Tallinna Ülikool

Informaatika Instituut

JavaScripti haldur veebirakenduses

Seminaritöö

Autor: Kirill Milovidov

Juhendaja: Jaagup Kippar

Tallinn 2015

Autorideklaratsioon

Deklareerin, et käesolev seminaritöö on minu töö tulemus ja seda ei ole kellegi teise poolt varem kaitsmisele esitatud. Kõik töö koostamisel kasutatud teise autorite tööd, olulised seisukohad, kirjandusallikatest ja mujalt pärinevad andmed on viidatud.
.... (kuupäev) (autor)

Sisukord

Sissejuhatus....4
1.Mis on JavaScript?....5
1.1.Programmeerimine JavaScriptis....5
1.2.JavaScript veebis....6
1.3.Node.js ja selle lisad....6
2.JavaScripti kasutamine veebirakenduses....7
2.1.„Tavaline“ JavaScripti laadimine....7
2.2.RequireJS ja AMD....8
2.3.Single Page Application....10
2.4.Symfony Assetic....11
3.JSMan....12
3.1.Klasside loetelu....12
3.2.Mida eeldab kasutamiseks, failistruktuur....12
3.3.Avalik pool....13
3.4.Implementeerimine....14
3.5.Failide lisamine....14
3.6.Failide kokkuehitamine....15
3.7.Mis on halvasti?....15
3.7.1.Koodi kvaliteet....15
3.7.2.JSMan ja vaate kiht....16
3.7.3.Pakkimisalgoritmide puudumine PHPs ja muud kokkupakimise probleemid....16
3.7.4.Muutujad....16
3.7.5.JavaScripti omavahelised sõltuvused ja failide lisamine....17
3.7.6.Failide muutmine....17
3.8.Mis on hästi?....18
Kokkuvõte....19
Lisad....21

Sissejuhatus

Väga raske on leida veebirakendust, kus JavaScripti ei kasutata. Kuid tihti on JavaScript liidestatud ebaefektiivselt, nii rakenduse arenduse kui ka lõpptulemuse seisukohalt.

Klassikalises veebiarenduses tehakse kaks poolt üksteisest eraldi – *frontend*, ehk väljaspoole nähtav ja *backend*, ehk tagumine pool, koodipool. JavaScript on seotud mõlema poolega, kuid tihti jõutakse tema tegemiseni just viimasena või siis püütakse teha samal ajal, kuid ilma erilise rõhuasetuseta.

JavaScripti saab kasutada erinevates keskkondades kuid see seminaritöö keskendub just nimelt veebilehitseja kaudu kasutatavatele rakendustele.

Käesoleva seminaritöö eesmärgiks on tutvustada autori loodud JavaScripti haldamiseks mõeldud teeki JSMan ja mõningaid võimalikke alternatiive ning seeläbi jõuda arusaamisteni kuidas enda rakenduses JavaScripti tasuks hallata.

JSMan valmistati kasutamiseks eelkõige virtuaalserveris kasutamiseks, kus kasutajal ei ole ligipääsu käsureale ega lisapakettide paigaldamisele. Teek on kasutuses portaalis Rada7.ee, alates 2011. aastast.

Esimeses peatükis tehakse kiire ülevaade JavaScriptist. Teises peatükis uuritakse konkreetseid alternatiive ning kolmandas kirjeldatakse ja analüüsitakse autori loodud JSMan teeki.

1. Mis on JavaScript?

JavaScript on laialt levinud dünaamiline programmeerimiskeel, mida saab sisuliselt kasutada igas keskkonnas. Tema kasutamiseks on vaja interpretaatorit, mingit keskkonda, mis oskab JavaScripti käske täita. Tänapäeval on arvutid piisavalt võimsad, et JavaScripti käske on võimalik täita ka *just-in-time* kompileerimisega, ehk siis samal ajal kui rakendus töötab. Just sellist meetodit kasutavad ka enamus tänapäeva veebilehitsejaid.

JavaScript on loodud eelkõige kasutamiseks koos teiste rakenduste/koodibaasidega, kuid üha enam on tänapäeval võimalik luua rakendusi ka ainult tervikuna JavaScriptis. Seminaritöö keskendub veebilehitsejates jooksva koodiga ning JavaScript on selles kontekstis lisa abivahend, mitte põhiline koodibaas. (Mozilla Developer Network, 2014)

Igal veebilehitsejal on tavaliselt oma mootor JavaScripti jooksutamiseks. Need mootorid on erinevad ainult oma kapoti all oleva koodi poolest, tihti on nad isegi saanud alguse samast koodibaasist, kuid ajapikku läinud oma rada. Veebilehitsejas olevad JavaScripti mootorid peavad vastama teatud spetsifikatsioonidele, et erinevatel klientidel oleks võimalik sama koodi jooksutades saada täpselt sama tulemus (MDN, 2014). Klient tähendab siinsel juhul veebilehitsejat, mis parajasti mingit JavaScripti koodi jooksutab. Edaspidi käsitletakse kliendi mõistet just selle definitsiooni järgi.

1.1. Programmeerimine JavaScriptis

JavaScripti peetakse üsna lihtsaks kuid võimsaks keeleks. Seda õpetatakse isegi mõnedes üldhariduskoolide algklassides. Ta on oma olemuselt skriptikeel, kuid toetab ka (läbi prototüüpimise) võimsaimaid objekt-orienteeritud programmeerimise aspekte. Ta võimaldab anda funktsiooni parameetrina teise funktsiooni ja tagastada väärtusena funktsiooni. Nagu eelpool juba mainitud, on tegemist dünaamilise keeleega. See tähendab, et JavaScriptis etteantud käske on võimalik täita ilma, et peaks koodi eelnevalt kompileerima. (MDN, 2014)

Üha enam kasutatakse JavaScripti koodi loomiseks abivahendeid nagu CoffeScript või LiveScript. Need on sisuliselt JavaScripti alamkeeled, mis kompileeritakse kokkuvõttes JavaScriptiks. Tänu neile on võimalik kirjutada paremini struktureeritud, lühemat ja loetavamat koodi. Nende kompileerimine ja rakendusse lisamine käib üldjuhul läbi eelnevalt loodud

ülesannete, kasutades Node.js-i ja selle laiendusi.

1.2. JavaScript veebis

Veebirakendustes kasutatakse JavaScripti *Document Object Model*-i ehk DOM-i mugavamaks töötlemiseks ja muutmiseks. DOM on veebirakenduses HTML leht, mida klient parasjagu vaatleb. JavaScripti abil on võimalik „kuulata“ erinevaid kasutajapoolseid interaktsioone, näiteks hiireklikke, klahvivajutusi ja puutetundliku ekraani puudutusi. Lisaks on võimalik sekkuda otseselt DOM-i struktuuri ja seda muuta, laadida andmeid juurde ja eemaldada; töödelda stiile.

1.3. Node.js ja selle lisad

Kuigi seminaritöö keskendub veebikeskkondadele, siis on tihti vaja kasutada lisavahendeid, et koodiga eelnevalt toimetada. Sellisteks töödeks kasutatakse Node.js-i ja sellele tehtud lisasid. Nende abil on võimalik automatiseerida erinevaid korduvaid ülesandeid: näiteks Node.js lisad nagu Grunt.js ja Gulp.js võimaldavad ehitada kokku kogu koodibaas, teha failid lühemaks, neid kokku pakkida jne.

Node.js-i on võimalik kasutada ka serverite ehitamiseks ja muudeks ülesanneteks, kuid need jäävad seminaritöö raamidest välja (Joyent Inc., 2014).

2. JavaScripti kasutamine veebirakenduses

Selleks, et kasutada JavaScripti veebiarenduses tuleb see paigutada HTML lehe sisse, kasutades selleks ettenähtud elementi. Selleks elemendiks on `<script>` (W3C, 2014). Selle kaudu on võimalik laadida JavaScripti faili kindlast asukohast (veebiaadress) või siis kirjutada JavaScripti koodi otse <script> elemendi sisse. Elementi ennast on võimalik HTML koodis paigutada arendaja äranägemise järgi, kuid siin tuleb järgida rakendusele omaseid eripärasid.

2.1. „Tavaline“ JavaScripti laadimine

Kuna HTML lehte loetakse järjest, siis kõik <script> elementide sisud laetakse ka vastavalt. Need mis on ees, laetakse enne. Üldjuhul asetatakse <script> kas <head> elemendi sisse või enne <body> elemendi lõppu. Midagi ei takista ka suvalisse kohta <script> elemendi asetamist, kuid siin tuleb silmas pidada etteantud HTML koodi kasutatava standardi järgimist. Lisaks sellele, on mitmesse erinevasse kohta lisatud koodi palju raskem hallata.

Klient laeb HTML lehte järjest, mis tähendab, et näiteks kui <script> elemendid asuvad <head> elemendis, ei saa lehte edasi laadida, enne kui on laetud <script> elemendi sisu. Siin võib tekkida probleem, kui arendaja soovib lisada kümneid või sadu JavaScripti faile. Kuna iga faili laadimiseks peab avama ühenduse, laadima, sulgema ühenduse, läheb järjest kümnete failide laadimine üsna aeglaseks, isegi siis kui faili suurus on väga väike. Eriti tundlikud on selle suhtes internetiühendused, mis on küll suure andmemahuga (*bandwidth*), kuid aeglase *ping*-iga. Peale selle, on veel vaja jälgida, et aadress, mis antakse, on korrektne ja, et failide järjekord oleks õige.

Käsitsi faile lisades on lihtne eksida õige järjekorraga. Näiteks JavaScripti raamistiku jQuery *plugin*-ite ehk lisade kasutamiseks on oluline, et eelnevalt oleks jQuery ise juba laetud. Antud juhul on tegemist üsna lihtsa sõltuvuse näitega, kuid juba kasutades mõnda jQuery *plugin*-i funktsionaalsust oma koodis, on oluline jälgida mis järjekorras midagi sisse laetakse.

```
<html>
<head>
	<!-- laadimine päises -->
	<script type="text/JavaScript" src="js/jquery.js"></script>
	<script type="text/JavaScript" src="js/jquery.ui.js"></script>
	<script type="text/JavaScript" src="js/validator.js"></script>
	<script type="text/JavaScript">
	$( document ).ready(function(){
		$('div').hide();
	})
```

```
	</script>
</head>
<body>
	<div>sisu</div>

	<!-- laadimine jaluses -->
	<script type="text/JavaScript">
	$( document ).ready(function(){
		$('div').show();
	})
	</script>
</body>
</html>
```

Näide 1. Tavaline JavaScripti laadimine.

Otstarbekas oleks luua mingi süsteem, kus JavaScript on hoiustatud koos muu koodibaasiga ja oleks olemas mingisugune lisarakendus, mis JavaScripti avalikku, kliendipoolele, laeks.

2.2. RequireJS ja AMD

RequireJS on loodud täpselt selleks, et hallata JavaScriptis loodud koodi laadimist oma veebirakendusse. Selle teegi eesmärk on pakkuda lihtsat lahendust failide/teekide omavahelistele sõltuvusprobleemidele ja laadida faile samaaegselt, ehk asünkroonselt. Sedasi faile allalaadides ei oodata ühe faili laadimise lõppu, enne kui järgmise laadimisega alustatakse, vaid laetakse kõik samaaegselt (RequireJS, 2014). Seetõttu on RequireJS puhul tegemist *asynchronous module definition* standardit kasutava teegiga (RequireJS, 2014).

Failide järgnevus ja omavaheline läbisaamine määratakse ära igas failis eraldi. Juurdelaetud failist võetakse vastavad sõltuvusreeglid kohe kasutusse ja määratakse seesmiselt teekide õige järjekord. Seega on RequireJS ka *dependency manager* ehk sõltuvushaldur. Iga faili nimetatakse RequireJS kontekstis mooduliks.

```
<html>
<head>
    <script data-main="js/main" src="js/require.js"></script>
</head>
<body>
    <div id="container">sisu</div>
</body>
</html>
```

Näide 2. RequireJS käimapanemine.

Näites 2 on ainult üks <script> element mis laeb sisse RequireJS teegi. Seda laadides peab juurde märkima (läbi data-main atribuudi) esimese faili (mooduli), kust alates hakatakse JavaScripti

koodi edasi laadima. Esimene fail on initsiaatorfail, tema sees on esimesed sõltuvused millest hakkavad hargnema ülejäänud (kui neid on). Kaks tähelepanekut:

1. data-main atribuudi väärtus on esimese mooduli failinimi ilma .js faililaienduseta
2. Edaspidi eeldatakse, et kõik moodulid asuvad /js kataloogis

Moodulid peavad sisaldama konkreetset süntaksit, et RequireJS nende sõltuvusi maha suudaks lugeda ja nende funktsionaalsust olemasolevasse rakendusse lisada. Samas, kui pole vaja lisada uusi sõltuvusi ega kasutada olemasolevaid, saab ka lihtsalt JavaScripti koodi kirjutada otse faili.

Vaatleme lihtsat moodulit, mis asendab HTML lehel elemendi sisu.

```
define(['jquery'], function(jQuery) {
    jQuery('#container').text('uus sisu');
});
```

Näide 3. Lihtne moodul, mis sõltub jQuery teegist.

Kuna moodul algab käsuga „define“ siis tähendab, et RequireJS annab mooduli koodi täitmiseks alles siis kui vastavad sõltuvused (teised moodulid) on laetud. Esimesel real peame defineerima faili nime („jquery“, ilma faililaienduseta .js) ja siis andma talle meie moodulis kasutatava muutuja (kasutatav selle nimega ainult selles moodulis), milleks on „jQuery“. Edaspidi saame seda muutujat kasutada tema funktsioonide väljakutsumiseks (RequireJS, 2014). Kuigi see näide kasutab ainult ühte sõltuvust, saab neid lisada korraga nii palju kui tarvis.

```
define(['jquery', 'chosen/chosen.jquery'], function(jQuery, chosen) {

    var selectBox = '<select class="chosen-select" data-placeholder="vali">'
        + '<option>valik 1</option>'
        + '</select>';
    ;
    jQuery('#container').append(selectBox);

    jQuery('.chosen-select').chosen();
});
```

Näide 4. Mitme sõltuvusega moodul.

RequireJS on vägagi mahukas teek, ning tema sügavamal lahtiseletamisel pole käesolevas seminaritöös mõtet kauem peatuda. Ta toetab keerulisemaid seadistusi, globaalsete muutujate defineerimist, kindlalt aadressilt moodulite laadimist (mitte etteantud kataloogist) jne.

2.3. Single Page Application

Viimased aastad on kanda kinnitanud uut tüüpi veebirakendused, kus kliendivaates kuvatav HTML ei ole enam genereeritud serveri poolt, nagu seda näiteks PHPga tehtud rakendused, vaid hoopis JavaScriptiga. Selliseid rakendusi nimetatakse *single-page application*-iteks, mis tõlkes tähendab ühelehelisi rakendusi.

Tööpõhimõte on lihtne. Arendaja loob HTML lehe, kus on üldine struktuur ja defineeritud mõned konteiner tüüpi elemendid (nt), ehk elemendid mille sisse võib mingit sisu laadida (MDN, 2003). Kliendi poole peal luuakse veebirakenduse aadresseerimist. Kui kasutaja teeb päringu teatud aadressile, loeb rakendus kliendi poole peal välja milliseid JavaScripti mooduleid tal vaja on ja laeb need. Kasutajale kuvatakse vastus ilma kogu lehte uuesti laadimata, laetakse ainult vastavate konteinerite uued sisud.

Tavapäraselt on näiteks PHP keeles loodud rakenduse ühe päringu tegemiseks protsess umbes järgmine:

1. Klient küsib serverilt ressurssi, nt http://veebirakendus.ee/asukohad
2. Serveri poole peal rakendus uurib kas ta oskab selle aadressiga (/asukohad) midagi peale hakata
3. Kui oskab, laetakse vastav kontroller ja selle päringu teostamiseks vajalik meetod
4. Täidetakse meetodis olevaid käske
5. Tagastatakse valmis HTML sisu.

SPA rakendustes laetakse kõik HTML sisud ehk vaated, otse JavaScriptist. Kui on vaja küsida mingeid andmeid andmebaasist, siis peab see toimuma läbi HTTP päringute. Nende päringute tegemiseks on vajalik eraldi serveripoolne rakendus, mis tagastabki ainult JavaScriptile loetavat JSONit.

JSON ehk JavaScript Object Notation, on andmete struktureerimise standard. Selle kaudu on võimalik edastada järjestavaid (ingl *serialized*) objekte JavaScriptile loetavas formaadis.

Selliste rakenduste loomisel on projektil kaks osa, mida tihti haldavad hoopis erinevad osapooled. On ainult JavaScripti kliendi pool, mis on mingi lehestik, ja on serveripool, mille ainsaks eesmärgiks on JavaScriptile loetavas formaadis andmete tagastamine.

2.4. Symfony Assetic

Teek on loodud PHP keeles, Symfony raamistiku ühe komponendina ning tema eesmärk on hallata kõiki HTML lehele laetavaid objekte nagu pildid, stiilid ja JavaScript. Sisuliselt on tegemist autori loodud JSMani paralleeliga, kuid lisaks JavaScriptile, suudab see ka töödelda pilte ja stiilifaile. Assetic on palju võimsam ning omab oluliselt rohkem funktsionaalsust, millest osa oli algselt planeeritud ka JSMan-i. JavaScripti poole pealt oskab Assetic järgmist:

1. Lisada faile lehele ühekaupa
2. Lisada faile kaustade kaupa
3. Pakkida failid kokku üheks failiks ja siis serveerida ühe failina
4. Töödelda faile, enne serveerimist, erinevate filtritega (pakkida väiksemaks, uglify jne)

Kuid Symfony Assetic-ut saab kasutada ainult väga spetsiifilistes keskkondades. Failide väiksemaks pakkmiseks ja uglify (kood tehakse väiksemaks ning inimesele loetamatuks) meetoditeks kasutab Assetic Node.js-i komponente, mitte PHPs kirjutatud tööriistu. Kuna enamus virtuaalservereid, mida Eestis ja mujal maailmas saab rentida, ei paku kasutajatele ligipääsu käsureale (või kui pakuvad, siis ei saa sinna oma suva järgi midagi võõrast seadistada), ei ole võimalik ka Node.js komponente kasutada. Autori poolt loodud JSMan kasutab ainult PHP komponente.

```
{% JavaScripts '@AcmeFooBundle/Resources/public/js/*' filter='uglifyjs2' %}
    <script src="{{ asset_url }}"></script>
{% endJavaScripts %}
```

Näide 5. JavaScripti sisaldamine ja uglify filtri lisamine Twig-i mallis.

Autor sai Asseticu olemasolust teada umbes aasta peale JSMan-i loomist, kuigi Assetic tuli välja umbes samal ajal kui JSMan-i esimesed versioonid kasutusse läksid.

3. JSMan

JSMan-i jaoks sai püstitatud üsna suur eesmärk: hallata kõikvõimalikke JavaScriptiga seotud aspekte terves veebirakenduses. Temaga pidi olema võimalik pakkida failid kokku, kasutada *uglify* ja *minify* funktsioone, tulemusena saadud JavaScripti faili pidi veel saama kokku pakkida .gz formaati, et sedasi suurust veel vähendada. Tänapäevased veebilehitsejad on võimelised Gzip formaati ise lahti pakkima ja sealt JavaScripti kasutama. Lisaks sellele, oli oluline, et erinevaid JavaScripti faile saaks lisada PHP kontrollerites, nt vastavalt kasutaja ligipääsudele.

Tähtis oli ka see, et JavaScripti muutujaid saaks lisada läbi PHP, kuid paraku selle funktsionaalsuse juurutamine jäi suuresti tegemata. Ebaõnnestumise põhjuseid on kirjeldatud allpool.

Portaalis Rada7.ee töötab veidi teistsugune kood, mis on veel tihedamalt põimitud Codeigniter-i ja Smarty objektidega. Kuid ülevaate mõttes on välja toodud võimalikult sõltuvustevaba versioon JSMan-ist.

3.1. Klasside loetelu

JSMan koosneb sellistest klassidest:

- jsman.php (JavaScripti haldamiseks mõeldud klass)
- gzipman.php (GNU gzip formaati pakkimiseks mõeldud lihtne adapterklass)
- pakkija klass (seminaritöös kasutati Dean Edwards-i loodud *packer*-it, mis on porditud PHP keelde Nicolas Martin-i poolt (Dean Edwards, 2006))

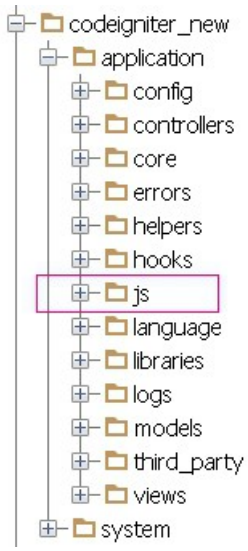
Kuna tegu ei ole PSR-0 ega PSR-4 formaadis klassidega, siis nende asukoht rakenduses sõltub kasutaja enda vajadustest ja soovidest. Kaasasolevas näidisrakenduses (Lisa 1 ja 3) asuvad autori loodud klassid `app/src` kaustas ning pakkijaklass `app/vendor` kataloogis.

3.2. Mida eeldab kasutamiseks, failistruktuur

JSMan ei tööta uutemate nimeruume toetavate raamistikega ilma vahele loodud adapterita (vt 3.4 Implementeerimine), mis aitaks teda kasutada PSR-0 või PSR-4 standardite järgi (PHP

Framework Interop Group, 2014). Üsna lihtsalt integreeritav on ta vanemates ja lihtsamates raamistikes nagu näiteks nagu Codeigniter.

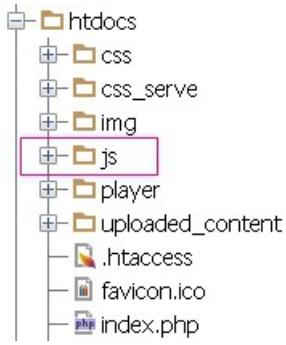
JavaScripti hoidmiseks tuleks tekitada kaust teiste rakenduses olevate failide juurde. Codeigniteris on selleks kaust „application“ kuhu saab luua uue kausta nimega „js”.



Illustratsioon 1: Rada7.ee Codeigniter-i rakenduse kataloogistruktuur

3.3. Avalik pool

Kasutamiseks on vajalik luua avalikult kättesaadav kaust, kuhu sisse hakatakse kirjutama genereeritud faile. Kaust peab olema rakendusele kirjutatav ka, seega peab talle anda õigused CHMOD 777.



Illustratsioon 2: Avaliku poole kaustade struktuur

3.4. Implementeerimine

Teek peab olema initsialiseeritud klassis, kus tegeletakse päise kuvamisega. Lähemalt saab seda uurida lisas olevas rakenduses. Allolev näide eeldab liidestust Smarty ja Codeigniter-iga.

```
<?php
class View extends Smarty
{
    // kuva lehe päis, sisu ja jalus
    public function display_content($template_name)
    {
        $config-> array(
            'default_scripts' => array(
            /**
             * lisa siia nimekiri JavaScripti failinimedest
             * mida alati laetakse kõige enne
             */
            ),
            'js_path' => '', // kausta kus hoitakse faile asukoht, nt
APPPATH.'/js/'
            'js_cache_path' => '', // kaust kuhu salvestakse genereeritud failid
            'js_url_path' => '', // link, kus fail asuma hakkab (siia lisatakse
otsa failinimi)
            'cache_enabled' => true, // aktiveeri faili salvestamine
            'use_packer' => true, // kasuta koodpakkijat
            'gzipman' => $this->gzip // viide GzipMan-ile
        );
        // laeme teegi
        $this->ci->load->library('jsman',$config);

        // käivitame
        $this->ci->jsman->init();

        $this->display('header.tpl');
        $this->display($template_name);
        $this->display('footer.tpl');
    }

    // ....
}
```

Näide 6. Implementeerimine Codeigniter-i ja Smartyga.

3.5. Failide lisamine

Faile saab lisada instantseerimisel, andes kaasa nimekirja alati sisalduvatest failidest (vt eelmist alampeatükki). Näiteks teegid, mida alati kasutatakse on sobivad siia lisamiseks.

Teine variant on lisada faile kontrollerist. Siin võib lisada näiteks kontrollerile vajalikke skripte.

```
<?php
class Page extends CI_Controller
{
    public function index()
    {
        $this->jsman->add('index_scripts.js');

        $this->view->display_content('page.tpl');
    }

    // ...
}
```

Näide 7. Application/js kaustast laetakse index_scripts.js nimeline fail.

Kui kasutaja liigub teatud kontrollerile, kogutakse kokku kõik etteantud skriptid ja loetakse nad kõik kokku üheks suureks failiks.

3.6. Failide kokkuehitamine

Failide kokkuehitamisel kõigepealt liidetakse kõik failinimed ja tehakse sellest räsi (vaikeseadistusena SHA1). Seda kasutatakse faili nimetamiseks. Näiteks laadides failid jquery, index_scripts saadakse räsiks df889807b0c157e4ee1650222ef99eaf06cd4ecf ja faili lõppnimeks saab df889807b0c157e4ee1650222ef99eaf06cd4ecf.js. Seejärel kontrollitakse kas selle nimega fail eksisteerib. Kui ei, liidetakse failisisud kokku, kasutatakse selle sisu peal koodipakkijat ning kirjutatakse sellest uus fail. Loodud fail pakitakse omakorda .gz formaati, et suurust veel vähendada.

Seejärel serveeritakse fail kasutaja veebilehitsejasse. Kui fail on juba olemas, ei tehta midagi. Serveerimise osa seminaritöös ei käsitleta, kuna see on sügavalt seotud konkreetse rakendusega, kus JSMan-i parajasti kasutatakse.

3.7. Mis on halvasti?

3.7.1.Koodi kvaliteet

JSMan on loodud kasutamiseks koos PHP raamistiku CodeIgniteriga ning vaate kihi raamistikuga Smarty. Loomise hetkel ei olnud PHP kogukonnas veel kanda kinnitanud PSR standardid ega ka PHP pakihaldur Composer. Ning ka raamistik millega teek koos töötas ei olnud juba loomise hetkel eriti moodne, ning ei omanud PHP nimeruumide tuge. Selle tõttu on JSMan üsna klassikaline PHP4 tüüpi klass, mida ei saa ilma mõningase ümberkirjutamiseta

hetkel populaarsete raamistikega koos kasutada (nt Laravel, Symfony).

Samas on teda võimalik ümber teha tänapäevaseks. Lihtsamatest sammudest tuleks kõigepealt vahetama konstruktori implementatsioon, lisada nimeruumid ning eemaldada otsesed viited ja sõltuvused koodipakkijale ja GzipMan-ile (GzipMan on autori loodud lihtne rakendus failide pakkimiseks .gz formaati). Nende asemel peaks teeki lisama sõltuvusi (*dependency injection*) Gzip ning ka *uglify* ja *minify* funktsionaalsuse jaoks.

3.7.2.JSMan ja vaate kiht

Selle klassi üks suurimaid puudujääke on võimekus töötada suvalise vaate (*view*) kihiga. PHPs pole väljakujunenud standardeid, mis ühtlustaks vaadete genereerimiseks kasutatavaid raamistikke/teeke. JSMan on algselt loodud töötamaks koos Smarty nimelise vaate kihiga ning nende kahe omavaheline sõltuvus on sügavalt põimitud. Lisas olevas rakenduses on kasutatud samuti liidestust Smarty-ga.

3.7.3.Pakkimisalgoritmide puudumine PHPs ja muud kokkupakimise probleemid

Kahjuks pole autoril õnnestunud saada tööle PHP kaudu *uglify* funktsiooni. Kuna olemasolevad pakkijad ei ole loodud koodianalüüsi silmas pidades, vaid kasutavad enamasti lihtsaid asendusvõtteid, siis genereerivad nad mittetöötavat koodi, kui koodis on vähegi keerulisem JavaScripti struktuur.

Kasutades kaasasolevat pakkimisklassi ei ole alati garanteeritud ka näiteks kommentaaride õige eemaldamine. Lisatud JavaScripti failides ei tohi olla kommentaarid kujul:

```
// kommentaar #lorem ipsum
```

Pakkimise interpretaator ei ole piisavalt tark, et aru saada kuidas see kommentaar eemaldada. Mis tähendab, et enne kolmandate osapoolte poolt loodud failide lisamist, tuleb sellised kommentaarid käsitsi eemaldada. Tõenäoliselt leidub veel kombinatsioone, mis ei sobi.

Kaasasolevas pakkijas on võimalik sättida pakkimise taset, mis paraku suurema läve puhul ei tööta õigesti.

3.7.4.Muutujad

Algselt oli plaanis ka võimaldada teegi kaudu muutujate lisamist, süntaksiga *a la* $this->jsman->assign('variable', 'value'); mida saaks seejärel kasutada JavaScriptis. Probleemiks osutus üsna

selge loogiline viga: kuna peale esimest genreerimist luuakse üks fail, siis dünaamiliselt väärtustatud muutuja tuleks kaasa vaid JavaScripti faili loomise hetkel. Seega peaks iga uue väärtuse tõttu genereerima uue faili, mida saaks ainult teha siis kui sisu iga kord võrrelda. Selle tõttu sai loodud meetod *js_variables()* kuhu sisse võib asetada konstantse väärtusega muutujaid.

Kui tahetakse laadida mingit JavaScripti paketti, kus on oma stiilifailid, siis JSMan suudaks küll laadida JavaScripti ennast, kuid jätaks muutmata kujule CSS failid. Kui nende stiilifailides asuvad pildid, siis nende url võib osutuda valeks. Selle probleemi lahenduseks on näiteks Symfony Assetic-us kasutusel *cssrewrite* filter. (Symfony, 2014)

3.7.5.JavaScripti omavahelised sõltuvused ja failide lisamine

Üks suurimaid probleeme on JavaScripti omavaheline sõltuvus. Nimelt peab järgima failide lisamisel teatud järgnevust, selleks et ei tekiks olukorda, kus kasutatakse koodi mida ei ole veel olemas. JSMani kaudu saab lisada faile igal pool üle rakenduse, mis teeb lisatud failide järgimist veel keerulisemaks. Sedasi võib väga lihtsalt lisada Javascripti faile, mille sõltuvused on puudu. Näiteks Wordpressis on kasutusel süsteem CSS ja Javascript failide haldamiseks, kus lisamisel saab defineerida sõltuvused.

```
wp_enqueue_script('jquery', '/js/jquery.js');
wp_enqueue_script('custom-script', '/js/custom_script.js', array('misc-script'));
wp_enqueue_script('misc-script', '/js/misc_script.js', array('jquery'));
```

Näide 8. Wordpressi lisatud Javascripti failid. Esimene parameeter on alias; teine on asukoht failisüsteemis; kolmas on massiiv sõltvustest (aliastest), mida lahendatakse etteantud järjekorras

Alguses laetakse failid millel ei ole sõltuvusi, siin näites on selleks jQuery. Järgmisena hakatakse lahendama kõrgemaid sõltuvusi. Selles näites on lõplikuks failide järjekorraks: jquery, misc-script, custom-script. Kuigi custom-script on koodis lisatud enne misc-script-i, on tal defineeritud misc-scripti sõltuvus ja seega laetakse ta peale misc-script-i.

Selline võimekus oleks kindlasti JSManile abiks, kuigi see ei annaks endiselt ülevaadet, mis hetkel on juba lisatud ja mis järjekorras. Ainuke viis kuidas sisestatud faile teada saada on kasutades väljakuvamiseks *stockpile()* meetodit.

3.7.6.Failide muutmine

Kui algfaile muudetakse (need failid mida *add()* meetodiga lisatakse kokkuehitamiseks) tuleb genereeritud JavaScripti fail kustutada. Lehe laadimisel genereeritakse siis uus kokkuehitatud

fail. JSMan ei kontrolli kas vahepeal on algfaile muudetud, sest selleks peaks igat faili alati lehe laadimisel lugema. Kui neid faile on väga palju, siis on see lihtsalt ebavajalik lisakulu. Seega tuleb arvestada sellega, et tekitada tuleb mingi kood mis tühjendab loodud failidest vahemälu. Lehekülje uuendamisel genereeritakse uus fail, mida siis edastatakse lõpptarbijale.

3.8. Mis on hästi?

JSMani puhul oli tähtis, et see töötaks haldurina virtuaalserveris ja selles plaanis saab ta oma tööga üsna hästi hakkama. Lisaks on suur eelis, et JSManiga saab genereerida Gzip formaadis faile, mida seejärel on võimalik serveerida kliendile. Tänu sellele saab serveeritava faili mahtu väga oluliselt vähendada. Näiteks portaalis Rada7.ee on näiteks pakkimata faili suurus 256KB ja pakitud faili suurus 85KB. Kui näiteks on üksikuid lehekülgi, kus on vaja kasutada mõnda suuremat JavaScripti raamistikku (nt jQuery UI, Highcharts) siis ei pea laadima neid teeke iga lehega kaasa vaid kasutada ainult üksikutes kohtades.

Kokkuvõte

JSMan töötab alates 2011 aasta aprillist portaalis Rada7.ee. Selle aja jooksul on tulnud välja kasutamise iseärasused, sh. muutujate sättimise võimalus (mis osutus oma olemuselt võimatuks ja lõpuks ka kasutuks), sõltuvuste lisamise puudumine ja laetud failidest ülevaate puudumine.

Üheks oluliseks põhjuseks miks JSMan loodud sai, oli puudus terviku kokkuehitamise võimekusest. Rada7.ee töötab virtuaalserveris, kuhu kasutajal pole õigusi midagi installida ning isegi käsurealt ligi pääseda. Seega pidi rakendus ise olema võimeline end kokku ehitama töötamise ajal.

Teine põhjus oli vajadus hoida *backend* osa ühes kohas, ning mitte asetada rakenduse esmaseks toimimiseks vajalikke faile avalikku kausta (Apache serveris nt htdocs kaust). Sedasi oli teiste, vähem tehniliste inimeste ligipääs neile piiratud. Ühtlasi aitas see koos hoida hädavajalikke JavaScripti faile, võrdsustades nende olulisust.

Wordpressi võidukäik tõestab, et mingil määral on selliste haldurite järgi vajadus olemas, vähemalt nendele inimestele kellel pole ligipääsu rohkem võimalusi pakkuvale serverile. Wordpress ei ehita JavaScripti faile kokku, seal serveeritakse igat faili eraldi. JSMan oleks selles kontekstis isegi mingil määral edumeelsem, eriti kui lisada Wordpressi sõltuvuste lahendamine.

JavaScripti tulevik näib aina helgem, järgmine implementatsioon, EcmaScript 6 näol, toob uuendusi, mis muudavad koodi kirjutamise veel lihtsamaks ja modulaarsemaks. Mitme aasta jooksul kasutatud JSMan ei paku piisavalt hüvesid, et selle kasutamine oleks õigustatud. Lihtsam oleks paigutada rakendusele JavaScripti kokkuehitamise ülesanne läbi Gulp.js-i või Grunt.js-i ja seejärel genereeritud fail ise üles laadida.

Lisaks, üha enam levivad *single-page application* tüüpi rakendused, kus JSMan oleks täiesti kasutu. Rakenduse loomise ajal polnud autor JavaScriptis kirjutatud halduritest kursis, eriti kuna nad olid alles tekkimas ning polnud omandanud laialdast kasutuselevõttu. Kui peaks hakkama täna kirjutama analoogset lahendust, siis on äärmiselt kahtlane, et see oleks tehtud serveri poole pealt.

Kasutatud kirjandus

Joyent Inc., About Node.js. Kasutamise kuupäev 5. aprill 2014, allikas http://nodejs.org/about/

Mozilla Development Network, Inner-browsing extending the browser navigation paradigm. Kasutamise kuupäev 7. aprill 2014, allikas https://developer.mozilla.org/en-US/docs/Inner-browsing_extending_the_browser_navigation_paradigm

Mozilla Development Network, JavaScript Overview. Kasutamise kuupäev 7. aprill 2014, allikas https://developer.mozilla.org/en-US/docs/Web/JavaScript/Guide/JavaScript_Overview

PHP Framework Interop Group, Autoloading standard PSR-0. Kasutamise kuupäev 7. aprill 2014, allikas http://www.php-fig.org/psr/psr-0/

PHP Framework Interop Group, Autoloading standard PSR-4. Kasutamise kuupäev 7. aprill 2014, allikas http://www.php-fig.org/psr/psr-4/

RequireJS, About. Kasutamise kuupäev 7. aprill 2014, allikas http://requirejs.org/

RequireJS, Define a Module. Kasutamise kuupäev 7. aprill 2014, allikas http://requirejs.org/docs/api.html#define

RequireJS, Why AMD? Kasutamise kuupäev 7. aprill 2014, allikas http://requirejs.org/docs/whyamd.html

Symfony, Cookbook > How to Use Assetic for Asset Management. Kasutamise kuupäev 7. aprill 2014, allikas http://symfony.com/doc/current/cookbook/assetic/asset_management.html

Tobie Langel, Lazy evaluation of CommonJS modules. Kasutamise kuupäev 7. aprill 2014, allikas http://calendar.perfplanet.com/2011/lazy-evaluation-of-commonjs-modules/

World Wide Web Consortium (W3C), Scripts. Kasutamise kuupäev 7. aprill 2014, allikas http://www.w3.org/TR/html401/interact/scripts.html

A Perfect Port of Packer to PHP. Kasutamise kuupäev 15. märts 2015, allikas http://dean.edwards.name/weblog/2006/12/packer-php/

Lisad

Lisa 1: Näidisrakenduse git-i repositoorium

https://bitbucket.org/officialkirill/jsman-example-app

Lisa 2: Näidisrakendus kokkupakituna

https://bitbucket.org/officialkirill/jsman-example-app/downloads
Siit valida „Tags” ja laadida alla kõige uuema versiooninumbriga variant

Lisa 3: Töötav näidisrakendus

http://kirill.rada7.ee/jsman-example-app/public