással bír, hiszen ezek a típusú modulok felelnek a böngészőben futó alkalmazás logikáért. A böngészőben futó kódot szerettem volna MVC mintára építeni úgy, hogy az egyes kontrollerek a felületen is jól látható módon különüljenek el egymástól. Többféle gyakorlati megvalósítása van ennek a mintának, viszont általában közös bennük, hogy a kontrollerek nem csak egy nézethez köthetőek, hanem bármelyiket példányosíthatják és használhatják. Ezért inkább a kontrollerek egy gyerekmodulját használom, a widgeteket, amely `Contour.Frontend.MVC.Widget` néven került implementálásra. A widget speciálisabb a kontrollernél, mivel neki biztosan van egy hozzárendelt nézet – `Contour.Frontend.MVC.View` – modulból származó példánya is. Ennek segítségével a widget által definiált metódusok rákötheőek a nézet egyes eseményeire. Így a nézeten keresztül érkező eseményt ténylegesen a widget tudja lekezelni, akárcsak az MVC mintában. Ebből adódik, hogy egy widget példányosításakor meg kell adni azt a DOM elemet, ami az ő nézetének a csomópontja lesz. Természetesen egy nézet példány nem egy DOM elemet jelent, hanem azoknak egy halmazát, mivel képes létrehozni több gyerek DOM elemet. A nézet tulajdonságainak csak csekély részét szolgáltatja a `View` modul, a többi a beépített *Node* konstruktor függvény segítségével jön létre, mely egy DOM csomópontnak feleltethető meg. A `View` viszont lehetőséget nyújt a *CSS*[23] kezelésre is, mely hasonló nézeteknél különösen hasznos opcióvá válik.

Továbbá a widget ismeri a szülőwidgetét, melyet paraméterként kap meg. Ezzel a lehetőséggel a szülő által nyújtott felülethez hozzáfér a gyerek, de még mindig a szülő

végzi el a saját hatáskörébe tartozó feladatokat. Ez számos területen hasznos, ha egy művelethez olyan adatra van szüksége egy gyerekwidgetnek, mely valójában nem tekinthető a sajátjának, viszont a művelethez kötelezően szüksége van rá.

A keretrendszer felkínálja a frontend alkalmazás számára a `HeadWidget` és `BodyWidget` modulokat, melyeket célszerű már a betöltődéskor példányosítani. A `HeadWidget` a *head* DOM elem tulajdonjogát kapja meg, így tud külső forrásokat betölteni, megadni az oldal meta elemeit, valamint beállítani az oldal címét. A `BodyWidget` ennél absztraktabb működést képvisel, ugyanis a `BodyWidget`-en belül példányosulnak az alkalmazás saját widgetei. Tehát célszerű az alkalmazásban ennek egy gyerekmodulját létrehozni.

3.3.5. Egyéb modulok

Ezeken kívül van még két fontos modul a keretrendszerben, melyeket nem lehetett a fent említett csomagokkal együtt kezelni.

Ezek közül először - a sokszor emlegetett - `Contour.Core.Bootstrap`-pel kezdeném. Azért szerveztem ki ezt a modult, mert az általam ismert webes keretrendszerek is hasonlóképp kezelik az alkalmazás indítására szolgáló egységeket. Ezzel is támogatva az egyéb keretrendszerektől érkező fejlesztők munkáját. A működése viszonylag egyszerű, hiszen csak a beadott konfiguráció szerint példányosít egy szervert, majd elindítja azt. Természetesen ezt is célszerű alkalmazásonként speciálisabbá formálni egy leszármaztatott modulban.

A JavaScript nyelvi szinten nem támogat néhány - más programnyelvekben megszokott - eszközt, amik általánosítása merőben megkönnyíti a fejlesztőknek a nyelv használatát. Ezek nem új keletű dolog, az elterjedtebb keretrendszerek is lehetőséget nyújtanak használatukra. A `Contour.Core.Util` modul statikus függvényei segítségével többek között képesek vagyunk egy objektumról megállapítani, hogy üres-e, egy másik objektum attribútumait átmásolni rekurzívan (*deep-extend*). Valamint lehetőséget nyújt egy konstruktor függvényben a *late binding*[24] alapú adattag létrehozásra. Ez felettébb hasznos az öröklések egyszerűsítésére, hiszen az így létrehozott adattag, csak akkor jön létre, ha a gyerekben még nem került az definiálásra.

Enélkül a gyerekben előre definiált adattagokat az ős felülírja a saját, hasonló nevű adattagjaival.

3.4. Eredmények összegzése

Ebben a pontban a dolgozat tárgyát, mint egy projekt termékét értékelem, a követelmények és elképzelések tükrében.

Elsődleges cél volt egy olyan MVC rendszer fejlesztése, mellyel eltűnik a DOM elemek karakterláncokként való kezelése. Az erre a célra fejlesztett widget és view modulok ezt átfogóan képesek karbantartani. Persze egy következő cél ezeket – fejlesztői visszajelzések alapján – speciális gyerekmodulokkal bővíteni, így gyorsítva a keretrendszerben való fejlesztést.

A tervezés során fellépett egy komoly probléma, a NodeJS modulkezelésének automatizálásakor. Az ennek megoldására fejlesztett saját modulrendszer segítségével sikerült megoldani a modulok automatikus betöltését függőségeikkel együtt. Majd ezt továbbfejlesztve sikerült a kliens oldali kódok előállítása is.

További követelményként fogalmazódott meg bennem a beépített terhelés elosztás segítése az API-kra bontással. A kommunikációs csomagokkal a leggyakrabban használt szolgáltatások építését – frontend, és adatbázis szolgáltatások – sikerült megfelelő absztrahálással egyszerűsíteni. Ezt jól mutatja a demonstrációs alkalmazás gyerekmoduljainak felépítése.

A fent leírtak alapján a dolgozatot – az elvárások jegyében - sikeresnek tekintem. Így elképzelhetőnek tartom, hogy megfelelő továbbfejlesztéssel keretet adjon webes szoftverek fejlesztésére.

3.5. Fejlesztéshez használt technológiák, módszerek

A fejlesztés során csak nyílt forráskódú, vagy ingyenesen elérhető programokat használtam, amelyeket egy 64-bites *Linux* operációs rendszeren[25] futtattam. Fejlesztői környezetként az *Eclipse Kepler*[26] verzióját választottam, melyhez telepítettem a *Nodeclipse*[27] beépülő modult. Az összes kód – beleértve a demonstrációs alkalmazást is – verziókezelve van, a *git* szoftver[28] segítségével és megtalálhatóak a

https://github.com/-on publikus repository-kban, melyek elérését az következő táblázatban (lásd: 4. táblázat) foglaltam össze.

Repository tartalma	**Repository elérése**
Demonstrációs alkalmazás frontend szolgáltatása	https://github.com/kipal/NodeMongoAdmin
Demonstrációs alkalmazás adatbázis API szolgáltatása:	https://github.com/kipal/MongoAPI
A keretrendszer kódja	https://github.com/kipal/contour-fw
A keretrendszerben használt – előzőleg elágaztatott repository	https://github.com/kipal/contour-require-directory

4. táblázat: Kódbázisok elérhetőségei

Kódoláskor a Douglas Crockford által ajánlott konvenciók[29] szerint formáztam a forráskódjaimat.

A dokumentációban szerepelő képeket a *Gimp*[30] képmanipulátor programmal szerkesztettem, az UML diagramokat pedig a *Dia*[31] nevű szoftverrel készítettem.

3.6. További fejlesztési lehetőségek

Egy jó keretrendszer fejlesztése sohasem áll le, főleg a weben, hiszen napról-napra jönnek ki az újabb technológiák, amikhez alkalmazkodni kell. Természetesen a Contour Framework még közel sem tart ott, hogy éles rendszereken lehessen használni, így a továbbfejlesztési lehetőségek iránya egy stabil, termék módban használható verzió elérése. Ehhez a következő listában szereplő pontok megvalósítása szükséges:

- Visszamenőleg megírni a létező modulok *unittest*-jeit, valamint a fejlesztést a tesztvezérelt módszertan szerint folytatni, így biztosítva a keretrendszer működésének megbízhatóságát.
- Fejlesztési és termék mód megkülönböztetése, a fejlesztési módba profilozó eszközök beépítése.
- Olyan naplózó csomag fejlesztése, ami többféle erőforrásra tud menteni.

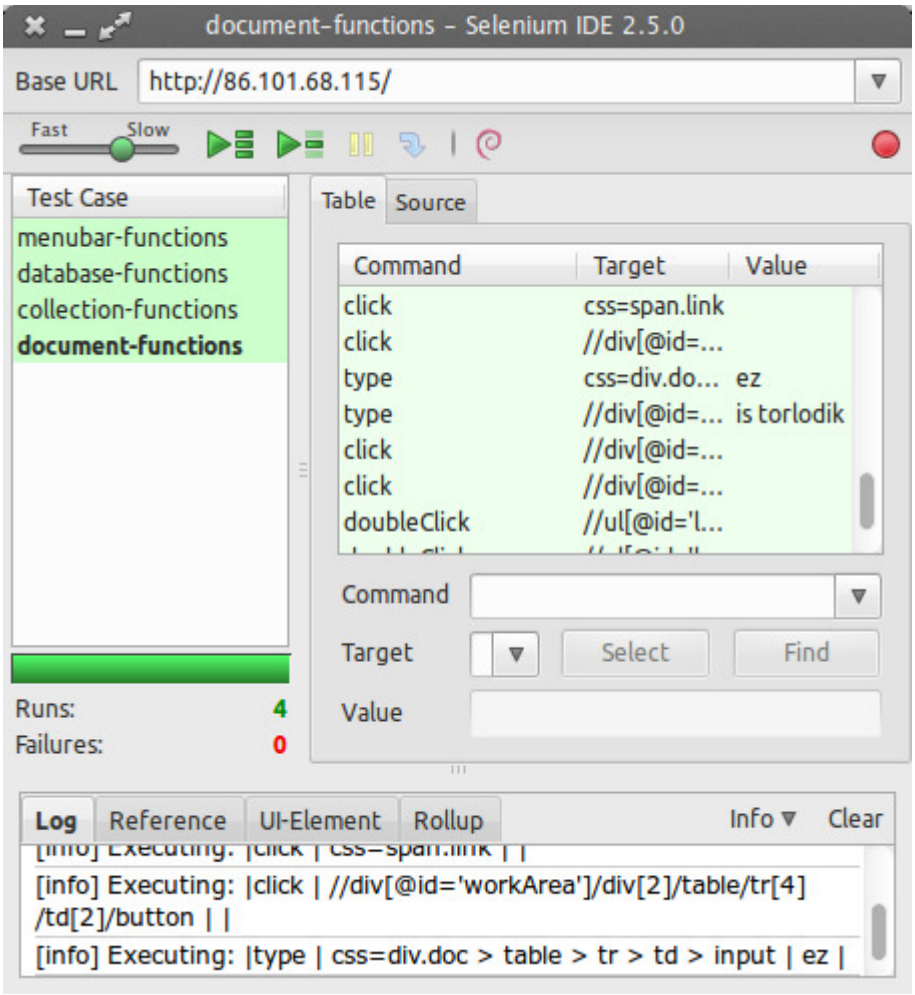
- „RequestQueue” - azaz kérés sor - modul tervezése, fejlesztése, mely a HTTP kérések automatikus kötegelését végezné.
- Biztonsági rések befoltozása az ismert módszerekkel, mint például „CSRF token” használata, „XSS” támadások kivédése a bemeneti karakterláncok vizsgálatával.
- Konfigurációt kezelő modulok fejlesztése, ezzel szabványosítva azokat.
- Új fejlesztések indításának megkönnyítése *skeleton* (váz) projekttel.
- Közös szerver-konfigurációs állomány további kihasználása. Hiszen ha minden API elérhető metódusa megtalálható lenne ebben a fájlban, akkor generáltatni lehetne modulokat, amik az erőforrások pehelysúlyú másait testesítenek meg, így megkönnyítve azok elérését. Ezzel a funkcióval a Működés, konzisztencia, korlátok alpontban (lásd: 3.2.2. fejezet) említett biztonsági követelmény teljesítése is megoldható lenne.
- Továbbá az újabb technológiák követése, támogatása. Ezek közül elsősorban a websocket protokoll beépítése.

4. Tesztelés

4.1. Demonstrációs alkalmazás tesztelése

4.1.1. Automatizált böngésző tesztek

A demonstrációs alkalmazás fejlesztése során az elkészült funkciók tesztelésére a Mozilla Firefox Selenium IDE[34] nevű beépülő modulját használtam. A szoftver fel tudja venni a felhasználói interakciókat, majd automatizálva reprodukálja azokat. Az alkalmazás funkcionális teszteléséhez készítettem egy teljes körű tesztet, melynek az eredménye alább, a 22. ábrán látható.

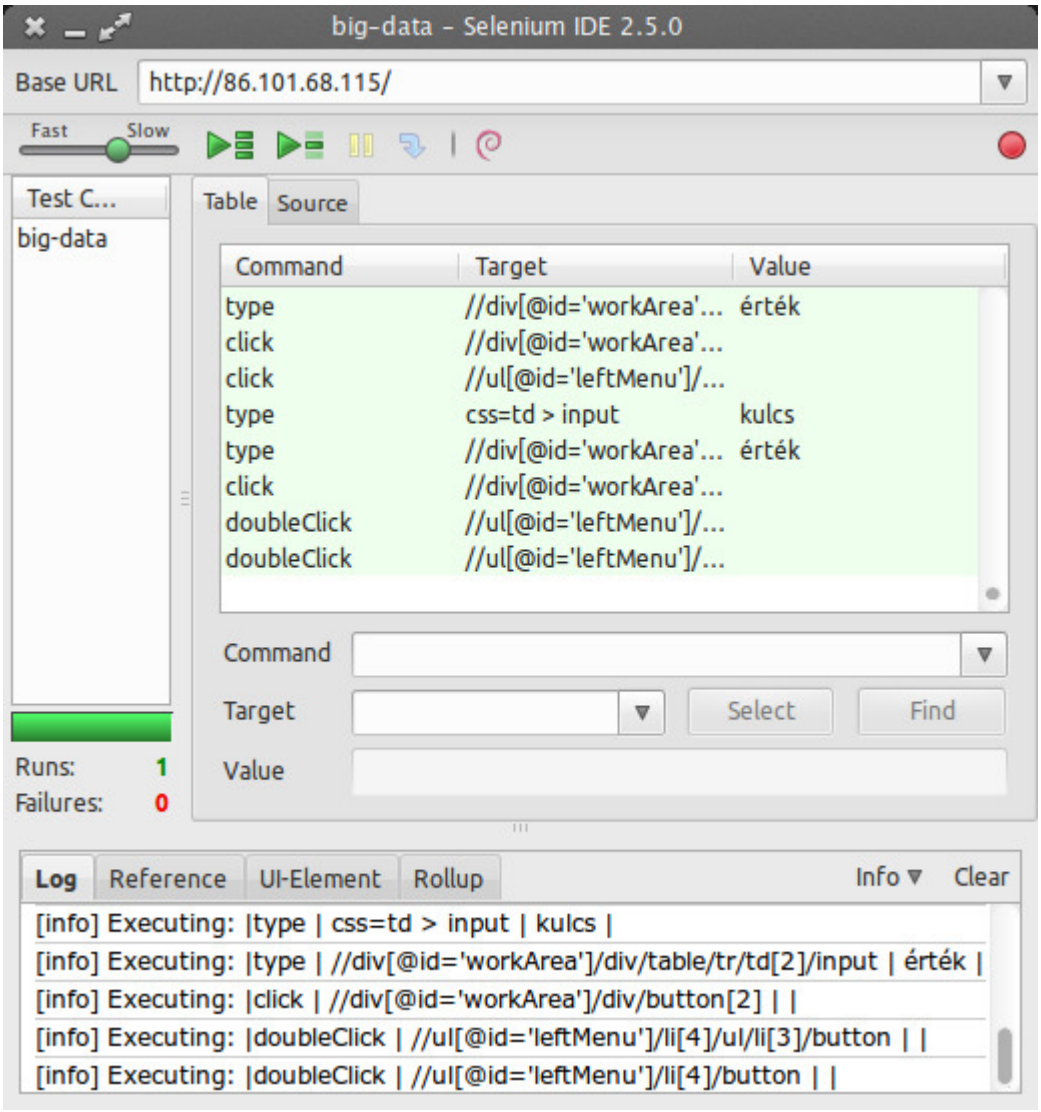


22. ábra: Demonstrációs alkalmazás Selenium IDE tesztje

A teljes körű teszt négy tesztesetet foglal magába, így egy teszteset egy funkciócsoport átfogó tesztelését valósítja meg, melyek a következők:

- A felső menüsor tesztje, amely kimerül a két menüpont kattinthatóságából, valamint a megjelenített tartalmak meglétének ellenőrzéséből.
- Az adatbázis funkciók tesztje, amely során felvételre kerül egy új adatbázis, listázódnak a kollekciói, majd törlődik az.
- A kollekció funkciók tesztjekor egy új adatbázisba felkerül egy újabb kollekció. Majd megnyílik annak dokumentum listája, és ezután törlődik az adatbázis kollekciói közül.
- Az utolsó pedig a dokumentum funkciók tesztje, ahol egy új kollekcióba felvételre kerül több dokumentum, különböző számú attribútummal. Majd ezek módosítása következik, mind érték, mind attribútum szinten. Utolsó lépésként pedig törlődnek a tesztadatok.

Az IDE segítségével terheléses tesztet is végeztem (lásd: 23. ábra), mely egymás után 1000 dokumentumot szúr be, itt a teszt sikeressége a dokumentum lista betöltésének sebességétől függ, mert ezeket is az alkalmazás tölti be. Ha túl lassú, akkor a JavaScript egyszálúsága miatt előfordulhat, hogy nem tudja elküldeni a kérést.

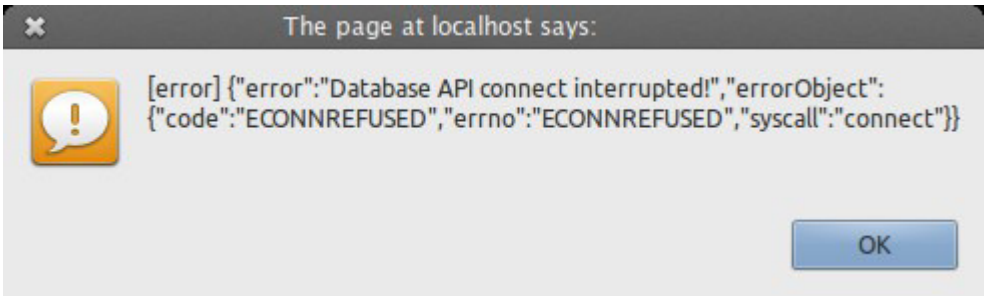


23. ábra: Selenium IDE terheléses teszt

4.1.2. Kommunikációs hibák tesztelése

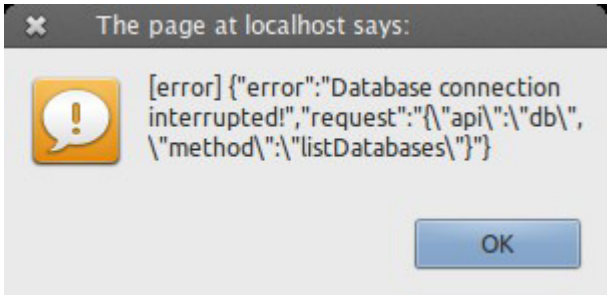
A demonstrációs alkalmazásban többféle kommunikációs hiba léphet fel, hiszen a két API közötti kapcsolat, és az adatbázis szerver kommunikációja is megszakadhat. A rendelkezésre-állás biztosítása érdekében ezeket a hibákat lekezelik az API-k, és arról értesítik a felhasználót.

Előfordulhat, hogy leáll az adatbázis API. Ekkor a kliens által kért adatbázis adatok nem érnek célba, és a frontend API a alábbi hibaüzenetet küldi vissza (lásd: 24. ábra).



24. ábra: Frontend által küldött hibaüzenet adatbázis API kapcsolat szakadása esetén.

Ha a MongoDB API él, de az adatbázis kapcsolat valamilyen módon megszakadt, akkor a felületen a következő üzenet olvasható (lásd: 25. ábra). Ekkor az API újra kezdeményezi a kapcsolódást az adatbázis felé, tehát ha időközben helyre áll a MongoDB szerver, akkor a következő kérésre már a helyes működés szerint reagál.



25. ábra: Értesítés az adatbázis kapcsolat szakadásáról.

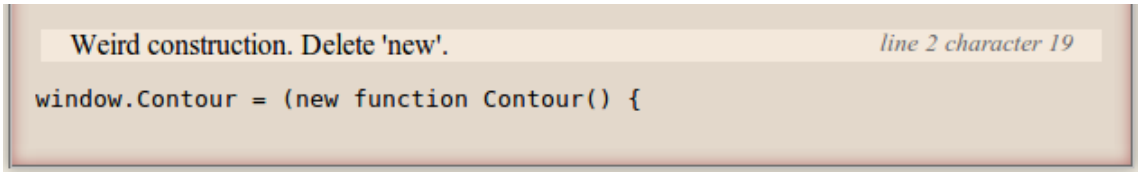
Ha a frontend alkalmazás működése szakad meg, akkor az Nginx webszerver nem tudja a kéréseket a megfelelő portra továbbítani, így a következő hibát küldi vissza (lásd: 26. ábra).

26. ábra: Nginx hibaüzenet a frontend API szolgáltatás leállásakor

A tesztek szerint az utolsó – kiadható verzió – az elvártaknak megfelelően működik, mind funkcionalitás, mind hibakezelés terén.

4.2. Statikus kódelemzés eredménye

A statikus kódanalízis a forráskódokban rejlő potenciális hibákat hivatott felderíteni. A *JSLint*[36] is egy ilyen eszköz, mely az *ECMAScript-262*[3] szabvány szerint ellenőrzi a forrásokat. Mivel a szerveren futó kódok problémáira a NodeJS felhívja a figyelmet futáskor, így fontosabbnak tartottam a keretrendszer kliens oldalra generált kódjának ellenőrzését. Az analízis során egy hiba lépett fel, ami lenti ábrán (27. ábra) látható. Eszerint a gyökérpont példányosítása nem megszokott, mert anonim függvényt hív konstruktornak. De ez nem nyelvi hiba, hanem csak egy konvenció sértés. Ennek kiküszöbölése egy plusz lépéssel elérhető. Előbb be kell tölteni a gyökérpont függvényét egy időleges változóba(„Tmp”), majd a `window.Contour = new Tmp();` utasítással példányosítani azt.



27. ábra: JSLint-tel végzett analízis eredménye

5. Irodalomjegyzék

[1] ELTE-IK, „Prognyelvek portál” [Online] Link: http://nyelvek.inf.elte.hu/leirasok/JavaScript. Elérés dátuma: 2014. április.

[2] WebFaction, „A little holiday present: 10,000 reqs/sec with Nginx! | WebFaction Blog” [Online] Link: http://blog.webfaction.com/2008/12/a-little-holiday-present-10000-reqssec-with-nginx-2/. Elérés dátuma: 2014. január.

[3] Ecma International, „ECMAScript Language Specification ” [Online] Link: http://www.ecma-international.org/publications/files/ECMA-ST/Ecma-262.pdf. Elérés dátuma: 2013. november.

[4] Wikipedia, „Wikipedia” [Online] Link: http://en.wikipedia.org/wiki/JScript. Elérés dátuma: 2014. május.

[5] Wikipedia, „Wikipedia” [Online] Link: http://en.wikipedia.org/wiki/ActionScript. Elérés dátuma: 2014. május

[6] PayPal, „Node.js at PayPal | PayPal Engineering Blog” [Online] Link: https://www.paypal-engineering.com/2013/11/22/node-js-at-paypal/. Elérés dátuma: 2013. november.

[7] Qt, „Qt Project” [Online] Link: http://qt-project.org/. Elérés dátuma: 2014. május.

[8] Wikipedia, „Wikipedia” [Online] Link: http://en.wikipedia.org/wiki/Swing_(Java) 2014. május.

[9] Modus Create, „Top Trends in Web Application Industry for 2014” [Online] Link: https://moduscreate.com/top-trends-in-web-application-industry-for-2014/. Elérés dátuma: 2014. február.

[10] Wikipedia, „Wikipedia” [Online] Link: http://en.wikipedia.org/wiki/NoSQL. Elérés dátuma: 2014. május.

[11] MongoDB, „MongoDB Manual 2.6.1” [Online] Link: http://docs.mongodb.org/manual/administration/production-notes/. Elérés dátuma: 2014. május.

[12] Joyent, „npm” [Online] Link: https://www.npmjs.org. Elérés dátuma: 2014. április.

[13] The phpMyAdmin Project, „phpMyAdmin” [Online] Link: http://www.phpmyadmin.net/home_page/index.php. Elérés dátuma: 2014. május.

[14] Nginx, „Using nginx as HTTP load balancer” [Online] Link: http://nginx.org/en/docs/http/load_balancing.html. Elérés dátuma: 2013. november.

[15] Meetup, „Node.js Budapest November” [Online] Link: http://www.meetup.com/nodebp/events/145101952/. Elérés dátuma: 2013. november.

[16] Kaazing, „Websocket Community” [Online] Link: http://www.websocket.org/. Elérés dátuma: 2014. május.

[17] DailyJS, „Demystifying CommonJS Modules” [Online] Link: http://dailyjs.com/2010/10/18/modules/. Elérés dátuma: 2014. január.

[18] Joyent, „node-require-directory”[Online] Link: https://www.npmjs.org/package/node-require-directory. Elérés dátuma: 2014. január.

[19] Douglas Crockford: JavaScript : The Good Parts, O'Reilly. 2008. [153], ISBN 978-0-596-51774-8.

[20] Oracle, „Java Platform SE 7” [Online] Link: http://docs.oracle.com/javase/7/docs/api/. Elérés dátuma: 2014. április.

[21] Joyent, „HTTP Node.js v0.10.28 Manual Documentation” [Online] Link: http://nodejs.org/api/http.html#http_response_end_data_encoding. Elérés dátuma: 2014. január.

[22] MDN, „XMLHttpRequest - Web API Interfaces” [Online] Link: https://developer.mozilla.org/en/docs/Web/API/XMLHttpRequest. Elérés dátuma: 2013. december.

[23] Wikipedia, „Wikipedia” [Online] Link: http://en.wikipedia.org/wiki/Cascading_Style_Sheets. Elérés dátuma: 2014. május.

[24] Wikipedia, „Wikipedia” [Online] Link: http://en.wikipedia.org/wiki/Late_binding. Elérés dátuma: 2014. március.

[25] Linux.org, „Linux.org” [Online] Link: http://www.linux.org/. Elérés dátuma: 2014. május.

[26] The Eclipse Foundation, „Eclipse” [Online] Link: http://www.eclipse.org/. Elérés dátuma: 2014. május.

[27] Nodeclipse, „Nodeclipse, Enide -- Node.JS development in Eclipse” [Online] Link: http://www.nodeclipse.org/. Elérés dátuma: 2014. január.

[28] Git, „Git” [Online] Link: http://git-scm.com/. Elérés dátuma: 2014. május.

[29] Douglas Crockford, „Code Conventions for the JavaScript Programming Language” [Online] Link: http://javascript.crockford.com/code.html. Elérés dátuma: 2013. október.

[30] GIMP Team, „GIMP - The GNU Image Manipulation Program” [Online] Link: http://www.gimp.org/. Elérés dátuma: 2014. május.

[31] The Dia developers, „Dia draws your structured diagrams” [Online] Link: http://dia-installer.de/. Elérés dátuma: 2014. május.

[32] Joyent, „Modules Node.js v0.10.28 Manual Documentation” [Online] Link: http://nodejs.org/api/modules.html#modules_module_exports. Elérés dátuma: 2013. szeptember.

[33] Joyent, „mongodb” [Online] Link: https://www.npmjs.org/package/mongodb. Elérés dátuma: 2014. február.

[34] SeleniumHQ „Selenium-IDE; Selenium Documentation” [Online] Link: http://docs.seleniumhq.org/docs/02_selenium_ide.jsp. Elérés dátuma: 2014. március.

[35] Joyent, „contour-require-directory” [Online] Link: https://www.npmjs.org/package/contour-require-directory. Elérés dátuma: 2014. május.

[36] Douglas Crockford, „JSLint,The JavaScript Code Quality Tool” [Online] Link: http://jslint.com/. Elérés dátuma: 2014. május.

[37] Drew Harry, „Practical socket.io Benchmarking” [Online] Link: http://drewww.github.io/socket.io-benchmarking/. Elérés dátuma 2013. január.

6. Mellékletek

6.1. számú melléklet: MIT licenc

The MIT License (MIT)

Copyright (c) 2014 Nándor Kiss

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.