LATVIJAS UNIVERSITĀTE
DATORIKAS FAKULTĀTE

TĪMEKĻA LIETOTŅU ĀTRDARBĪBAS OPTIMIZĀCIJA

MAĢISTRA DARBS

Autors: **Ēriks Šarapovs**
Studenta apliecības Nr.: es10146
Darba vadītājs: Asociētais profesors, Dr. dat. Darja Solodovņikova

RĪGA 2018

// ANOTĀCIJA

Šī darba ietvaros tika apskatīti vairāki veidi, kas ļauj optimizēt tīmekļa lietotņu lejupielādes laiku un klienta puses ātrdarbību.

Tika aplūkota satura optimizācijas tēma, kurā tika apskatītas iespējas samazināt nevajadzīgās lejupielādes, tekstuālu datu apjoma samazināšana, HTTP pieprasījumu un atbilžu kešdarbe, efektīva nepieciešamo resursu pārvalde.

Tika apskatītas darbības, kas ļautu pēc iespējas ātrāk sākt tīmekļa lapu atveidošanu, jeb samazināt kritisko renderēšanas ceļu. Kā arī, tika aplūkoti procesi, kas tiek veikti pie tīmekļa lapu atveidošanas, un iespējas uzlabot šo procesu ātrdarbību.

Darba ievaros tika apskatīta ātrdarbības optimizācija ne tikai no tehniskās puses, bet arī no cilvēku psiholoģiskās laika uztveres viedokļa.

Tika veikta arī praktiska ātrdarbības optimizēšanas metožu pielietošana – datu saspiešana, koda minificēšana, resursu priekšnolase, pieprasījumu optimizācija un JavaScript izpildes optimizācija.

Darbs ir balstīts uz dažādiem literatūras un interneta informācijas avotiem, kas tika izanalizēti, ar mērķi uzzināt par zināmiem veidiem tīmekļa lietotņu ātrdarbības optimizācijā.

Atslēgvārdi: Tīmekļa lietotne, Ātrdarbība, Satura optimizācija, JavaScript, Renderēšanas ātrdarbība, Laika uztvere.

ABSTRACT

Performance optimization of web applications

In this thesis were reviewed different solutions, which allows to improve downloading and client side performance of web applications.

One of themes are related to content optimization, which covers topics about unnecessary download eliminating, caching of HTTP requests and responses, textual data size reducing, effective resource handling.

Also in thesis were discussed about possible actions, which could make faster first-time rendering of web page – improvements of critical rendering path. Besides, processes which involves in web page rendering and opportunities to improve performance of these processes were also covered.

In scope of this thesis performance optimization was reviewed not only from technical perspective, but also from psychological time perception point of view.

Practical usage of performance optimization techniques were also applied – data compression, code minification, resource handling, optimization of requests, and JavaScript execution optimization.

The thesis is based on variety of different literature and Internet information sources, which were analyzed in order to find certain types of web application performance optimization.

Key words: Web application, Performance, Content optimization, JavaScript, Rendering performance, Time perception.

AUTOREFERĀTS

Maģistra darba ietvaros autors izpētīja metodes, ar kuru palīdzību iespējams optimizēt ātrdarbību tīmekļa vietņu klienta daļā. Aplūkotās metodes ietver sevī gan tīri tehniskus risinājumus, gan risinājumus, balstītus uz cilvēku laika uztveri.

Tika apskatīta arī tīmekļa lietotņu ātrdarbības nozīme un aktualitāte, balstoties uz datiem un faktiem.

Darba autors uzskata, ka, darba ietvaros apskatītās metodes ir aprakstītas pietiekamā detalizācijas līmenī, lai tīmekļu vietņu izstrādātājiem kļūtu skaidrs, kāpēc šīs metodes vajadzētu pielietot, un, kāds labums no tā tiks iegūts.

Aplūkotā informācija par cilvēku laika uztveri, sniedz iespēju plašāk skatīties uz ātrdarbības optimizācijas metožu pielietošanu, kā arī ļauj apskatīt ātrdarbību no cita skatu punkta.

Maģistra darba ietvaros tika veikta arī praktiska, vairāku atrasto un izpētīto ātrdarbības optimizēšanas metožu pielietošana. Tika pielietota datu saspiešana, kuras rezultātā, izmaiņas datu izmēros, bija līdzīgas izmaiņām, kas tika minētas izmantotajos informācijas avotos. Tika pielietota koda minificēšana, resursu priekšnolase, pieprasījumu optimizācija un JavaScript izpildes optimizācija.

Darba izstrādei tika izmantoti 42 dažādi informācijas avoti – 10 grāmatas un 32 tiešsaitē pieejamie raksti.

Šīs darbs tika vairākkārt pārlasīts, atrastās valodas kļūdas tika izlabotas. Kļūdu meklēšanai papildus tika izmantota teksta redaktorā esošā valodas pārbaudes programmatūra.

Darba autors ir pārliecināts, ka darbā nav atrodams plaģiāts, jo visiem izmantotajiem informācijas avotiem, no kuriem tika aizgūtas idejas, attēli, u.c., ir norādītas atsauces.

SATURA RĀDĪTĀJS

APZĪMĒJUMU SARAKSTS 7

IEVADS 9

1. ĀTRDARBĪBAS NOZĪME TĪMEKĻA VIETNĒS 10

2. SATURA OPTIMIZĀCIJA 13

2.1. Izvairīšanās no nevajadzīgajām lejupielādēm 13

2.1.1. Attēlu kartes 13

2.1.2. CSS Spraiti 14

2.1.3. Iekļautie attēli 14

2.1.4. JavaScript un stila lapu kombinēšana 15

2.1.5. Nevajadzīgā satura izmešana 16

2.2. Tekstuāla datu apjoma samazināšana 17

2.2.1. Koda minifikācija 17

2.2.2. Datu saspiešana 17

2.3. HTTP pieprasījumu un atbilžu kešdarbe 19

3. KRITISKAIS RENDERĒŠANAS CEĻŠ 21

3.1. Objektu struktūra 22

3.2. Atveidošanas koka izveide 24

3.3. Kritiskā renderēšanas ceļa optimizācija 24

4. RENDERĒŠANAS ĀTRDARBĪBA 26

4.1. JavaScript izpildes optimizācija 27

4.2. Stilu skaitļošanas optimizācija 28

4.3. Izvairīšanās no lieliem un sarežģītiem izkārtojumiem 29

4.3.1. Flexbox modelis 29

4.3.2. Piespiedu sinhronizācija ar izkārtojumu 29

4.4. Zīmēšanas procesa optimizācija ... 30
4.5. Izmaiņas kompozīcijā ... 30
5. RESURSU PĀRVALDE ... 31
5.1. DNS priekšienese ... 31
5.2. Priekšsavienošana ... 33
5.3. Priekšnolase ... 33
5.4. Priekšattēlošana ... 34
5.5. Priekšlejupielāde ... 34
5.5.1. Apslēpto resursu priekšlejupielāde ... 35
5.5.2. Skripta resursu priekšlejupielāde bez to izpildīšanas ... 35
5.5.3. Resursu priekšlejupielāde balstoties uz CSS @media noteikumiem ... 36
6. ĀTRDARBĪBAS OPTIMIZĀCIJA BALSTOTIES UZ LAIKA UZTVERI ... 38
6.1. Objektīvais laiks ... 38
6.1.1. Atšķirības slieksnis ... 38
6.1.2. Laika uztveres gradācija no psiholoģijas viedokļa ... 39
6.2. Subjektīvais ... 40
6.2.1. Apsteidzošs sākums ... 40
6.2.2. Priekšlaicīga pabeigšana ... 41
6.2.3. Citi paņēmieni ... 41
7. DARBA PRAKTISKĀ DAĻA ... 43
7.1. Satura optimizācija ... 45
7.2. Resursu pārvalde ... 49
7.3. Pieprasījumu optimizācija ... 50
7.4. JavaScript optimizācija ... 52
REZULTĀTI UN SECINĀJUMI ... 54
IZMANTOTĀ LITERATŪRA UN AVOTI ... 56

PIELIKUMI 60
1. pielikums 61
2. Pielikums 62

APZĪMĒJUMU SARAKSTS

HTML	(HyperText Markup Language) Iezīmēšanas valoda, izmanto tīmekļa vietņu veidošanā. Satur pārlūkprogrammā attēlojamo informāciju.
JavaScript	Dinamiski interpretējamā programmēšanas valoda, plaši izmantota tīmekļa balstītos risinājumos.
jQuery	Viena no visplašāk izmantotajām JavaScript bibliotēkām. Atvieglo darbu ar JavaScript.
CSS	(Cascading Style Sheets) Stila lapu valoda, paredzēta iezīmēšanas valodā veidotu dokumentu izskata aprakstam.
DOM	Dokumentu objektu modelis. Definē dokumenta struktūras loģiku, un veidus, kādos var tikt veikta manipulācija ar dokumentiem.
CSSOM	CSS objektu modelis. Definē saskarni informācijas līdzekļu pieprasījumiem, CSS selektoriem un pašam CSS.
HTTP	Hiperteksta pārsūtīšanas protokols. Tiek izmantots globālajā tīmeklī. Tas definē saziņu starp serveriem un pārlūkprogrammām.
URL	Vienotais resursu vietrādis, jeb tīmekļa adrese. Norāda uz tīmekļa resursu atrašanās vietām.
JSON	(JavaScript Object Notation) Sintakse datu glabāšanai un apmaiņai.
XML	Iezīmēšanas valoda. Atšķirībā no HTML ir iespējams veidot savas birkas, kā arī var saturēt atsauces uz vairākiem dokumentiem.
DNS	Domēnu nosaukumu sistēma – domāta, lai pēc domēna vārda, varētu atrast tā IP adresi.
TCP	Pārraides vadības protokols – paredz savienojumu izveidošanu starp sistēmām un nodrošina drošas datu pārraides mehānismus.
TLS	Transporta slāņa drošība – protokols, kurš nodrošina privātumu un drošību, lietotņu komunikācijai tīmeklī.
TypeScript	Microsoft izstrādāts JavaScript valodas papildinājums, kurš ļauj izmantot tipizāciju izstrādes laikā.

Webpack	Koda moduļu sasaistītājs, galvenokārt paredzēts JavaScript datņu apvienošanai koda saišķos.
React	Facebook izstrādāta JavaScript bibliotēka, kas paredzēta lietotāju grafiskās saskarnes veidošanai.
Redux	JavaScript bibliotēka, kas paredzēta lietotņu stāvokļu pārvaldei.
Lodash	Viena no populārākajiem utilītprogrammām, kas atvieglo JavaScript koda izstrādi.
Highcharts	Plaši izmantota JavaScript bibliotēka, kas paredzēta grafiku un diagrammu veidošanai.

IEVADS

Ar katru gadu Internets kļūst arvien plašāks un dziļāks, strauji paplašinās e-komercija, līdz ar telekomunikāciju attīstību pieaug arī Interneta lietotāju skaits, kuriem, agrākās limitētās Interneta datplūsmas vietā, parādās ar vien lielākas iespējas izmantot bezlimita datplūsmu, lai piekļūtu Interneta resursiem.

Neskatoties uz to ka gan servera puses aparatūra, gan lietotāju skaitļošanas iekārtas kļūst jaudīgākas, gan uz to ka uzlabojas komunikācijas ātrums un atbildes laiks – tīmekļa vietnēm ir tendence kļūt ar vien lēnākām [3][4].

Līdz ar tehnoloģiju attīstību parādījās iespēja veidot komplicētas tīmekļa vietnes, ar bagātīgu saturu, sarežģītu struktūru, kas būtu interaktīvas attiecībā pret lietotāju. Rezultātā palielinās tīmekļa vietņu izmērs, ielādējamo resursu skaits un izmērs, mijiedarbība starp tīmekļa vietnes komponentēm kļūst sarežģītāka, apstrādājamās informācijas kopas kļūst komplicētākas un apjomīgākas. Šie faktori arī noved pie problēmām ar ātrdarbību.

Salīdzinājumā ar 2010. gadu, tīmekļa vietņu izmērs ir palielinājies gandrīz 4 reizes, kur lielāko daļu aizņem attēli [3][4][5].

Lielākā ietekme uz ātrdarbību ir klienta daļas arhitektūrai. Tikai 5-20% no laika, kas nepieciešams pilnīgai tīmekļa vietnes lapas lejupielādei, aizņem HTML dokumenta iegūšana no servera [1][6].

Šī darba ietvaros uzmanība tiks veltīta tīmekļa vietņu izmēra un klienta daļas optimizācijai, ar mērķi uzlabot tīmekļa vietņu ātrdarbību. Tiks pētīti procesi, kas ietekmē tīmekļa vietņu ātrdarbību, un meklēti varianti, kā optimizēt šos procesus. Tiks aplūkoti veidi, ar kuru palīdzību varētu efektīvi pārvaldīt resursus, kas nepieciešami tīmekļa lietotnes darbībai. Tiks pētīta arī cilvēku laika uztvere no psiholoģijas viedokļa.

Darbs sastāv no septiņām nodaļām. Pirmajā nodaļā tiek aprakstīta tīmekļu lietotņu ātrdarbības nozīme un aktualitāte. Otrajā nodaļā tiek apskatīti veidi tīmekļa vietņu satura optimizācijai, ar mērķi, samazināt satura izmēru. Trešajā nodaļā tiek apskatītas lietas, kas saistītas ar kritisko renderēšanas ceļu. Ceturtajā nodaļā tiek apskatīti procesi, kas ietekmē renderēšanas ātrdarbību pārlūkprogrammās. Piektajā nodaļā tiek runāts par tīmekļa darbībai nepieciešamo resursu pārvaldes metodēm. Sestajā nodaļā tiek aplūkota cilvēku laika uztvere no psiholoģijas viedokļa. Septītajā nodaļā tiek aprakstītas, darba ietvaros veiktās ātrdarbības optimizācijas praktiski.

1. ĀTRDARBĪBAS NOZĪME TĪMEKĻA VIETNĒS

Ātrdarbība tīmekļa vietnēm ir svarīga galvenokārt komerciāliem projektiem. 2010. gadā tika veikta aptauja pēc kuras datiem, 57% e-veikalu klientu aizvērs lapu, kura lejupielādējas ilgāk par 3 sekundēm [13]. Pēc citiem datiem, lietotājs, kuram lejupielāde bija jāgaida astoņas sekundes, tikai vienu procentu no kopējā apskates laika patērē uz lapas nozīmīgākajām daļām, bet, ja lapas lejupielāde ir bijusi ļoti ātra - 20% laika. Kā arī, vietnei, kas lejupielādējas trīs sekundes, salīdzinājumā ar vietni, kura ielādējās vienas sekundes laikā, ir par 22% mazāks lappušu apmeklējums, par 50% lielāks atteikušos lietotāju skaits, un par 22% mazāks tālāko pāreju skaits uz citām tīmekļa lapām. Savukārt, vietnei, kas lejupielādējas piecas sekundes, šie skaitļi ir vēl lielāki - par 35% mazāk apmeklējumu, par 105% lielāks atteikušos lietotāju skaits, un par 38% mazāks pāreju skaits uz citām vietnes lapām, salīdzinājumā ar vietni, kas ielādējās vienas sekundes laikā [5].

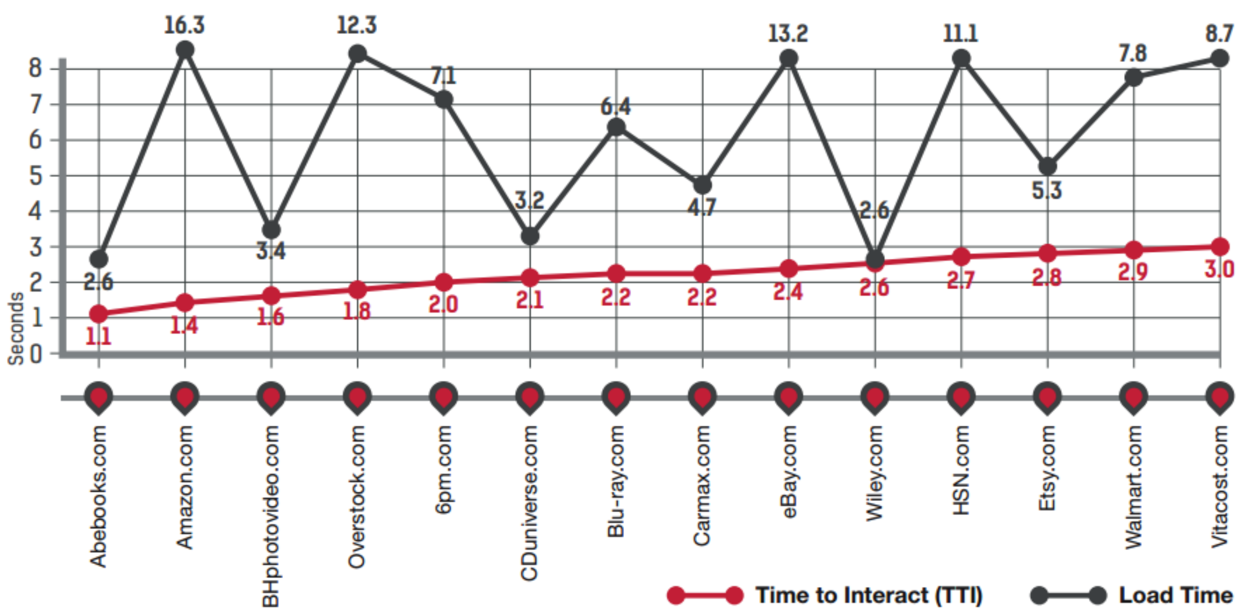
Ņemot vērā šos faktus, ir vērts aplūkot radware[11] salasītos datus par 2015. gadu. Statistikai tika izmantoti 100 populārākās e-komercijas vietnes, un tika iegūti sekojoši dati:

- Vidējais laiks līdz momentam, kad lietotājs varēja sākt darboties vietnē, sastādīja 5.2 sekunde, kas ir vairāk par iepriekšminētajām trīs sekundēm;
- Tikai 14% no aplūkotajām vietnēm atveidoja saturu, laikā mazākā par trīs sekundēm;
- Deviņiem procentiem vietņu bija nepieciešamas vairāk kā 10 sekundes, lai ar tām varētu sākt darboties;
- Sliktākais laika rādītājs bija 25.1 sekunde;
- Vidējais lapas izmērs bija 1354KB un vidēji saturēja 108 resursu pieprasījumus. Gandrīz visos gadījumos lapas ar lielāku izmēru un lielāku pieprasījumu skaitu ielādējās lēnāk;
- Attēli sastāda no 50 līdz 60% lapas izmēra, taču 43% pārbaudītajām vietnēm netika pielietota attēlu saspiešanas metode, 10% ieguva augstāko vērtējumu par attēlu saspiešanu.

Ne tikai lapu izmēram ir ietekme uz ātrdarbību, ne mazāk svarīgs ir lejupielādējamo resursu daudzums. Katrs lapai nepieciešamā resursa pieprasījums veic riņķveida ceļojumu no pārlūkprogrammas uz serveri, un tad atbilde atgriežas uz pārlūkprogrammu. Lai arī viens šāds pieprasījums vidēji aizņem 65-145 milisekundes, kopējais laiks strauji pieaug, kad runa iet par 100 un vairāk resursu pieprasījumiem [6].

Runājot par tīmekļa vietņu lapu lejupielādi, ir jāizdala 2 mērvienības:

1. Pilnas lejupielādes laiks - laiks, kurā pilnīgi visi nepieciešamie resursi tiek ielādēti un atveidoti. Pie šiem resursiem ir pieskaitāmi attēli, palīg skripti un trešo pušu skripti;
2. Laiks līdz vietnes izmantošanai - tīmekļa vietnei nepieciešamais laiks, kas nepieciešams lai ielādētu un atveidotu pašu galveno saturu, ar kuru lietotājs jau var sākt darboties.



1.1.att. **Ātrāko e-komercijas tīmekļa vietņu lejupielāde [11]**

Attēlā 1.1. melnā līkne norāda uz pilnas lejupielādes laiku, savukārt sarkanā līkne – laiku, kas nepieciešams, līdz vietni ir iespējams izmantot.

No lietotāja viedokļa, laiks, kurš nepieciešams, līdz būtu iespējams izmantot tīmekļa vietni, ir nozīmīgāks par pilnas lejupielādes laiku, par cik lietotāju īsti neinteresē fonā notiekoši procesi, kurus viņš nepamana.

Liela nozīme ātrdarbībai ir vietnēs, kurās tiek veiktas daudzas izmaiņas DOM struktūrā. Parasti, veikt manipulācijas ar nelielu DOM elementu kopu nerada problēmas, bet, kad vienlaicīgi tiek mainīti uzreiz daudzi elementi, rodas ātrdarbības problēmas, un ir stipri jāpiedomā par veidiem

tās risināt, galvenokārt tas skar CSS un JavaScript kodu. Tas ir jāveido tā, lai pēc iespējas mazāk noslogotu pārlūkprogrammas darbību.

Nereti rodas problēmas tīmekļa vietnēs, kurās lielā daudzumā tiek izmantotas animācijas, it īpaši tas skar mobilās iekārtas – lai animācija būtu plūstoša, ir jānodrošina, lai pārlūkprogramma spētu rādīt 60 kadrus sekundē.

Sistēmām, kuras strādā ar lieliem datu apjomiem, dati no servera uz klientu tiek sūtīti diezgan ilgi, arī šādām tīmekļa lietotnēm liela nozīme ir klienta puses optimizācijai, par cik, ar dažādiem vizuāliem efektiem var panākt šķietami ātrāku tīmekļa lietotnes ātrdarbību.

2. SATURA OPTIMIZĀCIJA

Kā jau tika minēts ievadā, tīmekļa lietotņu izmēri un pieprasāmo resursu daudzums palielinās ar katru gadu. Tas ir izskaidrojams ar to, ka tīmekļa lietotnes arvien vairāk aizstāj klasiskās darbvirsmas programmas, kā ari daudzi jauni risinājumi tiek veidoti tieši kā tīmekļa lietotnes, un paplašinoties šo lietotņu funkcionalitātei, būtiski pieaug nepieciešamie resursi un to izmēri, kas varētu atbalstīt šo funkcionalitāti. Kopumā šo resursu izmēra apjoms ātri paliek mērāms megabaitos, bet tos ir jāiegūst pēc iespējas ātrākā laikā. Tieši tāpēc satura optimizācija ir kritiska.

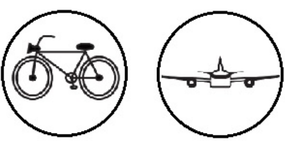
Šajā nodaļā tiks apskatīti tādi optimizācijas veidi kā HTTP pieprasījumu skaita samazināšana, tekstuālu datu saspiešana, koda minifikācija, HTTP pieprasījumu un atbilžu kešdarbe. Būtiskas lietas, runājot par satura optimizēšanu, ir attēlu un tīmekļa fontu optimizācija, bet šī darba ietvaros tās aplūkotas netiks.

2.1. Izvairīšanās no nevajadzīgajām lejupielādēm

Iedeja ir gaužām vienkārša – lai uzlabotu lietotnes atbildes laiku, ir jāsamazina nepieciešamo resursu skaits – tādā veidā samazinot HTTP pieprasījumu skaitu. Diemžēl, daudzos gadījumos tas nozīmē, ka jāmeklē kompromiss starp ātrdarbību un lietotnes dizainu, bet tomēr eksistē tehnikas, kas ļauj izbēgt šī kompromisa tehnikas.

2.1.1. Attēlu kartes

Ideja ir tāda, ka vairāki atsevišķi, parasti nelieli attēli tiek apvienoti vienā lielākā attēlā, un, attēla dažādās vietās tiek izveidotas saites uz dažādiem resursiem un/vai darbībām.



2.1. att. **Attēlu kartes piemērs**

```
<img src="icons.jpg" width="296" height="136" alt="Icons" usemap="#iconsmap">
<map name="iconsmap">
  <area shape="circle" coords="68,68,60" alt="Bicycle" href="bicycle.htm">
  <area shape="circle" coords="230,68,60" alt="Plane" href="Plane.htm">
</map>
```

2.2. att. **Attēlu kartes koda piemērs**

Attēlos 2.1. un 2.2. ir redzams piemērs klientu puses attēlu kartes pielietošanai. Piemērā esošais kods nodrošina to, ka noklikšķinot uz velosipēda ikoniņu tiks atvērts resurss “bycicle.htm”, bet noklikšķinot uz lidmašīnu – “plane.htm”. Par cik vienā attēla ir uzreiz 2 ikoniņas, HTML pieprasījumu samazināsies no diviem, līdz vienam.

Attēlu karšu manuāla izmantošana ir garlaicīga nodarbe, turklāt iespējams pieļaut kļūdas, bet par laimi tīmeklī ir atrodami bezmaksas rīki, kas šo procesu atvieglo. Šāds rīks ir, piemēram “Image Maps” [14]. Un, jāsaka ka CSS spraiti tiek izmantoti daudz biežāk.

Jāmin, ka attēlu kartes var būt servera puses, bet tie tiek izmantoti diezgan reti. Iemesls tam ir tāds, ka servera puses attēlu karšu izmantošanai, pārlūkprogrammai ir jāveic papildus vaicājums serverim, kurā tiek nosūtītas koordinātes, kur tika veikts klikšķis, un tad, serveris sniedz atbildi, kā tālāk rīkoties. Papildus vaicājam ir nepieciešams laiks. Kā arī, klientu puses attēlu kartes ir vieglāk realizējamas [27].

2.1.2. CSS Spraiti

Līdzīgi kā attēlu kartēs, arī šī tehnika paredz vairāku attēlu apvienošanu vienā. Princips ir tāds, ka attēls tiek iestatīts kā fons kādam DOM elementam. Šis DOM elements ir mazāks par pašu attēlu, un tad, mainot fona attēla pozīciju, tiek rādīts vajadzīgais attēla fragments. Attēls tiek bīdīts ar CSS “background-position” īpašību.

Ātrdarbības ziņā, CSS spraiti ir gandrīz tikpat ātri kā attēlu kartes, bet to izmantošana ir lokanāka salīdzinājumā ar attēlu kartēm. Apvienotiem attēliem, izņemot mazāku skaitu HTTP pieprasījumu ir vēl viena priekšrocība. Vairāki attēli apvienoti vienā lielākā, šķietamu varu būt lielāki izmēra ziņā, jo starp apvienotajiem attēliem ir jābūt nelielām atstarpēm, bet praksē, viena liela attēla izmērs sanāk mazāks nekā daudzu mazu attēlu kopējais izmērs [1].

2.1.3. Iekļautie attēli

Tīmekļa vietnēs ir iespēja izmantot iekļautos attēlus izmantojot URL shēmas, ļaujot samazināt HTTP pieprasījumu skaitu. Šīs tehnikas plašākais pielietojums ir izmantojams kopā ar

CSS – URL shēma tiek pielikta klāt “background-image” īpašībai. 2.3. attēlā ir redzams CSS koda piemērs, kurš izveido attēlā redzamo sarkano zvaigznīti.

★

```
.test {
    background-image: url(data:image/gif;base64,R0lGAAAAAAAAAAAAACH5BAEAAAsALAAAAAAMA...
    width:12px;
    height:12px;
}
```

2.3. att. **Iekļautais attēls izmantojot URL shēmu un CSS**

Priekšrocība šādiem attēliem ir tāda, ka tie tiek saglabāti kešatmiņā līdz ar ārējo CSS kodu, un var tikt atkārtoti izmantoti izmantojot CSS selektorus [1] [8].

Ir iespēja pārveidot binārus attēlus 64 skaitīšanas sistēmā, un izmanot tos kā iekļautos attēlus. Parasti tas notiek servera pusē, kur tiek ņemts kāds attēls no lokālās sistēmas un tad tiek nokodēts. Tiesa tas prasa papildus servera puses kalkulācijas. Zemāk esošajā attēlā ir šāds piemērs izmantojot PHP valodu.

```
.home { background-image: url(data:image/gif;base64,
   <?php echo base64_encode(file_get_contents("../images/home.gif")) ?>);}
```

2.4. att. **Lokāla attēla pārveidošana un izmantošana pie URL shēmām [1]**

Jāmin, ka šai tehnikai ir vairāki ierobežojumi. URL shēmas netiek atbalstītas pārlūkprogrammā Internet Explorer 7 un vecākās. Diez vai tas varētu būt īpaši aktuāli, bet prātā tomēr ir jāpatur. Otra lieta – ierobežojums URL simbolu skaitam gan pārlūkprogrammās, gan no serveru puses. Treškārt, Attēli nokodēti 64 skaitīšanas sistēmā ir lielāki par bināriem attēliem aptuveni par 33%. Tieši tāpēc šī tehnika galvenokārt tiek izmantoti maziem attēliem [8].

2.1.4. JavaScript un stila lapu kombinēšana

Līdzīgi, kā ar attēlu kartēm un CSS spraitiem, arī šajā gadījumā ideja ir apvienot vairākas atsevišķas daļas vienā. Ideālā gadījumā tīmekļa lapai vajadzētu būt vienai stilu lapai un vienam JavaScript koda failam.

Sākotnēji tas šķiet pretrunā labās programmēšanas praksei, kur kods tiek sadalīts moduļos, un katrs modulis tiek glabāts atsevišķā datnē. CSS gadījumā kods bieži vien tiek dalīts pa loģiskām

daļām – sekcijām uz kurām tas attiecas. Tādā veidā tiek veikti daudzi HTTP pieprasījumi, lai piekļūtu nepieciešamajam kodam, tādā veidā stipri ietekmējot tīmekļa vietnes ātrdarbību.

JavaScript un stila lapu kombinēšana nebūt nenozīmē, ka tagad viss ir jāraksta vienā failā. Vairāki koda gabali no dažādiem koda failiem tiek savienoti servera pusē, piemēram, kad kādā no failiem tika veiktas izmaiņas un jauninājumi tiek iesūtīti repozitorijā, serveris veic nepieciešamo apvienošanu. Tiesa, apvienot pilnīgi visus koda moduļus vienā datnē arī nebūtu īsti pareiza pieeja, par cik, katrai no lapām var būt vajadzīgi tikai daži moduļi, vēl citai lapai – pavisam cita moduļu kopa. Tas prasa dziļu analīzi, par cik, ar katru datni iespējamo kombināciju skaits strauji pieaug. Jāmin, ka koda apvienošanas solī, ir iespējams veikt arī koda minifikāciju, par kuru tiks darba 2.2.1. nodaļā.

JavaScript un stila lapu kombinēšana kļūst īpaši aktuāla ar JavaScript ietvaru izmantošanas pieaugumu. Daudzi šādi ietvari būvē visu HTML struktūru klienta daļā. Un, bieži vien, katram elementam ir sava atsevišķs JavaScript koda datne un CSS datne. Tā, piemēram, pogai būs 2 datnes, tabulai vēl 2 datnes, un tml. Parasti šādos ietvaros šādu elementu datnes jau ir apvienotas bibliotēkās, kur daudzu šādu elementu kods ir apvienots. Bet, šim momentam obligāti vajadzētu pievērst uzmanību.

2.1.5. Nevajadzīgā satura izmešana

Šis punkts ir diezgan strīdīgs, par cik tas nav realizējams izpildot kādus konkrētus soļus, jo nevar gluži vienkārši pateikt kas no dotā satura ir vajadzīgs, un kas nē. Pietam tas nozīmē izmaiņas tīmekļa lapas dizainā.

Labs piemērs šim gadījumam ir tā saucamais attēlu karuselis, kurš satur vairākus attēlus, un lietotājs tos var nomainīt vienu pēc otra ar navigācijas pogu palīdzību. Vairāki attēli nozīmē papildus nepieciešamos resursus, bet vai tiešām tas ir tā vērts? Šādiem gadījumiem vajadzīga analīze, vai lietotāji izmanto kādu konkrētu satura daļu. Ir liela iespējamība, ka kāda no satura daļām lietotājus īsti neinteresē, un, attēlu karuseļa piemērā, daudzi vairāki attēli tiek lejupielādēti velti, un absolūti lielākā daļa lietotāju tos nekad nemaz neapskata.

2.2. Tekstuāla datu apjoma samazināšana

Šajā nodaļā tiks aplūkotas tehnikas, ar kuru palīdzību var samazināt tekstuāla satura datņu izmēru. Runa ies par tādām tehnikām kā saspiešana un minifikācija. Šo tehniku pielietošana galvenokārt attiecas uz izstrādāto kodu.

2.2.1. Koda minifikācija

Koda minifikācija sevī ietver atstarpju un tabulāciju dzēšanu, tiek likvidēta koda pārnese jaunās rindās un tiek dzēsti komentāri, tiek dzēsts kods, kurš atkārtojas, piemēram, divu vienādu CSS selektoru gadījumā, kuru ķermenī aprakstītas dažādas īpašības, tiks apvienots zem viena selektora.

Šāds kods izstrādātājam praktiski nav salasāms, bet tam arī nav jābūt. Izstrādātāji strādā ar kodu kas iepriekš netika minificēts. Minifikācijas tiek pielietota pirms datnes tiek novietotas uz servera, un minificētās versijas nonāk lietotājiem pārlūkprogrammā. Minifikācijas procesu ir vērts automatizēt, un veikt kad tiek iesūtīti kādu datņu atjauninājumi. Ideālā gadījumā, pie izmaiņām, automātiski tiek veikta koda minifikācija, kombinēšana un saspiešana.

JavaScript minifikācijas gadījumā, vēl ir iespējams veikt funkciju un mainīgo pārsaukšanu. To garie nosaukumi parasti tiek nomainīti pret dažu, viena vai dažu simbolu garu virknīti. Tādā veidā JavaScript datnes apjomu ir iespējams samazināt vēl vairāk.

Atkarībā no esošā koda daudzuma, minifikācijas tehnikas pielietošana ļauj būtiski samazināt datņu izmēru. Par labu piemēru šeit kalpo plaši izmantotā jQuery bibliotēka. jQuery versijas 3.1.1 sākotnējais izmērs ir 260KB, savukārt tās saspiestā versija ir tikai 84.6KB liela.

2.2.2. Datu saspiešana

Runājot par datu saspiešanu, šī darba ietvaros, galvenais akcents tiks likt uz tehnoloģiju gzip.

Līdz ar HTTP/1.1 pārlūkprogrammās parādījās iespēja norādīt saspiešanas veidu, HTTP galvenē norādot "Accept-Encoding" īpašību, piemēram, Accept-Encoding: gzip, deflate. Ja serveris redz šādu īpašību pieprasījumā galvenā, tas var saspiest datnes norādītajos veidos, un, saspiešanas gadījumā, atbildes galvenei pieliek īpašību "Content-Encoding", piemēram, Content-Encoding: gzip [1][6].

Gzip saspiešanas tehnoloģija ir visvairāk izmantotā, to atbalsta praktiski visas pārlūkprogrammas. Gzip tehnoloģija balstās uz algoritmu DEFLATE. Izmantojot gzip tehnoloģiju, rodas iespēja saspiest HTML, CSS un JavaScript datnes [1][6].

Galvenās gzip priekšrocības ir tādas, ka tas ir viegli konfigurējams – atliek ierakstīt dažas rindiņas servera konfigurācijā. Otrkārt, gzip izmantošana praktiski nenoslogo pārlūkprogrammas, un saspiestās datnes atjaunošana notiek ļoti ātri. Papildus noslodze var būt uz serveri, ja saspiešana notiek dinamiski reālā laikā, bet parasti ir pietiekoši failus saspiest statiski, piemēram, kad tie tika izmainīti [6][9].

Ir vērts pielietot gzip saspiešanu HTML, CSS un JavaScript datnēm lielākām par 1KB. Gadījumā, ja HTTP atbilde ir XML vai JSON formātā, un līdzīgi kā iepriekš minētajām datnēm , izmērs ir lielāks par 1KB tad vērts saspiest arī to. Tiesa, lai saspiestu šādas atbildes, nāksies pielietot dinamisko saspiešanu, un tas prasīs papildus servera procesora resursus [1][6][9].

Saspiešanu nevajadzētu pielietot attēliem un PDF datnēm, jo šīs datnes jau ir saspiestas, tāpēc to saspiešana tikai velti tērēs servera procesora resursus, turklāt šīs datnes pēc saspiešanas var kļūt lielākas [1][6][9].

2.1. tabula

Saspiešanas izmēri pielietojot gzip [9]

Domēns	**Sākotnējais izmērs KB**	**Izmērs KB ar gzip**	**Saspiešanas koeficients %**
Google.com	13.7	5.2	62
Facebook.com	30.9	9.6	69
Youtube.com	67.1	16.3	76
Yahoo.com	161.0	39.5	75
Baidu.com	6.3	3.0	52
Wikipedia.org	46.9	11.7	75
Twitter.com	44.1	9.4	79
Amazon.com	138.9	25.6	82

Jāmin, ka pirms koda datņu saspiešanas, lietderīgi ir sākumā veikt koda minifikāciju, un kombinēšanu, tādā veidā, palielinot saspiešanas efektivitāti vidēji par 5% [6].

Eksistē arī citas datu saspiešanas tehnoloģijas [6]:

- deflate – līdzīgi, kā gzip, ir balstīts uz DEFLATE algoritmu, bet tam nav galvenes un metadatu. Tomēr gzip ir atbalstīts lielākā skaitā serveru un pārlūkprogrammu;
- Compress – vēl viena teksta saspiešanas tehnoloģija. Darbības ziņā ir ātrāks par gzip, bet saspiešanas koeficients ir būtiski zemāks, tāpēc daudzas pārlūkprogrammas to neatbalsta;
- bzip2 – saspiešanas tehnoloģija, kuras saspiešanas koeficients ir augstāks par gzip, bet tas ir būtiski lēnāks un tam ir vajadzīgs vairāk procesora resursu. Tiek slikti atbalstīts pārlūkprogrammās.

2.3. HTTP pieprasījumu un atbilžu kešdarbe

Kešdarbe ir viens no kritiskākajiem aspektiem, kad runa ir par tīmekļa lietotņu ātrdarbību. Pārlūkprogrammas tiecas saglabāt pēc iespējas vairāk informācijas, kuru varētu izmantot atkārtoti. Pareiza pieprasījumu un atbilžu kešdarbe būtiski ļauj ietaupīt laiku samazinot nepieciešamo laiku resursu piekļuvei, par cik to ieguve no kešatmiņas ir nesalīdzināmi ātrāka uz datu pārraides, caur tīmekli, fona. Bez tam, resursu piekļuvei, kas glabājas kešatmiņā, lietotājam nav vajadzīga papildus izdevumi datu pārraidei, kas var būt būtiski ierobežotas datu pārraides gadījumā.

Kešdarbes mehānisms ir ieviešams samērā vienkārši. Serverim, sniedzot atbildi uz kādu no pieprasījumiem, ir iespēja pievienot atbildes galvenē instrukcijas par pieprasījuma atbildes saglabāšanu kešatmiņā. Pārlūkprogramma šīs instrukcijas saprot, un pēc iespējas tās mēģina ievērot.

Pirmā svarīgākā lieta, runājot par HTTP pieprasījumu kešdarbi, ir korekta instrukciju nodošana par atbildes glabāšanu kešatmiņā. Tas notiek ar īpašības "Cache-Control", palīdzību, kura tiek pievienota atbildes galvenē. Šīs īpašības vērtība nosaka kādā veida, un cik ilgi dotā atbilde var glabāties kešatmiņā. Īpašībai "Cache-Control" var norādīt sekojošas vērtības [10]:

- "no-cache" – norāda, visi pieprasījumi uz šo saiti sākumā ir jāpārbauda servera pusē, vai atbilde nav mainījusies;
- "no-store" - norāda, ka dotā atbilde nevar tikt glabāta kešatmiņā;
- "max-age" – norāda laiku sekundēs, uz cik ilgu laiku atbilde var tikt saglabāta kešatmiņā no pieprasījuma brīža;
- "public" – norāda, ka atbilde var tikt brīvi glabāta kešatmiņā, izmanto reti, jo, izmantojot "max-age", pēc noklusējuma atbilde tiek interpretēta kā "public";

- “private” – atšķirība ir tā, ka atbilde var tikt glabāta tikai lietotāja kešatmiņā, nekādas starp-ierīces šo atbildi glabāt nevar;

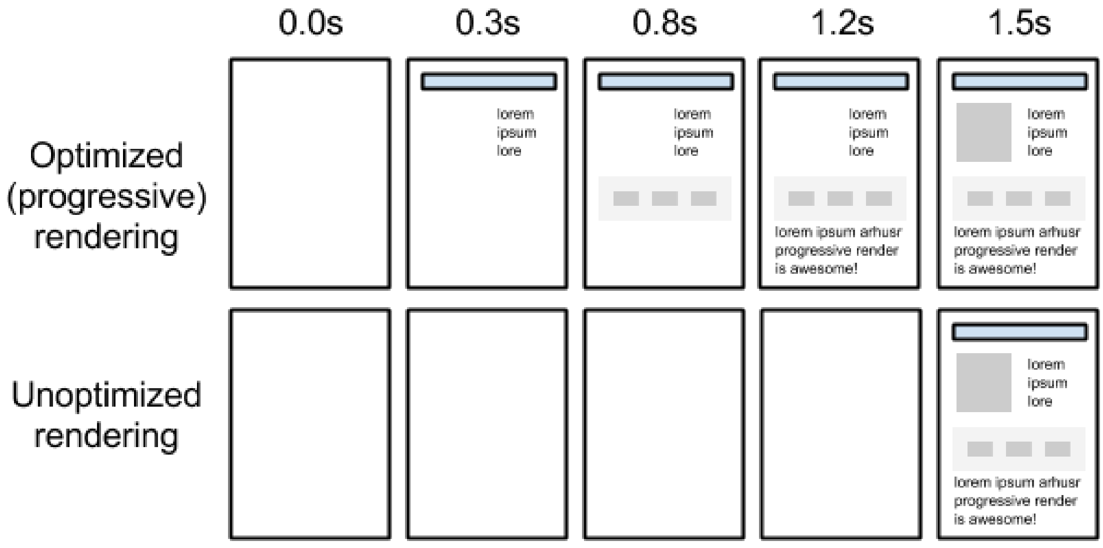
Otra svarīgākā lieta, ir “ETag” īpašība, kuru serveris pievieno atbildes galvenē, bet pārlūkprogramma – pieprasījuma galvenē īpašībai “If-None-Match” pieliek iepriekš saņemto “ETag” vērtību, protams, ja tāda tika saņemta. “ETag” vērtība ir kāds unikāls kods, pēc kura serveris saprot, vai atbilde, uz doto pieprasījumu būs mainījusies. Sākumā pārlūkprogramma pārbauda lokālo kešatmiņu, un ja tajā glabājas pieprasījuma atbilde, kuru pārlūkprogrammai vajadzētu veikt, tā apskata vai pieprasījuma atbildes termiņš nav izbeidzies, gadījumā ja ir, tad serverim tiek nosūtīta iepriekš saņemtā “ETag” vērtība. Serveris pārbauda šo vērtību, un, ja tā nav mainījusies, serveris atgriež atbildi ar 304 kodu, kas nozīmē “Nav mainīts”, un, rezultātā pārlūkprogramma var izmantot kešatmiņā saglabāto atbildi. Labā ziņa izstrādātājiem ir tāda, ka pašiem “ETag” vērtības validācija nav jānodrošina. Ir pietiekoši to iespējot servera pusē, bet par cik dažādiem serveriem tas ir izdarāms dažādos veidos, šī darba ietvaros tas netiks aplūkots [6][10].

Trešā svarīgākā lieta, ir zināt, kā rīkoties gadījumos, kad kādai no atbildēm glabāšanas termiņš tika norādīts salīdzinoši liels, bet tika veiktas izmaiņas, un doto atbildi pārlūkprogrammai ir jāsaņem no jauna. Tipiska pieeja ir nomainīt saiti, pēc kuras tiek pieprasīts kāds no resursiem. Visvieglākais veids to izdarīt, pievienot datnes nosaukumam versiju vai unikālu identifikatoru. Piemēram, ja datnes sākotnējais nosaukums ir “style.css”, pārdēvējam to par “style.001.css”. Kad tajā tiks veiktas svarīgas izmaiņas, pārdēvējam datni par “style.002.css”. Zinot par šādu pieeju, tipiska situācija ir kad galvenā HTML lapa netiek glabāta kešatmiņā, jo tajā tiek mainītas citu resursu versijas. Tādā veidā tiek nodrošināts, ka lietotājam vienmēr būs pieejami atjauninātie resursi.

3. KRITISKAIS RENDERĒŠANAS CEĻŠ

Šajā nodaļā tiks apskatītas tādas lietas, kas attiecas uz to, kā tīmekļa vietnes tiek attēlotas pārlūkprogrammās. Tās ir lietas, par kurām bieži vien tīmekļa lietotņu izstrādātājiem nav ne jausmas – kādā veidā pārlūkprogramma saprot, kas ir uzrakstīts HTML, CSS un JavaScript datnēs, un, kādā veidā pārlūkprogrammas šo informāciju par lietotajiem redzamajiem pikseļiem.

Kritiskais renderēšanās ceļš ietver sevī visu to, kas notiek no HTML, CSS un JavaScript datņu saņemšanas brīža, līdz to pārvēršanai pārlūkprogrammā redzamajiem pikseļiem, ar kuriem lietotājs jau var sākt darboties [12].



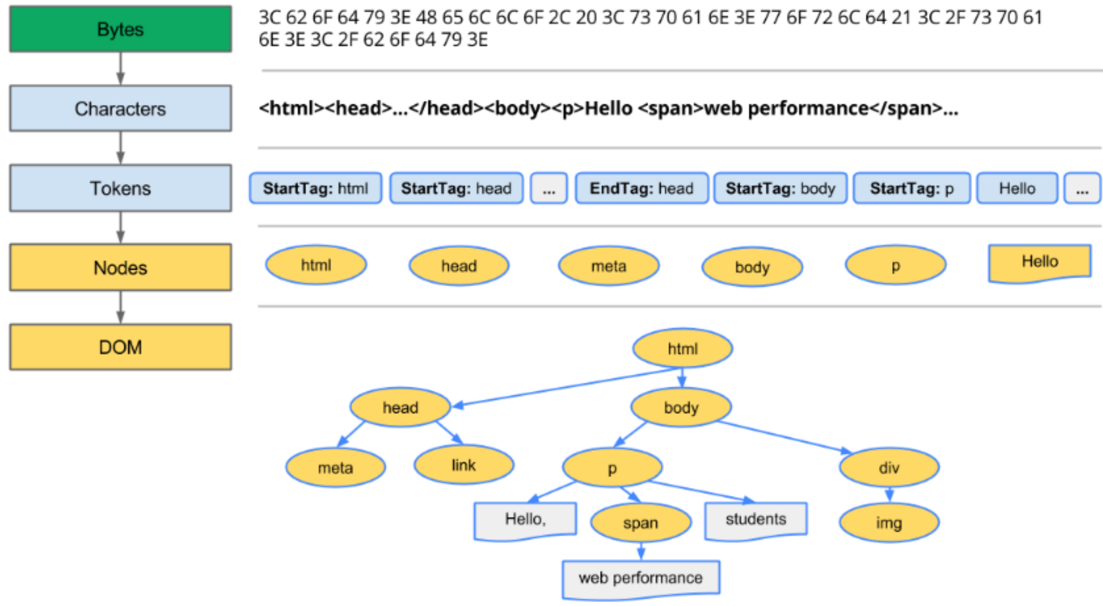
3.1. att. **Optimizētas un neoptimizētas tīmekļa lapas atveidošanas piemērs [12]**

Galvenā doma ir pēc iespējas ātrāk parādīt lietotājam tās lietas, ar kurām viņš uzreiz varētu veikt darbības, pirms pati tīmekļa vietnes lapa ir pilnībā ielādējusies. Piemēram, būtu labi, ja e-veikalā uzreiz tiktu parādītas produktu kategorijas, līdz pati lapa ir ielādējusies, jo ar lielu varbūtību, lietotājs interesēsies par produktu no kādas konkrētas kategorijas.

3.1. Objektu struktūra

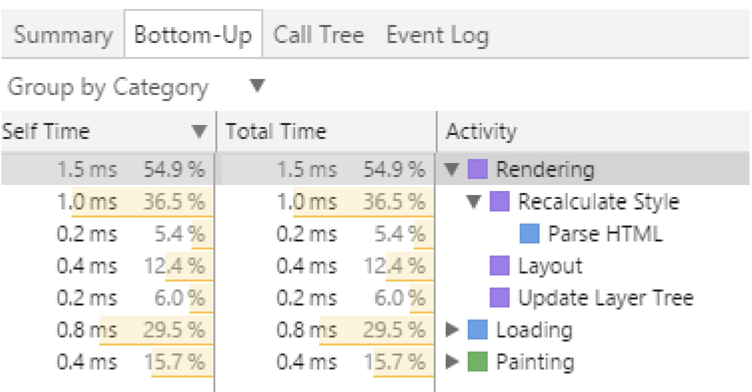
Pirmā, un galvenā lieta pirms tīmekļa vietnes atveidošanas ir pareiza un ātra DOM un CSSOM izveidošana. Banāli, bet tas nozīmē ka HTML un CSS datnēm jānonāk līdz pārlūkprogrammai pēc iespējas ātrāk.

Pārlūkprogrammā DOM izveidošanas process notiek sekojoši: tiek saņemti baiti, tie tiek pārveidoti par simboliem, simbolu virknēs tiek meklēti marķieri, no marķieriem tiek veidoti mezgli, un no mezgliem tiek izveidots kopējais modelis. HTML marķējums tiek pārveidots par DOM, bet CSS marķējums par CSSOM. Katram CSS un HTML marķierim ir savs nolūks, turklāt, katram DOM un CSS mezglam tiek reģistrēta saistība starp pašu mezglu un šī mezgla vecāku un bērniem. Gala rezultāts ir DOM un CSSOM, ar kuriem pārlūkprogramma veic jebkādas turpmākās darbības [15].



3.2. att. **DOM izveides piemērs [15]**

Visas šīs darbības, kas nepieciešamas no bitu pārvēršanas līdz objekta modeļa izveidei, aizņem laiku, it īpaši ja HTML un CSS daudzums ir liels. Šo laiku ir iespējams redzēt izstrādātāja rīkā, kurš ir pieejams praktiski jebkurā pārlūkprogrammā, piemērs ir redzams attēlā 3.3.



3.3. att. **Tīmekļa lapas atveidošanai nepieciešamo laiku piemērs Google Chrome pārlūkprogrammā**

Tādā veidā, var atrast šaurās vietas, kas rodas pie tīmekļa lapas atrādīšanas.

Svarīgi minēt, ka pats par sevi, DOM koks pasaka tikai sakarības starp mezgliem, bet nepasaka, kā konkrēts mezgls izskatīsies, kad tas tiks attēlots. Par to atbild CSSOM [15].

DOM struktūras izveidošanas laikā, pārlūkprogramma parasti sastop pieprasījumus uz citiem resursiem, teiksim, kādu CSS datni. Sastopot šādu pieprasījumu, pārlūkprogramma saprot, ka turpmākā attēlošana nevar notikt bez šī resursa, tāpēc, tas nekavējoties tiek pieprasīts [15].

Protams, CSS noteikumus var rakstīt pa tiešo HTML marķieros, bet tas netiek darīts jau labu laiku, tādā veidā tiek uzlabota koda lasāmība, kā arī, dizaineri var darboties neatkarīgi no izstrādātāju mērķiem.

CSSOM, līdzīgi kā HTML, ir sava koka struktūra. Šāda struktūra ir nepieciešama, jo pārlūkprogramma visiem mezgliem pielieto vispārējus noteikumus, kas definēti augstāk kokā, un, tad rekursīvi izstaigājot koku, katra mezglam pielieto tā specifiskos CSS noteikumus [15].

No izstrādātāju viedokļa, nopietna lieta, ko vajadzētu atcerēties – jebkura ārēja CSS datne nav CSSOM koka augšpusē. Katrai pārlūkprogrammai ir savi iekšējie CSS noteikumi, kas nosaka, kā vienam vai otram mezglam ir jāizskatās. Dažādās pārlūkprogrammās šie noteikumi var būt dažādi. Tāpēc bieži vien tiek pielietots CSS RESET [16]. CSS RESET notīra šos pārlūkprogrammu CSS noteikumus.

Lai apskatītos cik laika nepieciešams CSS apstrādei, pārlūkprogrammu izstrādātāju rīkos ir ieraksts “Recalculate Style”. Tas ir redzams arī 3.3. attēlā. Tajā tiek parādīts nepieciešamais laiks CSS parsēšanai, CSSOM koka izveidei, un stilu rekursīvai aprēķināšanai.

3.2. Atveidošanas koka izveide

DOM un CSSOM tiek apvienoti atveidošanas kokā, kurš tiek izmantots izkārtojuma aprēķiniem, priekš katra redzamā elementa. Balstoties uz atveidošanas koku, tiek palaists zīmēšanas process, kurš atveido pikseļus uz ekrāna. Katra šī procesa optimizācija ir svarīga galējai attēlošanas ātrdarbības uzlabošanai [17].

Veidojot atveidošanas koku, pārlūkprogramma veic sekojošas darbības [17]:

- Sākot no DOM koka saknēs, tiek apskatīti visi redzamie mezgli. Mezgli, kas nav redzami, piemēram skripta marķieri, mezgli, kuriem noslēpti ar CSS, un tml., tiek ignorēti, jo tie neietekmēs atveidošanu;
- Katram redzamajam mezglam tiek meklēti un pielietoti CSSOM kokā esošie noteikumi;
- Redzamajiem mezgliem tiek izveidots to saturs un pielietoti aprēķinātie stili.

Kad atveidošanas koks ir izveidots, vēl nav zināma katra mezgla tiešā pozīcija un izmērs. Tāpēc tiek veikts izkārtošanas process. Tas iziet cauri visam attēlošanas kokam, sākot no saknes, un, katram mezglam aprēķina tā tiešo pozīciju un izmēru. Rezultātā tiek iegūts lodziņa modelis, kurā katra mezgla relatīvie lielumi ir pārveidoti par absolūtiem lielumiem [17].

Izmantojot visu esošo informāciju, tiek palaists zīmēšanas process, kurš pārveido katru mezglu par redzamiem pikseļiem [17].

Laiks, kas nepieciešams attēlošanas koka izveidei, pozīciju un izmēru aprēķināšanai ir aplūkojams pārlūkprogrammu izstrādātāju rīkos. Attēlā 3.3. tas ir punkts “Layout”, savukārt, laiks, kas nepieciešams zīmēšanai, ir redzams attēlā pie punkta “Painting”.

3.3. Kritiskā renderēšanas ceļa optimizācija

Lai pēc iespējas ātrāk tiktu veikta sākotnējā lapas atveidošana, ir jācenšas samazināt trīs mainīgos – kritisko resursu skaits, kritiskā ceļa garums un kritisko baitu skaits [18].

Kritiskie resursi ir tādi resursi, kas bloķē sākotnējo tīmekļa lapas atveidošanu. Šādi resursi ir gan HTML, gan CSS gan JavaScript datnes.

Ir daudzi gadījumi, kad visi CSS resursi vienlaicīgi nav vajadzīgi. Piemēram, ja ir divas CSS datnes, viena no tām paredzēta vispārējam gadījumam, bet otra, lapas drukas vajadzībām. Saprotams, ka otrā datne būs vajadzīga ne būt ne visos gadījumos, tāpēc to vērts ielādēt tikai drukāšanas režīmā. To var paveikt izmantojot "media" īpašību iekš HTML "link" marķiera. Savukārt, CSS vispārējam gadījumam ir jāpiegādā pārlūkprogrammai pēc iespējas ātrāk, tāpēc tās izsaukumu parasti veic HTML dokumenta augšpusē.

JavaScript gadījumā, pēc iespējas ir jāizmanto tā asinhrona izpilde, tādā veidā atveidošana netiks nobloķēta. Ja timekļa lapa ir lietojama bez kāda no JavaScript resursiem, tad laba doma ir izsaukt šos resursus pašās HTML dokumenta beigās.

Kritiskais ceļa garums ir atkarība starp kritiskajiem resursiem un to izmēru. Daži kritiskie resursi var tikt ielādēti tikai tad, kad kāds iepriekšējais resurss tika apstrādāts. Jo lielāks ir šāds resurss, jo ilgāks laiks ir nepieciešams tā apstrādei, un, jo ilgāk laika nepieciešams, lai nākamais kritiskais resurss sāktu ielādēties. Svarīgi ir arī optimizēt kritisko resursu lejupielādes secību [17].

Samazinot nepieciešamo kritisko baitu daudzumu, pārlūkprogramma tos lejupielādēs ātrāk, tādā veidā pirmā tīmekļa lapas attēlošana sāksies ātrāk [17].

Pie vispārīgām rekomendācijām, kritiskā atveidošanas ceļa optimizācijas tiek minēti vēl citi sekojoši punkti [19]:

- Jāizvairās no sinhroniem servera pieprasījumiem;
- Izvairīties no CSS importēšanas no vienas stilu lapas, citā;
- Apsvērt domu, par CSS rakstīšanu iekš HTML marķieriem, kurš nepieciešams primārai tīmekļa lapas atveidošanai.

4. RENDERĒŠANAS ĀTRDARBĪBA

Ne tikai tīmekļa vietņu lejupielādes ātrumam ir būtiska nozīme. Lietotāji sagaida, ka viņu veiktās darbības tiks veiktas ātri, mijiedarbība starp lapu un lietotāju notiks gludi, animācija nebremzēs, lapas ritināšana notiks acumirklī un bez raustīšanās, utt.

Lai izstrādātu tīmekļa vietne, kuras ātrdarbība ir labā līmenī, ir jāsaprot, kā HTML, CSS un JavaScript tiek apstrādāti pārlūkprogrammā.

Lielākā daļa mūsdienu ierīču atjaunina ekrānu 60 reizes sekundē. Tas nozīmē, ka animācijām un dažādām vizuālām pārejām kas notiek pārlūkprogrammā, ir jāiekļaujas katrā no šādām ekrāna atjaunināšanās reizēm, un jāparāda jauns kadrs [20].

Būtībā tas nozīme, ka pārlūkprogrammai ir aptuveni 10 milisekundes viena šāda kadra atrādīšanai. Ja pārlūkprogramma neiekļaujas šajā laikā, saturs, kuram tiek izpildīta animācija, sāk raustīties, rezultātā negatīvi ietekmējot lietotāju pieredzi tīmekļa vietnes izmantošanā [20].

Ir pieci galvenie apgabali, par kuriem izstrādātājam nepārtraukti ir jāpiedomā, un kurus vajag kontrolēt. Šie apgabali ir sekojoši [20]:

- JavaScript – parasti JavaScript izpildes rezultātā tiek veiktas kādas vizuālas izmaiņas. JavaScript var veikt animāciju, jaunu DOM mezglu pievienošanu vai dzēšanu;
- Stila skaitļošana – process, kurā tiek noskaidrots kādi CSS noteikumi tiek piemēroti kādam elementam, kas vajadzīgie noteikumi tiek atrasti, tie tiek pielietoti elementam, un beigu stili tiek izskaitļoti priekš šī elementa;
- Izkārtošana – kad pārlūkprogramma zina kādus CSS noteikumus jāpielieto konkrētam mezglam, tā var sākt skaitļot šī mezgla atrašanās vietu un izmēru. Turklāt viena mezgla izmēri un atrašanās vieta parasti ietekmē citus elementus, tāpēc šis process ir resursu prasīgs;
- Zīmēšana – process kurā tiek zīmēti pikseļi. Zīmēšanas procesā tiek zīmēti teksti, krāsas, attēli, robežas, ēnas, u.c. Zīmēšana parasti tiek veikta vairākos slāņos;
- Kompozīcija – uzzīmēto slāņu apvienošana pareizā secībā. Šī secība ir svarīga, lai lapa tiktu pareizi attēlota, jo ir elementi kuri atrodas zem citiem elementiem utml;

Veicot kādas vizuālas izmaiņas, tiek aizskarts viens vai vairāki augstākminētie apgabali. Tā, piemēram, ja tiek veiktas izmaiņas, kas skar elementu izkārtošanu, pārlūkam ir jāpārskaitļo visu

elementu izkārtošana, un no jauna jāveic zīmēšanas un kompozīcijas process. Bet, ja tiek izmainīta, piemēram, elementa fona krāsa, tad tas skar tikai zīmēšanas procesu.

4.1. JavaScript izpildes optimizācija

Šajā nodaļā tiks aplūkotas dažas lietas, kas ļautu veikt JavaScript izpildes rezultātā radušās nepieciešamās vizuālās izmaiņas.

Vizuālo izmaiņu veikšanai vajadzētu atteikties no tādām JavaScript metodēm kā setTimeout un setInterval, to vietā vajadzētu izmantot requestAnimationFrame metodi. Tā nodrošina, ka JavaScript kāds koda gabals tiks izpildīts tieši kadra veidošanas pašā sākumā. setTimeout un setInterval to nodrošināt nevar, un ir varbūtība, ka kods tiks izpildīts kadra veidošanas beigās, tādā veidā koda izpildes pabeigsies pēc tā, kad kadrs tiks izveidots, rezultātā, šis kadrs tiks palaists garām, kas novedīs, piemēram, pie animācijas noraustīšanās [21].

Otra svarīga lieta ir JavaScript koda izpildes, kurai nepieciešams ilgs laiks, apstrāde. Ideālā gadījumā, pie viena kadra izveides, JavaScript kodam nevajadzētu izpildīties vairāk par 3-4 milisekundēm [21].

Viens no variantiem, ko darīt ar laikietilpīgām JavaScript skaitļošanai, ir JavaScript izpildes pārcelšana no pamat pavediena uz citiem pavedieniem. Tas ir nepieciešams tāpēc, ka JavaScript, kurš izpildās pamatpavedienā, uz izpildes laiku nobloķē stilu skaitļošanu, izkārtošanas uz zīmēšanas procesus. Lai to paveiktu palīgā nāk tīmekļa strādnis, tas ļauj JavaScript kodam izpildīties citā pavedienā. Jāmin, ka variants ar tīmekļa strādni derēs ne būt ne visiem risinājumiem, par cik tam nav pieejas DOM, tāpēc timekļa strādņa izmantošana derēs tikai darbībām saistītām ar datu manipulāciju, piemēram, datu meklēšanu, kārtošanu utml [21].

Otrs veids, kā pārvaldīt laikietilpīgu JavaScript izpildi, ir liela uzdevuma sadale mazos uzdevumos, kur katrs šāds uzdevuma izpildei nepieciešams pavisam nedaudz laika. Piemērs redzams 4.1. attēlā [21].

```
var taskList = breakBigTaskIntoMicroTasks(monsterTaskList);
requestAnimationFrame(processTaskList);

function processTaskList(taskStartTime) {
  var taskFinishTime;

  do {
    // Assume the next task is pushed onto a stack.
    var nextTask = taskList.pop();

    // Process nextTask.
    processTask(nextTask);

    // Go again if there's enough time to do the next task.
    taskFinishTime = window.performance.now();
  } while (taskFinishTime - taskStartTime < 3);

  if (taskList.length > 0)
    requestAnimationFrame(processTaskList);

}
```

4.1.att. **Liela uzdevuma sadalīšana mazākos, ar pārbaudi, ka uzdevums netiks izpildīts vairāk par 3 milisekundēm [21]**

Izmantojot šo pieeju, jārēķinās ar to, ka vajadzētu sniegt informāciju lietotājam par to, ka pamatuzdevums tiek pildīts. To var paveikt, piemēram, ieviešot progresa indikatoru. Jebkurā gadījumā, šī pieeja nenobloķē citu procesus, un lietotājs var turpināt darboties timekļa vietnē.

4.2. Stilu skaitļošanas optimizācija

Jebkuras izmaiņas DOM struktūrā noved pie tā, ka pārlūkprogrammai ir jāpārskaitļo mezglu stili, un, bieži vien jāveic tīmekļa lapas vai tās daļu izkārtošanas atkārtotu skaitļošanu. Protams, ka šīs skaitļošanas aizņem laiku.

Lai paātrinātu stilu skaitļošanu, vajadzētu veidot pēc iespējas vienkāršākus CSS selektorus. Tāpēc, ja ir iespējams, mezgliem vajadzētu pievienot klases vai identifikatorus, tādā veidā atvieglojot pārlūkprogrammas procesu veikšanu, kas nepieciešami mezglu atlasei. Pārlūkprogrammai ir vieglāk atlasīt mezglus, pēc klases, nevis, piemēram, atlasīt mezglus, kuri ir kāda cita mezgla n-tais bērns [22].

Otra lieta, kas ļautu paātrināt stilu skaitļošanu, ir stila izmaiņu veikšana pēc iespējas mazākam mezglu skaitam, iespēju robežās norādīt konkrētu elementu, kuram jāveic izmaiņas.

4.3. Izvairīšanās no lieliem un sarežģītiem izkārtojumiem

Izkārtošanās process ietver sevī elementu ģeometrisko īpašību noteikšanu – izmēru un atrašanās vietu. Šīs īpašības ietekmē pielietotie CSS noteikumi, elementa saturs un elementa vecāks. Izkārtošanas aprēķināšanas procesa izmaksas sastāda elementu daudzums, kurus nepieciešams izkārtot, kā arī izkārtošanas sarežģītība [23].

Jāņem vērā, ka elementa ģeometrisko īpašību izmaiņas, gandrīz vienmēr nozīmē izkārtojuma pārrēķināšanu, un, izkārtošanas tvērumā parasti ir visa tīmekļa lapa. Ja lapā ir daudz elementu, izkārtojuma pārrēķināšana kļūst par laikietilpīgu procesu [23].

Apskatīties, kādu stilu īpašību izmaiņas izsauc izkārtojuma pārrēķināšanu, var apskatīties CSS Triggers tīmekļa vietnē [24].

4.3.1. Flexbox modelis

Svarīga lieta, runājot par izkārtojumu, ir dažādi izkārtojuma izveides modeļi. Uz doto brīdi tiek ieteikts izmantot Flexbox modeli. Par cik izkārtojuma aprēķināšana, izmantojot šo modeli ir daudz ātrāka par citiem, novecojušiem modeļiem. Tika salīdzināta izkārtojuma skaitļošana 1300 peldošiem elementiem, un 1300 elementiem, kuriem tika pielietots Flexbox modelis. Pirmajā gadījumā izkārtojums tika aprēķināts 14 milisekundēs, savukārt otrajā gadījumā 3,5 milisekundēs [23].

Jāmin, ka Flexbox izmantošana nav atbalstīta vecākās pārlūkprogrammās, tāpēc pirms tā lietošanas, ir jāpārliecinās, ka tas darbosies pārlūkprogrammas, kuras paredzēts atbalstīt izstrādājamai tīmekļa vietnei.

4.3.2. Piespiedu sinhronizācija ar izkārtojumu

Normālā gadījumā, kadra izveidošanā ir sekojoša secība – tiek izpildīts JavaScript, tiek veikta stilu skaitļošana, un tad tiek rēķināts izkārtojums. Tomēr ir iespējams ar JavaScript palīdzību, piespiedu sinhronizēšanos ar izkārtojumu. No šādiem gadījumiem vajadzētu izvairīties.

Piespiedu sinhronizācija ar izkārtojumu rodas gadījumā, kad mēs kādam elementam izmainām kādu tā ģeometrisko vērtību, un uzreiz mēģinām šo vērtību nolasīt. Par cik, JavaScript

pieejamie dati par elementu ir no pagājušā kadra, tad, pārlūkprogrammai šādā gadījumā jāveic izkārtojuma skaitļošanu piespiedu kārtā. Tāpēc, elementa vērtība sākumā ir jānolasa, un tikai tad jāmaina [23].

4.4. Zīmēšanas procesa optimizācija

Izmainot jebkuru mezgla stila īpašību, tiek izsaukts zīmēšanas process. Pārlūkprogrammām ir iespēja veikt zīmēšanas procesu dažādiem slāņiem, tāpēc svarīgi ir izdalīt elementus, kuri nepārtraukti pārzīmēti, jaunos slāņos. Tas ļauj pārzīmēt šos elementus neietekmējot citus, tīmekļa lapā esošos elementus. Izvietot elementu jaunā slānī ir iespējams ar "will-change" CSS īpašību. Pārlūkprogrammām, kas neatbalsta "will-change" īpašību, ir iespējams īpašībai "transform" piešķirt vērtību "translateZ". Bet ar jaunu slāņu izveidi nevar pārāk aizrauties, jo katram slānim nepieciešama papildus vieta atmiņā. Kā arī, obligāti vajadzētu veikt ātrdarbības mērījumus, pēc jauna slāņa izveidošanas, ar mērķi pārliecināties, ka ātrdarbība tiešām no tā uzlabojas [25].

Vajadzētu censties pārzīmēt pēc iespējas mazākus apgabalus, tā, piemēram ja kādam no elementiem virspusē ir citi elementi, un šim elementam tiks nomainīta fona krāsa, tas nebūs redzams vietās, kur atrodas augšpusē esošie elementi – tiks pārzīmēta nevajadzīgs apgabals [25].

Svarīgi minēt, ka dažādu CSS īpašību zīmēšana aizņem dažādus laika resursus piemēram, uzzīmēt viendabīgu krāsu ir ātrāks process, nekā uzzīmēt ēnu.

4.5. Izmaiņas kompozīcijā

Runājot par kompozīciju, ir divi galvenie faktori, kas ietekmē ātrdarbību – slāņu skaits un īpašības, kas tiek izmantotas animācijai [26].

Lai izvairītos no izkārtojuma aprēķināšanas un zīmēšanas procesa, ir iespējams izmantot "transform" un "opacity" CSS īpašības. Tās ietekmē tikai kompozīcijas procesu. Tāpēc, tiek stingri rekomendēts iespēju robežās izmainīt tieši šīs īpašības [26].

5. RESURSU PĀRVALDE

Daudzās jaunāko versiju pārlūkprogrammās ir iespēja pārvaldīt resursus, norādot, kādus no resursiem vajag lejupielādēt ar augstāko prioritāti, vai tieši otrādāk – veikt to lejupielādi pēc visu pārējo resursu lejupielādes, parasti tas tiek veikts balstoties uz pieņēmumiem par tālākajām lietotāja darbībām. Zemāk tiks apskatīts, kādā veidā tas ir izdarāms.

Uz doto brīdi eksistē pieci dažādi resursu pārvaldes veidi – DNS priekšienese, priekšsavienošana, priekšnolase, priekšattēlošana un priekšlejupielāde.

5.1. DNS priekšienese

Lai pārlūks varētu saņemt resursus no kādas adreses, vai pārietu uz to, sākumā pārlūkam ir jāatrod IP adrese domēna nosaukumu sistēma, kas atbilst adreses nosaukumam. Tas parasti aizņem kādu laiku. Lai ietaupīt šo laiku, pārlūkam var tieši norādīt atrast IP adresi noteiktajai adresei.

```
<link rel="dns-prefetch" href="//www.example.com">
```

5.1. att. **DNS priekšieneses izmantošanas piemērs**

5.1 attēlā redzams, kādā veidā tiek veikta DNS priekšienese. Kā redzams iekš taga “link” atribūta “rel” vērtība “dns-prefetch” norāda uz to, ka pārlūkam jāveic DNS priekšienese adresei, kas norādīta “href” atribūtam.

Jāmin, ka pārlūkprogrammas pēc noklusējuma veic DNS priekšienesi vairākām adresēm. Piemēram Chrome pārlūkprogrammā šīs adreses var apskatīties adrešu joslā ievadot “chrome://DNS”, savukārt, lai apskatītos laikus, kas tiek tērēti adrešu atrisināšanai DNS, adrešu joslā jāievada “chrome://histograms/DNS.ResolveSuccess” [28].

Host name	How long ago (HH:MM:SS)	Motivation
http://keycdn.com	47:45:55	n/a
https://docs.google.com/	47:45:57	n/a
https://fonts.googleapis.com/	47:45:56	n/a
https://fonts.gstatic.com/	47:45:56	n/a
https://lh3.googleusercontent.com/	47:45:56	n/a
https://lh4.googleusercontent.com/	47:45:56	n/a
https://lh5.googleusercontent.com/	47:45:56	n/a
https://lh6.googleusercontent.com/	47:45:56	n/a
https://keycdn.com	47:45:56	n/a
https://ssl.gstatic.com/	47:45:56	n/a

5.2. att. **Piemērs kurā redzamas adreses kurām pārlūkprogramma veiks DNS priekšienesi**

```
Histogram: Net.DNS.ResolveSuccessTime recorded 5391 samples, mean = 16.1 (flags = 0x41)
0     ----------------------------------------------------------------------O (921 = 17.1%)
1     ----------------------------------------------O                        (615 = 11.4%) {17.1%}
2     -O                                                                      (19 = 0.4%) {28.5%}
3     --O                                                                     (30 = 0.6%) {28.8%}
4     -------------------------------------O                                  (499 = 9.3%) {29.4%}
5     ------------------------------------------------------O                 (712 = 13.2%) {38.7%}
6     ---------------------------O                                            (364 = 6.8%) {51.9%}
7     ----------------O                                                       (217 = 4.0%) {58.6%}
58    -O                                                                      (50 = 0.9%) {94.3%}
67    O                                                                       (28 = 0.5%) {95.2%}
77    O                                                                       (31 = 0.6%) {95.7%}
89    O                                                                       (31 = 0.6%) {96.3%}
210   O                                                                       (13 = 0.2%) {99.3%}
242   O                                                                       (10 = 0.2%) {99.6%}
279   O                                                                       (5 = 0.1%) {99.8%}
571   O                                                                       (1 = 0.0%) {99.9%}
659   ...
1013  O                                                                       (4 = 0.1%) {99.9%}
1169  ...
```

5.3. att. **Piemērs kurā redzami laiki, kas tiek tērēti adrešu atrisināšanai DNS**

5.3. attēla kreisajā pusē ir redzams laiks milisekundēs, kas nepieciešams adreses atrisināšanai. Lielākoties šie laiki nav lieli, taču dažos gadījumos laiks adreses atrisināšanai pārsniedz pat 1 sekundi. Tādā veidā DNS priekšielase salīdzinoši viegli ļauj ietaupīt šo laiku.

5.2. Priekšsavienošana

Priekšsavienošanas mērķis ir veikt savienojumu ar kādu sistēmu – atrisināt adresi no DNS, veikt TCP izaicinājumrokasspiedienu, un, nepieciešamības gadījumā, veikt vienošanos TLS līmenī. Tādā veidā, pārlūkprogramma jau iepriekš būs gatava komunicēt ar sistēmu, ar kuru tika veikts savienojums, kā rezultātā tiks ietaupīts laiks vēlākai šī savienojuma izveidei.

```
<link rel="preconnect" href="http://www.example.com" crossorigin>
```

5.4. att. **Priekšsavienošanas izmantošanas piemērs**

Kā redzams 5.4. attēlā, priekšsavienošana ir uztaisāma līdzīgi kā DNS priekšienese, atšķiras tikai atribūta “rel” vērtība – “preconnect”. Atribūts “crossorigin” nav obligāti norādāms. Tas tiek norādīts, ja savienojums tiek veikts ar kādu ārējo sistēmu un ir nepieciešams veikt vienošanos TLS līmenī.

5.3. Priekšnolase

Priekšnolase tiek izmantota gadījumos, kad tiek pieņemts, ka, ka kāds noteikts resurss visdrīzāk tiks izmantots vēlāk. Priekšnolase var tikt pielietota jebkuriem resursiem, kas var tikt saglabāti kešatmiņā, piemēram attēli, JavaScript un CSS datnes. Šis noteikums ir būtisks, jo īstenībā tas, ko dara priekšnolasīšana, ir resursa lejupielāde un tā saglabāšana kešatmiņā, un, kad lejupielādētais resurss būs nepieciešams, pārlūkprogramma to pa tiešo nolasīs no kešatmiņas.

```
<link rel="prefetch" href="image.png">
```

5.5. att. **Priekšnolases izmantošanas piemērs**

Kā redzams attēlā 5.5., norādīt pārlūkprogrammai par vajadzību priekšnolasīt kādu resursu, var paveikt līdzīgi kā iepriekš minētajām resursu pārvaldīšanas komandām. Atribūta “rel” vērtībai priekšnolases gadījumā jābūt “prefetch”.

Vērts minēts, ka priekšnolase var arī netikt veikta. Tā, pārlūkprogramma var atteikties veikt priekšnolasi, ja tīmekļa ātrums ir lēns, bet resurss ir liels, vai arī gadījumos ja pārlūkprogramma ir

“aizņemta” ar citiem darbiem. Šāda pārlūkprogrammu uzvedība var mainīties to dažādās versijās, un tīmekļa vietņu izstrādātāji tieši never šo uzvedību ietekmēt [29].

5.4. Priekšattēlošana

Pirms aprakstīt priekšattēlošanas paņēmienu, gribētos uzreiz minēt, ka uz darba rakstīšanas brīdi, šis paņēmiens ir novecojis un tiek atbalstīts pavisam nedaudzās pārlūkprogrammās. Vienīgais iemesls, kāpēc priekšattēlošana tiek minēta šī darba ietvaros - iepriekš tā tika diezgan plaši apspriesta un izmantota, ir palikuši arī daudzi apraksti par šī paņēmiena realizēšanu, un, izstrādātājs, kurš tikko sācis iepazīties ar resursu pārvaldes iespējām, var izšķērdēt laiku, mēģinot pielietot šo paņēmienu.

Priekšattēlošanas paņēmiens ļauj lejupielādēt norādīto adresi, tādā veidā, it kā tā tiktu atvērta atsevišķā pārlūkprogrammas logā vai cilnē. Tiek tielādēti visi nepieciešamie resursi lapas darbībai, izveidots lapas DOM un CSSOM, tiek veikta izkārtošana un izpildīts JavaScript. Vienīgā atšķirība ir tāda, ka tas notiek fonā, lietotājam to nemaz nezinot [28].

Tādā veidā priekšattēlošana ļauj pārlūkprogrammai, gadījumā ja lietotājs navigē uz priekšattēloto lapu, parādīt to ar minimālām aizturēm.

Galvenais šī paņēmiena mīnuss ir tāds, ka tas ir resursu ietilpīgs, un bieži priekšattēlošana tika izmantota nepamatoti, tādā veidā ieguvums no šī paņēmiena biežāk bija mazāks par radītiem zaudējumiem. Tas ir viens no iemesliem, kāpēc pārlūkprogrammas atteicās no tā atbalsta, kā arī tiek minēts, ka šī paņēmiena uzturēšana bijusi sarežģīta [30].

Jebkurā gadījumā šis paņēmiens deva iespēju uzlabot tīmekļa vietņu ātrdarbību, un kas zina, iespējams kādreiz pārlūkprogrammas sāks to atkal atbalstīt, jo šī paņēmiena ideja tik tiešam ir interesanta.

5.5. Priekšlejupielāde

Savā ziņā, priekšlejupielāde ir līdzīga priekšnolasei. Būtiskākā atšķirība ir tāda, ka priekšnolase tiek izmantota, galvenokārt, gadījumos, kad tiek pieņemts, ka iespējams būs vajadzīgi noteikti resursi, pēc lietotāja veiktajām darbībām, parasti tā ir navigēšana uz citu lapu. Līdz ar to priekšnolase tiek izpildīta ar zemu prioritāti. Savukārt priekšlejupielāde galvenokārt

ir paredzēta resursu lejupielādei, kas būs nepieciešami tekošajā tīmekļa lapā. Tāpēc priekšlejupielāde tiks veikta pēc iespējas ātrāk.

Priekšejupielādei ir iespējams norādīt lejupielādējamā resursa tipu, tādā veidā pārlūkprogramma vēl papildus var noteikt prioritātes starp resursiem, kurus paredzēts priekšlejupielādēt. Tas ir izdarāms ar atribūta “as” palīdzību.

Bez tam, priekšlejupielādei nav ietekmes uz pārlūkprogrammas galvenā “window” objekta “onload” notikumu. Tas nozīmē, ka pašas lapas pašu nepieciešamāko resursu lejupielāde notiks asinhroni priekšlejupielādei, un lapa varēs tikt atveidota, pat tad ja priekšlejupielāde netiks pabeigta visiem resursiem.

Apskatīsimies variantus, kad priekšejupielāde varētu būt noderīga.

5.5.1. Apslēpto resursu priekšlejupielāde

Par apslēptiem resursiem šeit tiek saukti tādi resursi, par kuru nepieciešamību pārlūkprogramma uzreiz nevar pateikt. Kā piemērs varētu būt fona attēla izmantošana iekš CSS koda – pārlūkprogramma uzzinās par šāda attēlā nepieciešamību tikai pēc tam, kad tiks izveidots CSSOM. Tāpat ir arī ar izmantotajiem fontiem, un, par cik tie ietekmē teksta renderēšanos, tad fontu priekšejupielāde ir vēl jo vairāk ieteicama.

```
<link rel="preload" href="font.woff2" as="font" type="font/woff2" crossorigin>
```

5.6. att. **Priekšlejupielādes izmantošanas piemērs**

Kā redzams attēlā 5.6. priekšlejupielādes atribūta “ref” vērtība ir “preload”. Ar atribūta “as”, palīdzību pārlūkprogrammai tik pateikts, ka ielādējamais resurss būs fonts. Atribūts “type” norāda konkrētu fonta tipu. Tas tiek darīts, lai, ja gadījumā pārlūkprogramma neatbalsta šo konkrētu tipu, tas netiktu priekšlejupielādēts. Gan atribūts “as”, gan “type” nav obligāti jānorāda.

5.5.2. Skripta resursu priekšlejupielāde bez to izpildīšanas

Priekšielāde paver jaunas iespējas pārvaldīt skriptu izpildi. Rodas iespēja tikai lejupielādēt skriptu resursu, bet veikt to izpildi tikai pie kādiem noteiktiem nosacījumiem.

Tādā veidā rodas iespēja kontrolēt to, ka pārlūkprogramma sākumā noparsēs un izpildīs kritiski nepieciešamos skriptus, tādus kas, piemēram, nepieciešami sākotnējai ekrāna renderēšanai, un tikai tad sāks apstrādāt pārējos skriptus.

```
<link rel="preload" href="script.js" as="script">

//Later under some conditions
var script = document.createElement("script");
script.src = "script.js";
document.body.appendChild(script);
```

5.7. att. **Skripta resursa priekšlejupielāde un šī skripta pievienošana izpildei**

Attēla 5.8. piemērā redzams, ka pārlūkprogrammai tika norādīts priekšlejupielādēt resursu “script.js”. Un tad ar JavaScript palīdzību tiek izveidots “script” tags, par kura avotu kalpotu priekšielādētais skripts. Pēctam notiek šī taga pievienošana pārlūka dokumentam. Pēc pievienošanas, pārlūkprogramma noparsēs doto skriptu un sāks to izpildīt.

5.5.3. Resursu priekšlejupielāde balstoties uz CSS @media noteikumiem

Priekšejupielādei ir iespējams norādīt “media” atribūtu (CSS @media noteikumu analogs - var norādīt jebkurus CSS @media noteikumus), tādā veidā kontrolējot, ka kāds no resursiem tiks ielādēts tikai pie noteiktiem apstākļiem.

Tadā veidā CSS @media noteikumi ļauj ne tikai pielietot noteiktus stilus, ja tika izpildīti noteikti noteikumi, bet arī regulēt resursu lejupielādi. Ar šiem noteikumi var aprakstīt uzvedību atkarībā no sekojošiem punktiem:

- Pārlūkprogrammas augstums un/vai platums;
- Iekārtas ekrāna augstums un/vai platums;
- Iekārtas orientācija – vertikāla vai horizontāla;
- Iekārtas izšķirtspēja u.c. [31].

```
<link rel="preload" as="script" href="script_mobile.js" media="(max-width: 600px)">

<link rel="preload" as="script" href="script_desktop.js" media="(min-width: 601px)">
```

5.8. att. **Atribūta "media" izmantošanas piemērs priekšlejupielādē**

Tā, 5.8. attēla redzamajā piemērā, gadījumā, ja lietotāja izmantotās iekārtas ekrāna platums nepārsniegs 600 pikseļus, tiks ielādēts resurss "script_mobile.js", pretējā gadījumā – resurss "script_desktop.js".

Kopumā, visas augstāk minētās resursu pārvaldes metodes ļauj būtiski uzlabot tīmekļa vietnes ātrdarbību. Lielākā sarežģītība to pielietošanā, ir noteikt, kad un kuriem resursiem tās pielietot, par cik pastāv liela varbūtība, ka tīmekļa lietotne veiks lieku resursu lejupielādi, vai arī tiks veiktas nevajadzīgas savienošanās, un tas negatīvi atspoguļosies uz lietotāju patērēto trafiku.

6. ĀTRDARBĪBAS OPTIMIZĀCIJA BALSTOTIES UZ LAIKA UZTVERI

Šajā nodaļā tiks apskatīti aspekti, kas palīdzētu saprast cilvēku laika uztveri no psiholoģijas viedokļa. Tas ļaus labāk saprast kāpēc, piemēram, ir nepieciešams samazināt kritisko renderēšanas ceļu, kādas vēl ir iespējas ar vizuālām manipulācijām šķietami samazināt laiku, kas nepieciešams, lai lietotājs varētu izmantot tīmekļa vietni. Izprotot cilvēka laika uztveri, kļūs arī skaidrāks, kāpēc tiek pielietotas vienas vai otras tīmekļa lietotņu ātrdarbības optimizācijas metodes.

Laiku ir iespējams analizēt no diviem skatu punktiem – objektīvā un subjektīvā. Par objektīvo laiku tiek saukts tāds laiks, kuru iespējams konkrēti izmērīt ar pulksteņa palīdzību. Savukārt, runājot par subjektīvo laiku, tiek domāts laiks, kurš tiek mērīts pēc lietotāju "iekšējā pulksteņa". Subjektīvajam laikam parasti nevar piemērot tādas mērvienības kā milisekundes, sekundes, minūtes u.tml., šeit parasti runa iet par tādām laika definīcijām kā "tūlītējs", "ātrs", "lēns" u.c. Šajā sadaļā lielākā daļa uzmanības tiks veltīta subjektīvajam laikam, bet objektīvais laiks arī tiks aplūkots sīkāk.

6.1. Objektīvais laiks

Runājot par objektīvo laiku un cilvēka laika uztveri, būtu interesanti noteikt atšķirības laikā, kuras cilvēks ir spējīgs pamanīt, kā arī izdalīt laika intervālus, kurus cilvēks uztver dažādi.

6.1.1. Atšķirības slieksnis

Psiholoģija eksistē jēdziens "vienkārši pamanāmā starpība" jeb "atšķirības slieksnis" – tas nosaka stimula minimālo starpību, lai cilvēks to varētu ievērot lielākajā daļā gadījumu [32]. Šis likums vēlāk tika pielietots cilvēku uztveres mērīšanā, kā rezultātā, psihofizikā parādījās Vēbera-Fehnera likums, kas tiek saistīts ar cilvēka uztveri – ar to tiek apskatīta saistība starp faktiskajām, kāda fiziska stimula izmaiņām, un cilvēka uztvertajām izmaiņām [33].

Vēbera-Fehnera likums ir attiecināms arī uz laika uztveri. Tā, balstoties uz eksperimentiem psihofizikā, laiku nepieciešams izmainīt par 7-18%, lai cilvēks būtu spējīgs pamainīt izmaiņas. Šie lielumi ir attiecināmi runājot par laika intervāliem līdz 30 sekundēm. Nav slikta doma noapaļot šos skaitļus uz augšu un turpmāk pielietot 20%, tādā veidā atvieglojot aprēķinus [34].

Tātad, mērot uzlabojumus ātrdarbībā objektīvajā laikā, vajadzētu pieturēties šiem 20%, lai lietotāji būtu spējīgi pamanīt izmaiņas. Tā, piemēram, ja atbilde uz kādu lietotāja darbību tiek sniegta piecās sekundēs, un 20% no piecām sekundēm ir viena sekunde, tad, lai lietotājs pamanītu uzlabojumus ātrdarbībā, atbilde jāsniedz vismaz četrās sekundēs. Jāmin, ka, ja gadījumā kādu iemeslu dēļ atbildes laiks palielinās, tam nevajadzētu būt lielākam par sešām sekundēm – šajā gadījumā lietotājs visdrīzāk nepamanīs ātrdarbības pazemināšanos.

6.1.2. Laika uztveres gradācija no psiholoģijas viedokļa

Runājot par nelieliem laika ilgumiem, ir svarīgi apzināties četrus laika intervālus, kurus cilvēks uztver dažādi. Visi zemāk minētie laika intervāli ir domāti kontekstā kā atbildes laiks uz kādu lietotāja darbību.

Laika uztveres intervāli [34]:

- 100 – 200 milisekundes: šāds laika diapazons, kas nepieciešams atbildei uz lietotāja darbību, tiek saukts par momentānu. Atbilde, kas tiek dota šajā laika intervālā, no lietotāju puses tiek uztverta kā atbilde bez aiztures. Reālajā dzīvē aptuveni šāds laiks nepieciešams cilvēkam lai noreaģētu uz kādu ārējo stimulu;
- 0,5 – 1 sekunde: šis laika intervāls no cilvēka uztveres viedokļa traktējams kā tūlītējs. Ja atbilde uz kādu lietotāja darbību tiek sniegta šajā laika intervālā, aiztures tiek uztvertas, bet tās ir pieņemamas un, parasti, neizraisa gaidīšanas sajūtu. Aptuveni šādi laika intervāli parasti tiek ieturēti cilvēkam sarunājoties ar citu cilvēku. Balstoties uz vienu no pētījumiem, viena sekunde ir tas laiks, kad cilvēka domu plūsmu paliek nepārtraukta [35]. No tīmekļu lietotņu darbības viedokļa, tas būtu tas laiks, kurā vajadzētu likt lietotājam saprast, ka viņa izpildītā darbība ir saprasta;
- 2-5 sekundes: Laika posms, kurā lietotājs spēj pilnībā koncentrēties un iedziļināties viņa veiktajās aktivitātes. Šāds stāvoklis psiholoģijā tiek saukts pat "plūsmu" vai "optimālo pieredzi" [36]. Ja atbilde uz kādu no lietotāja darbībām tiek dota šajā laika intervālā, lietotājs turpina koncentrēties uz izpildāmajām aktivitātēm.
- 5-10 sekundes: Ja atbildes uz lietotāja darbību tiek sniegta šajā laika intervālā, lietotājs joprojām ir spējīgs fokusēties uz izpildāmo uzdevumu, bet visdrīzāk viņa uzmanība tiks novērsta. Laiks līdz desmit sekundēm ir tas limits, ko sauc par cilvēka uzmanības laika intervālu.

6.2. Subjektīvais

Parasti laika uztvere cilvēka smadzenēs ir atšķirīga no laika, kas tiek mērīts ar pulksteņa palīdzību. Subjektīvā laika uztvere ir atkarīga no daudziem faktoriem, piemēram, cilvēka vecums, emocionālais stāvoklis, apkārtējās vides temperatūra, diennakts puse, psihoaktīvās vielas, kā arī daudz kas cits [37].

No psiholoģiskā viedokļa, laiks sastāv no notikumiem, katram šādam notikumam ir sākums un beigas, tas šie notikuma sākuma un beigu stāvokļi arī veido to, kas tiek saukts par laiku [38].

Katram šādam notikumam ir divas fāzes – aktīvā un pasīvā. Aktīvā fāze ir tāda fāze, kuras laikā cilvēks veic kādas apzinātas darbības. Tās var būt gan fiziska veida aktivitātes, gan domāšanas procesi, piemēram, kāda intelektuāla uzdevuma risināšana. Savukārt pasīvā fāze ir tāda fāze, kuras laikā no cilvēka darbībām kopējais notikuma gaita nevar tikt ietekmēta. Kā piemērs varētu būt lifta gaidīšana, stāvēšana rindā vai autobusa gaidīšana pieturā [38].

No laika uztveres viedokļa, pasīvās fāzes laikā, laiks tiek uztverts kā ilgāks laika periods, nekā aktīvās fāzes laikā, pat ja objektīvā laika intervāli abos gadījumos ir vienādi. Balstoties uz pētījumiem, cilvēki pasīvās fāzes laikā, gaidīšanas laiku pārvērtē aptuveni par 36% no objektīvā laika. Savukārt aktīvās fāzes laikā pagājušais laiks lielākā vairumā gadījumu netiek pamanīts jeb netiek uztverts kā gaidīšana. Tas saistīts ar to ka aktīvās fāzes laikā persona nodarbojas ar kādu apzinātu darbību [39][40].

Viegli var secināt, ka, lai varētu pārvaldīt ar psiholoģisko laiku, un, no tīmekļa lietotņu lietotāju laika uztveres tiem šķistu, ka ir patērēts mazāk laika, lai sniegtu atbildi kādām to veiktajām darbībām, pēc iespējas ir jāsamazina pasīvā fāze, palielinot aktīvo fāzi.

Lai to sasniegt, tiek piedāvāts pieturēties divām tehnikām – apsteidzošs sākums un priekšlaicīga pabeigšana. Pielietojot apsteidzošā sākuma tehniku, notikuma aktīvā fāze tiek izveidota tā sākumā, savukārt pāreja uz pasīvo fāzi tiek veikta pēc iespējas vēlāk. Savukārt priekšlaicīgas pabeigšanas tehnika nozīmē to, ka notikums sākas ar pasīvo fāzi, bet pāreja uz aktīvo fāzi notiek pēc iespējas ātrāk [34].

6.2.1. Apsteidzošs sākums

Runājot par apsteidzošo sākumu, tīmekļa lietotnēs šo tehniku palīdzēs realizēt 5. nodaļā aprakstītās resursu pārvaldes metodes. Kā piemēru varētu minēt interneta veikala tīmekļa lietotni. Tā, ja lietotājs ir pievienojis iepirkumu grozam kādas no precēm, un, tad viņš pāriet uz iepirkumu

grozu, mēs varam norādīt pārlūkprogrammai priekšnolasīt resursus, kas būs nepieciešami tālākajos iepirkšanās soļos, piemēram, resursus, kas būs nepieciešami rēķina apmaksas solī, personīgās informācijas ievades solī un tml.

6.2.2. Priekšlaicīga pabeigšana

Šeit lietā iet tīmekļa lapas atveidošanas optimizācijas, kas aprakstītas šī darba trešajā nodaļā. Pārlūkprogramma sāk rādīt lietotājam lapas saturu pa daļām, tā lietotājs pāriet uz aktīvo fāzi, pētot, kas lapā jau ir parādījies, bet pārējā lapas satura lejupielāde un atveidošana notiek, kad lietotājs jau ir aktīvajā notikuma fāzē. Tādā veidā, lietotajam rodas iespaids, ka laiks, kas patērēts uz lapas atveidošanu ir mazāks nekā tas ir īstenībā. Tieši tāpēc kritiskā renderēšanas ceļa samazināšana ir svarīgā – tā ļauj lietotājam ātrāk pāriet no pasīvās uz aktīvo fāzi.

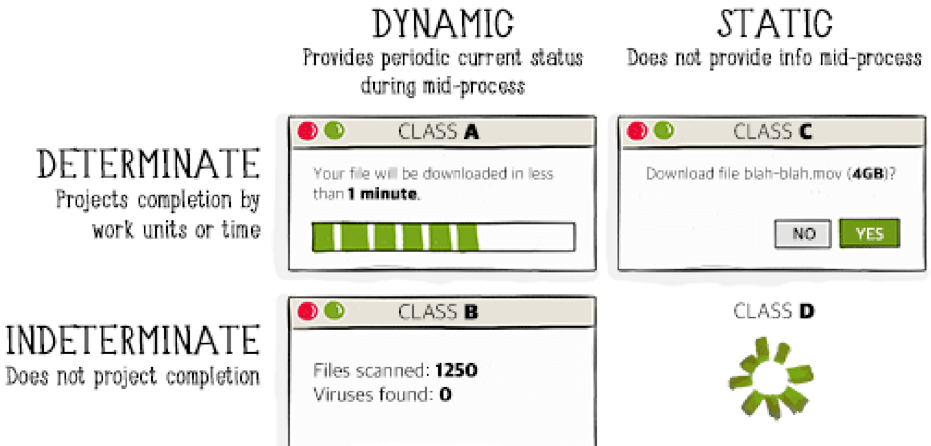
Šī tehnika arī plaši tiek izmantota video un audio straumēšanā. Sākot atskaņot kādu video vai audio datni, lietotājs to redz un/vai dzird tiklīdz minimālais datu apjoms straumēšanai ir lejupielādēts, bet pārējā video vai audio daļa tiek lejupielādēta fonā, kad lietotājs jau skatās video vai klausās audio, jeb ir pārslēdzies uz aktīvo fāzi.

6.2.3. Citi paņēmieni

Atklāts jautājums paliek, kā rīkoties gadījumos, kad notikumam aktīvā fāze ir minimāla, vai tās vispār nav.

Gadījumā, kad tīmekļa vietņu lietotājiem nākas gaidīt, lielāko neapmierinātību lietotājos izraisa nenoteiktība par to, cik ilgi nāksies gaidīt, kāpēc ir jāgaida, un, vai gaidīšanas rezultātā vispār tiks sagaidīts rezultāts [41].

Acīmredzami, vajadzētu mēģināt parādīt lietotājiem gaidīšanas procesa gaitu. Viens no tādiem veidiem ir progresa indikatori. Progresa indikatorus var iedalīt divās grupās – dinamiskie un statiskie. Savukārt katrai no šīm grupām ir 2 apakšgrupas – determinētie un nenoteiktie. Dinamiskajos indikatoros, atšķirībā no statiskajiem, informācija ar laiku mainās. Savukārt determinētie no nenoteiktajiem atšķiras ar to, ka pirmie parāda laiku, vai kādas citas vienības, pēc kurām lietotājs var noteikt, kad process tiks pabeigts [34].



6.1. att. ***Progresa indikatoru piemēri*** **[42]**

Bet, kad un kuru no procesa indikatoriem rādīt lietotājam? Atcerēsimies 6.1.2. nodaļā aprakstīto laika uztveres gradāciju, katram no laika intervāliem būtu ieteicams pielietot sekojošus procesa indikatorus [42]:

- Tā, gadījumā, kad lietotājam atbilde tiek sniegta momentāni, procesa indikatoru rādīt nevajadzētu.
- Ja atbilde tiek sniegta tūlītēji, varētu parādīt vienkāršu progresa indikatoru, kas 6.1. attēlā klasificēts kā D klases indikators.
- Ja atbildei nepieciešamais laika intervāls ietilpst optimālās pieredzes laika intervālā, varam parādīt D klases vai vienkāršotu A klases indikatoru.
- Un, savukārt, ja atbildes laiks ir lielāks par 5 sekundēm, lietotājam vajadzētu sniegt detalizētāku informāciju. Šeit varētu noderēt A klases indikators, kurā tiek norādīta informācija par atlikušo gaidīšanas laiku vai arī tekošais stāvoklis, piemēram procentuāli, vai, norādot, ka ir lejupielādēts daudzums X no kopējā daudzuma Y.

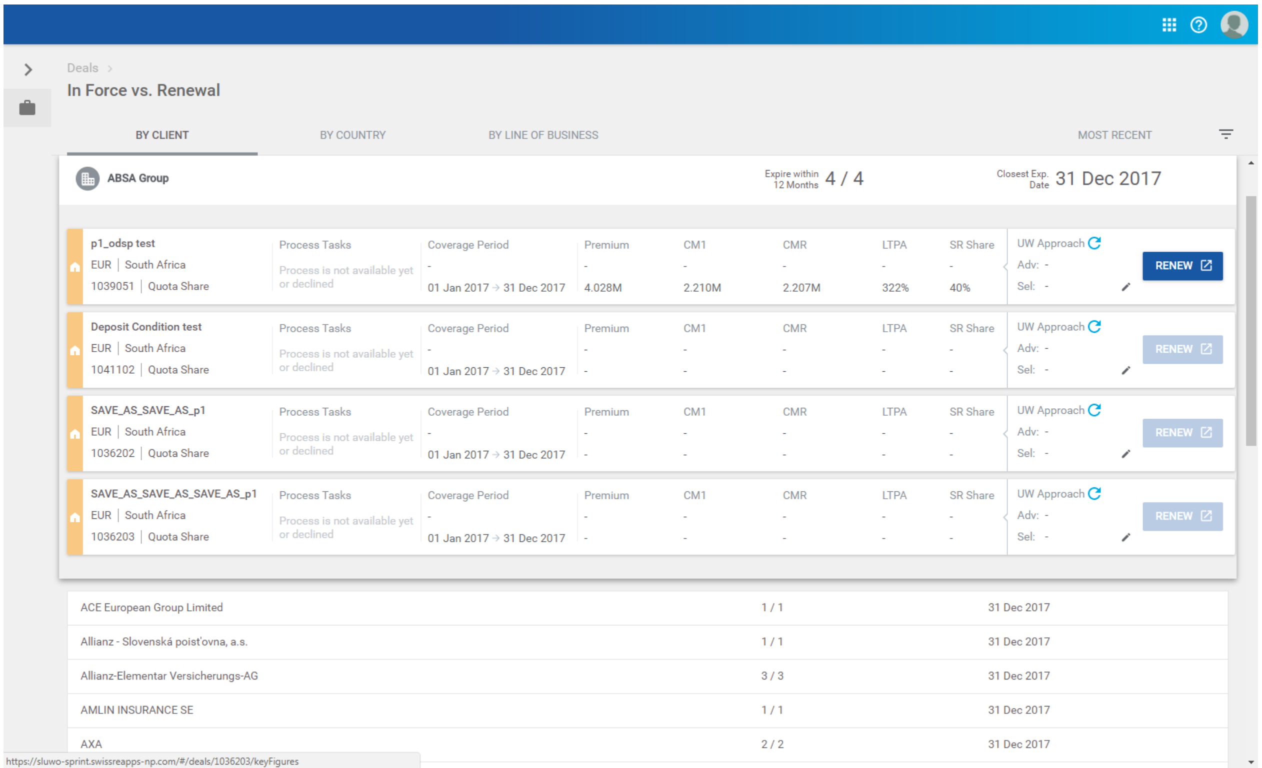
Papildus progresa indikatoriem, gadījumos, kad gaidīšanas laiks ir liels, ne slikta doma ir parādīt kādu papildus informāciju, piemēram, “Vai jūs zinājāt ka mūsu lietotnes pēdējā versijā tika ieviesta funkcionalitāte A ar kuras palīdzību jūs varat sasniegt rezultātu B”. Tādā veidā lietotājs pārslēgsies no pasīvās fāzēs uz aktīvo.

Cits variants ir ļaut veikt lietotājam, kāda izpildes procesa laikā, papildus darbības. Tā, piemēram, ja lietotājs augšupielādē kādu datni, lietotājam varētu parādīt ne tikai progresa indikatoru ar augšupielādes statusu, bet arī teksta formu, kurā varētu ievadīt augšupielādējamās datnes aprakstu (protams, ja ir tāda nepieciešamība). Arī šajā gadījumā pasīvā fāze tiks aizstāta ar aktīvo.

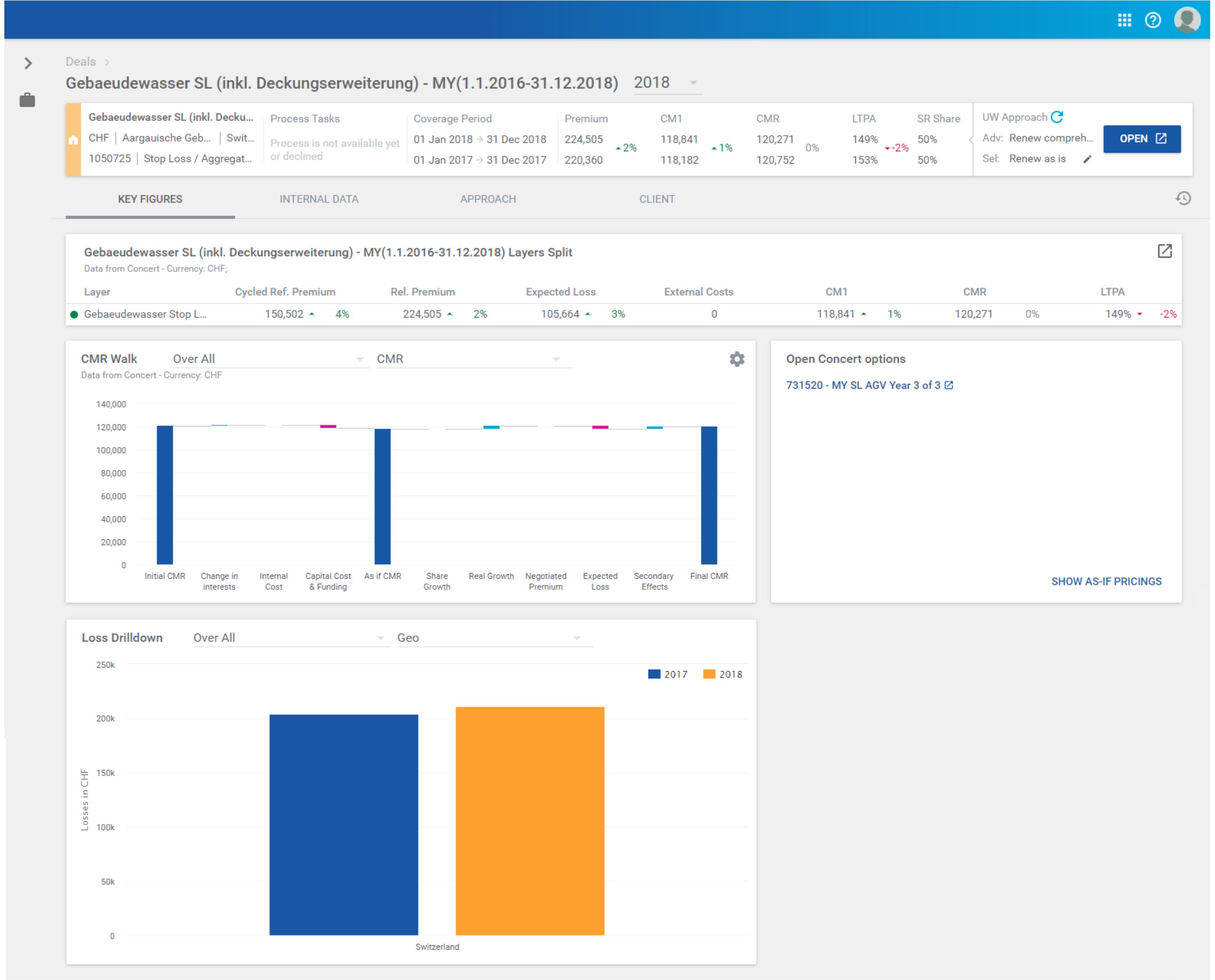
7. DARBA PRAKTISKĀ DAĻA

Šī darba izstrādes ietvaros tika mēģināts pielietot dažās ātrdarbības optimizācijas metodes praktiski. Optimizācija tikai veikta projekta "S&L CORE" izstrādājamai tīmekļa vietnei. Tīmekļa vietne tiek izstrādāta vienai no Šveices pārapdrošināšanas kompānijām, ar mērķi atvieglot darbu līgumu parakstītajiem, balstoties uz iepriekšējos gados parakstītajiem līgumiem.

Izstrādājamās lietotnes klienta puses daļu var iedalīt divās galvenās daļās – plānošanas skats, kurā ir redzams līgumu pārskats saraksta veidā, piemērs redzams 7.1.attēlā, un viena konkrēta līguma detalizēts pārskats, piemērs attēlā 7.2.



7.1. att. **Plānošanas skats**



7.2. att. **Līguma detalizētā pārskatā skats**

Tīmekļa vietnes klienta daļas izstrādei tiek izmantota skriptēšanas valoda TypeScript, kas ir JavaScript valodas papildinājums, kurs ļauj izmantot tipizāciju izstrādes laikā. TypeScript valoda tiek pārveidota JavaScript valodā, un pārlūkprogrammā nonāk jau kā JavaScript, tāpēc, tālāk, šī darba ietvaros, TypeScript atsevišķi netiks izcelts.

Ātrākai un ērtākai izstrādei tiek izmantots React ietvars. Papildus tiek izmantots Webpack saistītāj rīks, kurš ļauj apvienot atsevišķus JavaScript moduļus saišķos, bāzējoties uz to savstarpējām atkarībām. Tīmekļa vietnes stāvokļu kontrolēšanai tiek izmantots Redux stāvokļu konteineris, paredzēts tieši JavaScript lietotnēm.

"S&L CORE" tīmekļa vietnei tika konstatētas vairākas ātrdarbības problēmas. Viena no tām saistīta ar lielāku sarakstu atspoguļošanu. Otra lielāka problēma ir ilgas atbildes no servera puses, turklāt, arhitektūras specifikas dēļ, šo atbilžu ātruma uzlabošana ir ļoti ierobežota, par cik, nepieciešamie dati tiek krāti no daudzām citām ārējām sistēmām, kā rezultātā, ātrdarbība servera

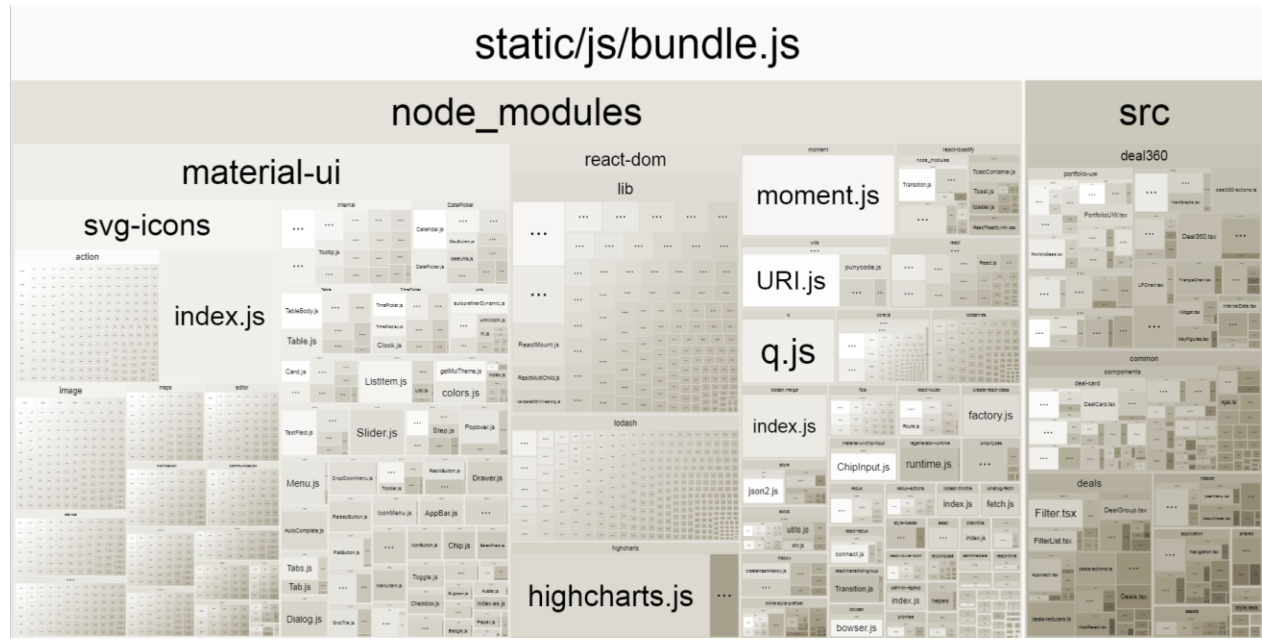
pusē ir atkarīga no šo ārējo sistēmu ātrdarbības. Rezultātā rodas nepieciešamība klienta pusē uzlabot ātrdarbību, balstoties uz cilvēku laika uztveri.

Strādājot pie tīmekļa vietnes klienta puses ātrdarbības optimizēšanas, protams, atradās vieta arī tādām metodēm, kā satura optimizācija, resursu pārvalde un kritiskā renderēšanas ceļa samazināšana.

7.1. Satura optimizācija

Liela uzmanība tika vērsta JavaScript datņu izmēru samazināšanai. Plašas iespējas šajā ziņā sniedz Webpack saistītāj rīks. Ar tā palīdzību ir iespējams veikt gan datu saspiešanu, gan koda minifikāciju, kā arī tas dod iespēju sadalīt eksistējošo kodu atsevišķos saišķos.

Priekš projektiem, kuru datnes tiek savā starpā saistītas ar Webpack rīku, ir izstrādāta bibliotēka webpack-bundle-analyzer, kas ļauj apskatīties visas datnes, kuras tiek izmantotas projekta ievaros, kā šīs datnes tiek dalītas atsevišķos koda saišķos, šo saišķu un datņu izmērus.



7.3. att. **"S&L CORE" projekta struktūra webpack-bundle-analyzer rīkā pirms optimzācijas**

Attēlā 7.3. redzams, ka tiek veidots viens JavaScript koda saišķis "bundle.js", kurš satur sevī gan paša projekta kodu, gan tā izmantojamo bibliotēku kodu. Šī saišķa izmērs ir 6,5 megabaiti. Par cik, šāda izmēra datnes lejupielāde var aizņemt salīdzinoši ilgu laiku, tika mēģināts šo koda saišķi

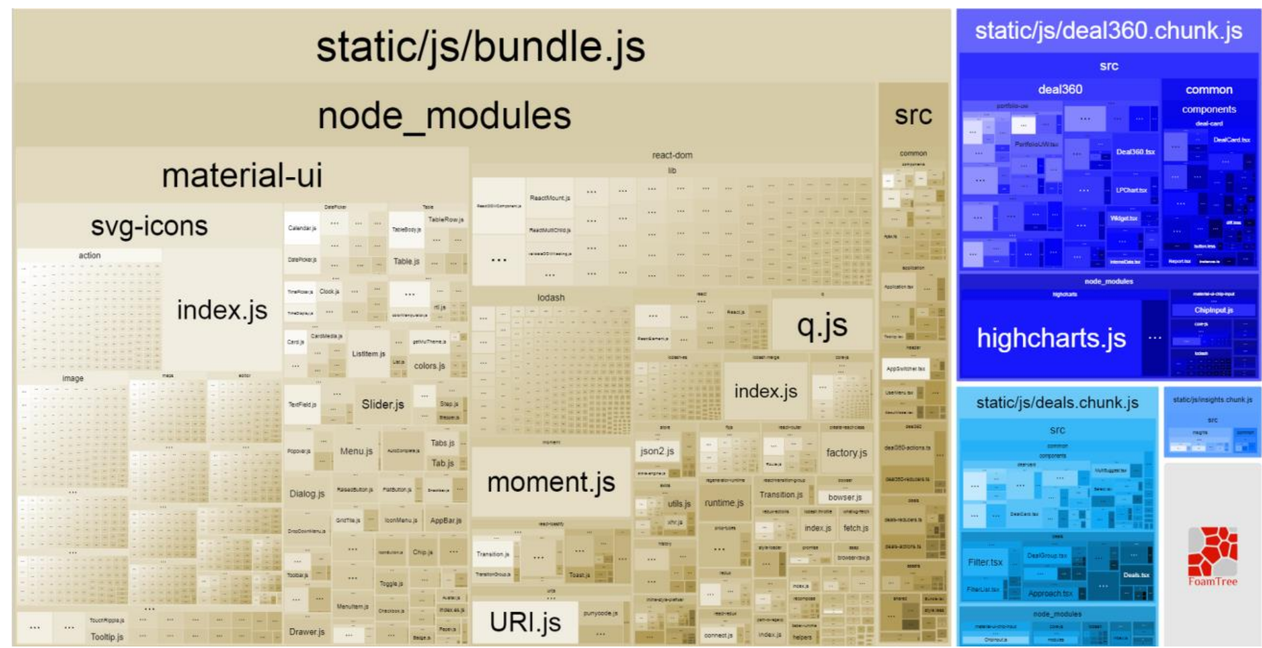
sadalīt vairākos, mazāka izmēra saišķos, kuri pārlūkprogrammā tiks lejupielādēti atkarībā no tā, uz kuru skatu lietotājs ir navigējis.

Pirmajā pielikumā ir iespējams apskatīties React komponentes "Bundle" piemēru, kas tiek izmantota citu komponenšu ietīšanai, ar mērķi, izveidot no tām atsevišķus koda saišķus.

Name	Status	Type	Initiator	Size	Time
bundle.js	200	script	(index)	6.5 MB	13.46 s

7.4. att. **Saišķa "bundle.js" lejupielāde pārlūkprogrammā**

Sākotnējais saišķis "bundle.js" tika sadalīts četros mazākos saišķos. Optimizētajā variantā saišķis "bundle.js" satur kodu, kurš tiek izmantos visos tīmekļa vietnes skatos. Papildus tika izveidots saišķis "deal360.chunk.js", tas satur kodu, kas nepieciešams tikai līgumu detalizētā pārskata skatam, "deals.chunk.js" – satur kodu, kas nepieciešams plānošanas skatam, un saišķis "insights.chunk.js", kurš satur kodu statistikas skatam. Šie saišķi redzami attēlā 7.5.



7.5. att. ***"S&L CORE" projekta struktūra webpack-bundle-analyzer rīkā pēc optimizācijas***

Saišķis "bundle.js" tiek lejupielādēts vienmēr, savukārt pārējie trīs saišķi tiek lejupielādēti atkarībā no tā, kurā skatā lietotājs ir nonācis.

Vērts minēt, ka jaunizveidotie saišķi satur ne tikai projekta ietvaros rakstīto kodu, bet arī to bibliotēku kodu, kuras tiek izmantotas tikai kādā noteiktā skatā. Tā, attēlā 7.5. redzams, piemēram, ka grafiku zīmēšanas bibliotēka Highcharts tiek izmantota tikai saišķī "deal360.chunk.js".

bundle.js	200	script	(index)	5.3 MB	10.93 s
Roboto-Medium.fe13e417.ttf	200	font	C:\dev\sluw...	159 KB	950 ms
Roboto-Regular.ac3f799d.ttf	200	font	C:\dev\sluw...	159 KB	943 ms
deals.chunk.js	200	script	C:\dev\sluw...	533 KB	1.82 s

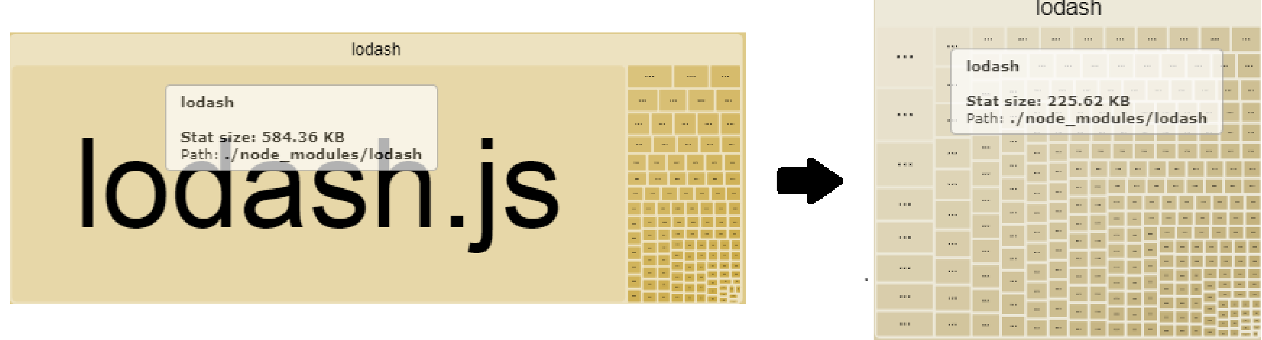
7.6. att. **Saišķa "bundle.js" un "deals.chunk.js" lejupielāde plānošanas skatā**

Mēģinot samazināt tīmekļa lietotnes koda lielumu, tika izmainīta bibliotēkas Lodash izmantošana. Sākotnēji, "bundle.js" koda saišķī tika pievienota visa Lodash bibliotēka. Bet, izrādās, ka pastāv iespēja, izmantot tikai atsevišķus šīs bibliotēkas moduļus.

```
//General usage
import {find} from 'lodash';
import {map} from 'lodash';
//Modular usage
import find from 'lodash/find';
import map from 'lodash/map';
```

7.7. att. **Bibliotēkas Lodash vispārējas un modulāras izmantošanas piemērs**

Nomainot Lodash izmantošanas pieeju visā tīmekļa vietnes kodā, pēc principa, kā tas redzams attēlā 7.7., kopējais Lodash koda lielums tika samazināts vairāk kā divas reizes – no 584.36 KB līdz 225.62 KB.



7.8. att. **Bibliotēkas Lodash sākotnējais un optimizētais izmērs**

Ne tik daudzas bibliotēkas piedāvā atsevišķu to moduļu izmantošanu, bet izstrādātājam noteikti vajadzētu pārbaudīt, vai viņa izmantotajām bibliotēkām pastāv šāda iespēja, jo, tādā veidā var būtiski samazināt tīmekļa lietotnei nepieciešamā koda apjomu.

Veicot Webpack rīka pareizu konfigurāciju, ar tā palīdzību, pie koda saišķu izveides, iespējams veikt arī datu saspiešanu ar gzip algoritmu, kā arī ir iespējams veikt koda minificēšanu.

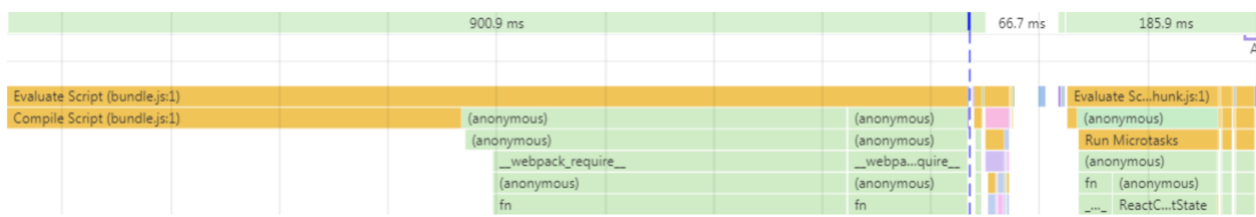
7.1. tabula

Koda saišķu izmēri, pielietojot datu saspiešanu un koda minificēšanu

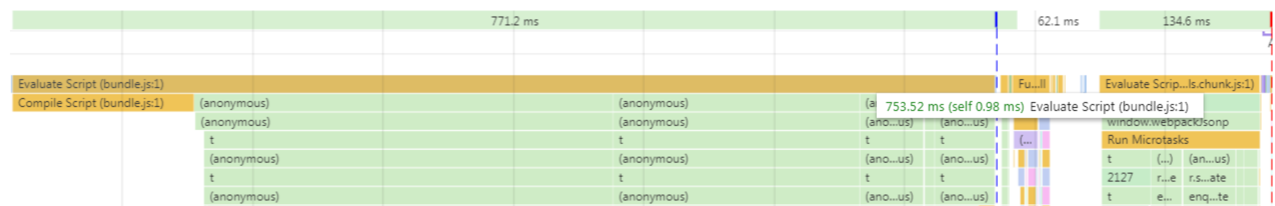
	bundle.js	deals.chunk.js	deal360.chunk.js	insights.chunk.js
Sākotnējais izmērs	5427 KB	533 KB	1024 KB	73.3 KB
Izmērs pēc gzip pielietošanas	897 KB	74.3 KB	179 KB	9.6 KB
Saspiešanas koeficients pielietojot gzip	83.47 %	86.06 %	82.52 %	82.9 %
Izmērs pēc koda minificēšanas	1740.8 KB	168 KB	461 KB	22.2 KB
Saspiešanas koeficients pielietojot minificēšanu	67.92 %	68.48 %	54.98 %	69.71 %
Izmērs pēc gzip un minificēšanas pielietošanas	413 KB	40.9 KB	127KB	5.6KB
Saspiešanas koeficients pielietojot gzip un minificēšanu	92.39 %	92.33 %	87.6 %	92.36 %

Kā redzams 7.1. tabulā, datu saspiešana ar gzip algoritmu un koda minificēšana ļauj būtiski samazināt datu apjomu. Var pat teikt, ka datu saspiešana un koda minificēšana ir primārā lieta, uz kuru izstrādātājam vajadzētu vērst uzmanību, kad runa iet par tīmekļa lietotņu ātrdarbības optimizāciju. Webpack konfigurācijas koda fragmenti ir atrodami šī darba otrajā pielikumā.

Veicot eksperimentus ar koda minificēšanu, tika novērota arī tāda interesanta lieta, kā laika samazināšanās, kas pārlūkprogrammai nepieciešams, lai veiktu skriptu parsēšanu.



7.9. att. **Saišķu “bundle.js” un “deals.chunk.js” parsēšanas laiks pirms minificēčanas**



7.10. **Saišķu "bundle.js" un "deals.chunk.js" parsēšanas laiks pēc minificēčanas**

Tā, 7.9. attēlā redzams, ka neminificētā saišķa "bundle.js" parsēšanai nepieciešamais laiks ir aptuveni 900 milisekundes, un saišķa "deals.chunks.js" parsēšanai – aptuveni 186 milisekundes. Savukārt attēlā 7.10. ir redzami laiki, kas nepieciešami augstāk minēto saišķu minificēto versiju parsēšanai – aptuveni 771 milisekunde un 135 milisekundes attiecīgi.

Šāda skriptu parsēšanas laika uzlabošana varētu būt skaidrojama ar to, ka koda minificētajā versijā, lauku un metožu nosaukumi ir krietni īsāki, tādā veidā, pārlūkprogrammai nepieciešams mazāk laika to nolasīšanai un citu manipulāciju veikšanai. Negaidīta, bet patīkama ātrdarbības uzlabošana.

7.2. Resursu pārvalde

Pavisam neliela optimizācija tika veikta izmantojot resursu pārvaldes metodes. Par cik, lielākā daudzumā gadījumu, lietotāji pāriet no plānošanas skata uz līguma detalizēto pārskatu, tika ieviesta priekšnolases metode, kas tika aprakstīta darba 5.3. nodaļā, priekš saišķa "deal360.chunk.js", kas nepieciešams līguma detalizētajam pārskatam,

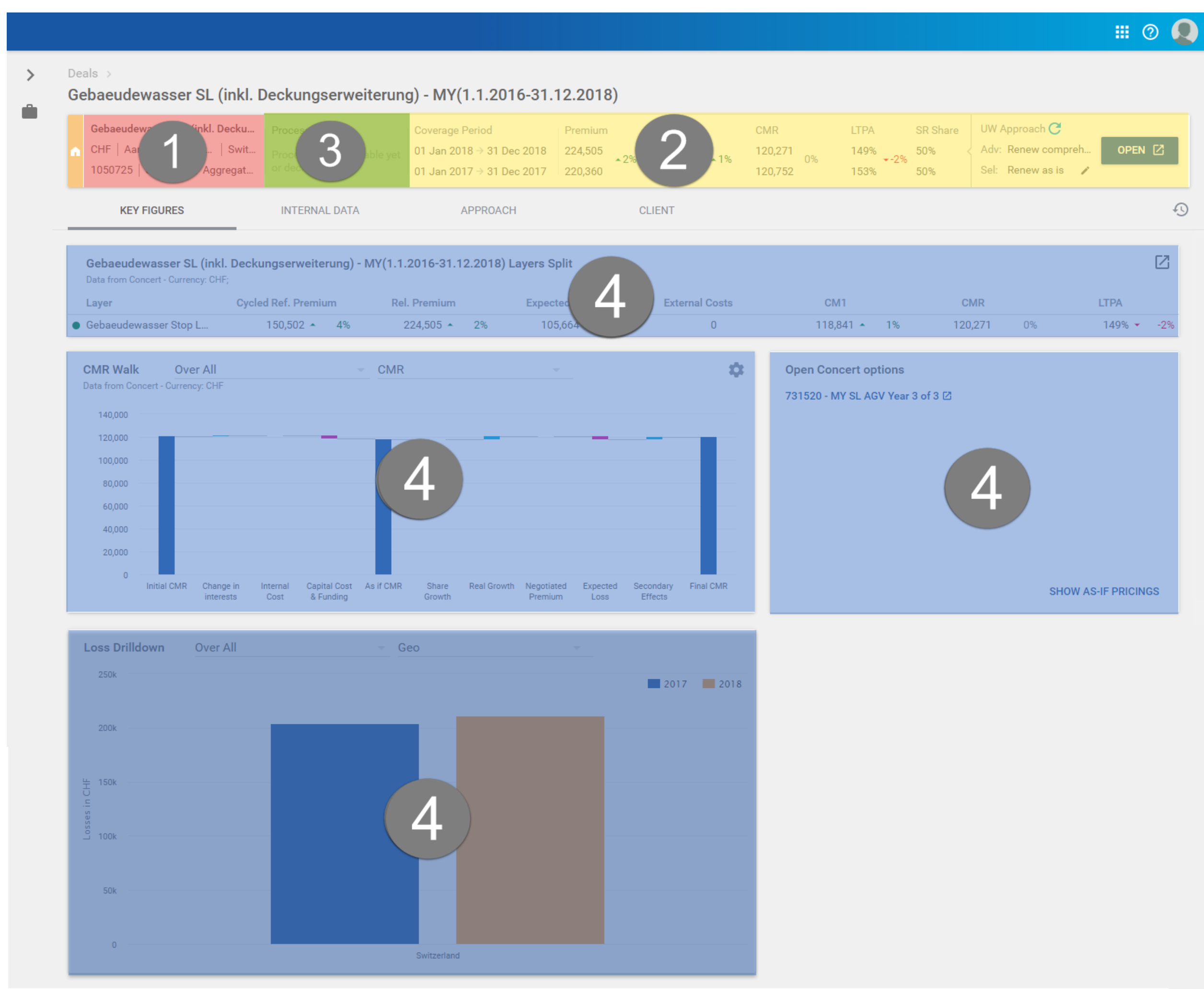
Name	Status	Type	Initiator	Size	Time
bundle.js	200	script	(index)	413 KB	941 ms
deals.chunk.js	200	script	C:\dev\sluwo-advisor-cli...	40.9 KB	241 ms
Roboto-Medium.fe13e417.ttf	200	font	C:\dev\sluwo-advisor-cli...	85.5 KB	364 ms
Roboto-Regular.ac3f799d.ttf	200	font	C:\dev\sluwo-advisor-cli...	85.2 KB	611 ms
approachTypeDocuments	200	xhr	C:\dev\sluwo-advisor-cli...	1.9 KB	1.23 s
filters	200	xhr	C:\dev\sluwo-advisor-cli...	1.5 KB	1.23 s
commentReasonTypeDocuments?sort=order	200	xhr	C:\dev\sluwo-advisor-cli...	2.2 KB	1.28 s
getActual?projection=simple	200	xhr	C:\dev\sluwo-advisor-cli...	1.4 KB	1.20 s
profile	200	xhr	C:\dev\sluwo-advisor-cli...	1.4 KB	1.30 s
deal360.chunk.js	200	javascript	C:\dev\sluwo-advisor-cli...	127 KB	505 ms

7.11. att. **Optimizēto saišķu lejupielāde un saišķa "deal360.chunk.js" priekšnolase**

Tādā veidā, lietotājam, pārejot no plānošanas skata uz līguma detalizētā pārskata skatu, ir iespējams bez īpašām aizturēt parādīt šī skata saturu, tādā veidā samazinot kritisko renderēšanas ceļu, par cik, pārlūkprogramma nolasīs saišķi "deal360.chunk.js" no kešatmiņas.

7.3. Pieprasījumu optimizācija

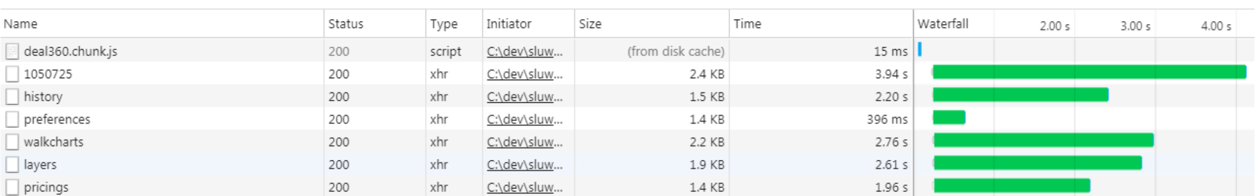
Liela uzmanība tika vērsta paralēlai pieprasījumu sūtīšanai uz serveri. Tīmekļa vietnē tika meklētas iespējas veikt vienu vai otru pieprasījumu pēc iespējas ātrāk, kā arī, iespēju robežās, veikt pieprasījumus paralēli, gadījumā ja tie nav atkarīgi no citu pieprasījumu atgrieztajiem rezultātiem.



7.12. att. **Līguma detalizētā pārskata pieprasījumu secība pirms optimizācijas**

Lielākās problēmas ar servera pieprasījumu secību bija līguma detalizētā pārskata lapā. Attēlā 7.12. ir parādīti četri pieprasījumu posmi. Sākumā tika secīgi veikti augšējā logrīka nepieciešamo datu vaicājumi trijos posmos, un tikai tad tika veikti paralēli pieprasījumi datiem, kas nepieciešami logrīkiem, kas apzīmēti ar ceturto numuru.

Par cik, tieši šādai pieprasījumu hierarhijai netika atrasts iemesls, tika veikta šo pieprasījumu optimizācija.



Name	Status	Type	Initiator	Size	Time
deal360.chunk.js	200	script	C:\dev\sluw...	(from disk cache)	15 ms
1050725	200	xhr	C:\dev\sluw...	2.4 KB	3.94 s
history	200	xhr	C:\dev\sluw...	1.5 KB	2.20 s
preferences	200	xhr	C:\dev\sluw...	1.4 KB	396 ms
walkcharts	200	xhr	C:\dev\sluw...	2.2 KB	2.76 s
layers	200	xhr	C:\dev\sluw...	1.9 KB	2.61 s
pricings	200	xhr	C:\dev\sluw...	1.4 KB	1.96 s

7.13. att. **Līguma detalizētā pārskata pieprasījumu secība pārlūkprogrammā pēc optimizācijas**

7.14. att. **Līguma detalizētā pārskata pieprasījumu secība pirms optimizācijas**

Attēlā 7.13 un 7.14. ir redzama logrīku, vai to daļu, optimizētā pieprasījumu secība. Šīs izmaiņas prasīja nopietnas izmaiņas tīmekļa vietnes arhitektūrā, par cik ne labākajā veidā tika veikti un apstrādāti sākotnējie izsaukumi. Bet neskatoties uz to, tika sasniegts diezgan nopietns ātrdarbības uzlabojums - šīs lapas apakšējo daļu kļuva iespējams parādīt, vidēji par aptuveni piecām sekundēm, ātrāk.

Attēlā 7.14. ir arī redzams, ka pārlūkprogramma paņem no kešatmiņas priekšnolasīto koda saišķi “deal360.chunk.js” –tas aizņēma vien 15 milisekundes.

7.4. JavaScript optimizācija

Šī darba ietvaros, mērot tīmekļa vietnes ātrdarbību pie dažādām lietotāja darbībām, plānošanas skatā, tika atrasta kritiska vieta JavaScript izpildes ātrdarbībā, pie līgumu grupu atvēršanas, gadījumos, kad grupas saturēja lielu daudzumu līgumu.

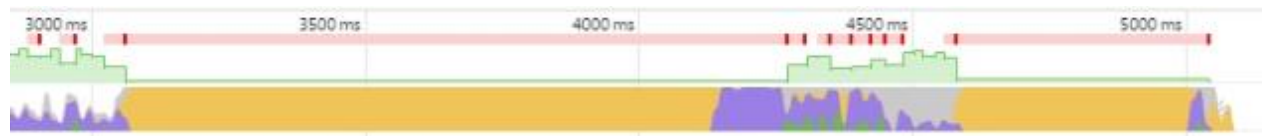
7.15. att. **Neoptimizēta līgumu grupas atvēršana, kas satur 35 līgumus**

Attēlā 7.15. ir redzams grupas, kas sastāv no 35 līgumiem. atvēršanās laiks. Attēlā ar dzelteno joslu ir norādīta JavaScript izpilde. Kā redzams, skripts aktīvi izpildās vairāk par 16 sekundēm. Visu šo laiku, lietotājam nebija iespējas veikt jebkādas citas darbības tīmekļa lietotnē, jo ekrāns tika bloķēts – pārlūkprogramma spēja attēlot ne vairāk par vienu kadru sekundē.

Problēma slēpās nepareizā React ietvara izmantošanā. Atverot līgumu grupu, tika veikti divi pieprasījumi serverim, katram pieprasījumam tika atgriezta atbilde, kas sastāvēja no 35 ierakstiem. Apstrādājot atbildi, grupai tika padota informācija par katru no šiem ieraktiem atsevišķi, tādā veidā, React ietvars, uz katru informācijas atjauninājumu, lika pārzīmēt visu grupu ar visiem tās 35 līgumiem. Sanāk, ka sākotnēji tika uzzīmēti 35 līgumi, tad katrs no šiem līgumiem tika pārzīmēts vēl 35 reizes, pēc informācijas apstrādes, kas tika saņemta no pirmā pieprasījuma, un vēl 35 reizes

pēc saņemtās informācijas no otrā pieprasījuma. Rezultātā, lai parādītu grupu, kas saturēja 35 līgumus, pārlūkprogramma veica 42875 līgumu zīmēšanas operācijas.

Kods tika izmainīts tādā veidā, ka sākumā tika apstrādāti visi no servera puses saņemtie ieraksti, un tikai tad informācija tika nodota grupai, tādā veidā izdevās izvairīties no nevajadzīgo pārzīmēšanu lielā skaita.



7.16. att. **Optimizēta līgumu grupas atvēršana, kas satur 35 līgumus**

Attēlā 7.16. redzams, ka grupas, ar 35 līgumiem, atvēršanās laiks pēc optimizācijas, aizņem aptuveni divas sekundes. Tas arī ir tālu no ideāla varianta, bet atšķirība ir acīmredzama.

Saskaroties ar augstāk minēto ātrdarbības problēmu, kļuva skaidra pavisam vienkārša lieta – izstrādātājam ir pilnībā jāizprot kā darbojas rīki, kuri tiek izmantoti izstrādē. Tā, šajā gadījumā, problēma visdrīzāk radās, tikai tāpēc, ka netika pilnībā pārzināti React ietvara komponenšu dzīves cikli.

REZULTĀTI UN SECINĀJUMI

Darbs tika izstrādāts vadoties pēc sākumā izvirzītā mērķa – aplūkot veidus, ar kuru palīdzību ir iespējams optimizēt tīmekļa vietņu ātrdarbību. Darba autors šī mērķa sasniegšanai izpētīja daudzu dažādus literatūras un interneta informācijas avotus, veica šo avotu analīzi un salīdzināšanu.

Darba ietvaros tika aplūkotas tādas tēmas kā tīmekļa lietotņu satura optimizācija, kritiskā renderēšanas ceļa optimizācija, renderēšanas ātrdarbība, resursu pārvaldība. Nozīmīga darba daļa tika veltīta arī ātrdarbības optimizācijai, balstoties uz cilvēku laika uztveri. Tika aplūkoti arī iemesli, kāpēc tīmekļa vietņu ātrdarbības optimizācija ir nozīmīga un aktuāla.

Darbā tika sniegta informācija par galvenajiem veidiem, ar kuru palīdzību varētu uzlabot tīmekļa vietņu ātrdarbību, precīzāk, lejupielādes laiku, sākotnējai renderēšanai nepieciešamā laika samazināšanu un renderēšanai nepieciešamo procesu izpildes laiku.

Darba ietvaros vairākas no apskatītajām ātrdarbības uzlabošanas metodēm, tika pielietotas praktiski. JavaScript kods tika sadalīts vairākos saišķos, atsevišķiem saišķiem tika pielietota priekšnolase, tādā veidā tika samazināts kritiskais renderēšanas ceļš. Tika veikta datu saspiešana izmantojot gzip algoritmu, kā arī tika veikta koda minifikācija, kas rezultātā ļāva samazināt JavaScript datņu izmērus aptuveni par 90%. Tika veiktas optimizācijas pieprasījumiem, tādā veidā tika samazinās gaidīšanas laiks un šķietami samazināta lietotnes ātrdarbība. Tikai veiktas arī JavaScript optimizācijas, kas ļāva paātrināt procesu izpildi pārlūkprogrammā.

Līdz ar šo, darba izvirzītie mērķi tiek uzskatīti par sasniegtiem.

Darba autors secina, ka tīmekļa lietotņu ātrdarbības optimizācija ir ļoti svarīga, it īpaši, lietotnēm, kas ir saistītas ar e-komerciju, jo, lietotnes ātrdarbība, lai arī netieši, bet būtiski ietekmē kompāniju peļņu.

Ne mazāk svarīga ātrdarbība ir tīmekļa lietotnēm, kurās tiek veiktas sarežģītas un apjomīgas manipulācijas ar vizuāliem, ekrānā redzamajiem elementiem, jo, lai vizuālās izmaiņas tiktu veiktas gludi, bez raustīšanās, ir sagaidāms, ka lietotne spēs atrādīt 60 kadrus sekundē, kas ir lielākās ierīču daļas ekrāna atjaunošanās ātrums, bet tas var būt nopietns izaicinājums.

Ātrdarbība ir svarīga arī lietotnēm, kurās servera atbildes nav ātras, jo ir iespēja ar dažādiem vizuāliem efektiem šķietami paātrināt ātrdarbību, samazinot lietotāju pasīvo gaidīšanas laiku.

Darba izstrādes laikā, kļuva skaidrs, ka ātrdarbības optimizācija lielākoties ir ilgstošs process, un, par ātrdarbību ir nepieciešams domāt uzreiz, izstrādājot vienu vai otru funkcionalitāti, jo, jau esošās funkcionalitātes novēlota ātrdarbības optimizēšana, var būt apgrūtinoša. Eksistē arī metodes, kā, piemēram, datu saspiešana un koda minificēšana, kas ļauj, ar salīdzinoši maziem resursiem, būtiski uzlabot tīmekļa vietņu ātrdarbību.

Dzīvē, ātrdarbība, no programmatūras pasūtītāju puses, bieži vien tiek atlikta uz otro plānu, jo funkcionalitātes esamība un tās kvalitāte ir daudz svarīgāka, tomēr, pašiem izstrādātājiem, vajadzētu veikt ātrdarbības mērījumus to izstrādātajai funkcionalitātei. Svarīgi ir arī saprast, ka ātrdarbības optimizācija ir process, kurš ir atkārtoti jāveic visu laiku - izstrādājām, veicam ātrdarbības mērījumus, optimizējam, un atkārtojam procesu no jauna.

IZMANTOTĀ LITERATŪRA UN AVOTI

1. Steve Souders, *High Performance Web Sites*, O'Reilly Media, 2008
2. The Growth of Web Page Size, [tiešsaite]. [Atsauce 27.12.2016] Pieejams: https://www.keycdn.com/support/the-growth-of-web-page-size/
3. Ecommerce Page Speed & Web Performance – Spring 2015, [tiešsaite]. [Atsauce 27.12.2016] Pieejams: http://www.radware.com/social/spring-sotu2015/
4. Terrence Dorsey, *Web Page Size, Speed, and Performance*, O'Reilly Media, 2014
5. The Performance Beacon, [tiešsaite]. [Atsauce 27.12.2016] Pieejams: https://www.soasta.com/blog/page-bloat-average-web-page-2-mb/
6. Николай Мациевский, Евгений Степанищев, Глеб Кондратенко, *Реактивные веб-сайты*, Интернет-университет информационных технологий, Бином. Лаборатория знаний, 2010
7. Optimizing Content Efficiency, [tiešsaite]. [Atsauce 29.12.2016] Pieejams: https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/
8. Inline Images with Data URLs, [tiešsaite]. [Atsauce 29.12.2016] Pieejams: http://www.websiteoptimization.com/speed/tweak/inline-images/
9. Peter Smith, Professional Website Performance, Wrox, 2012
10. HTTP Caching, [tiešsaite]. [Atsauce 15.01.2017]. Pieejams: https://developers.google.com/web/fundamentals/performance/optimizing-content-efficiency/http-caching
11. Ecommerce Page Speed & Web Performance – Spring 2015, [tiešsaite]. [Atsauce 15.01.2017] Pieejams: http://www.radware.com/spring-sotu2015-lpc-6442455858/
12. Critical Rendering Path, [tiešsaite]. [Atsauce 18.01.2017]. Pieejams: https://developers.google.com/web/fundamentals/performance/critical-rendering-path/
13. Consumer Response to Travel Site Performance, [tiešsaite]. [Atsauce 18.01.2017]. Pieejams: http://www.phocuswright.com/Free-Travel-Research/Consumer-Response-to-Travel-Site-Performance
14. Image Maps, [tiešsaite]. [Atsauce 18.01.2017]. Pieejams: https://www.image-maps.com/

15. Constructing the object model, [tiešsaite]. [Atsauce 19.01.2017]. Pieejams: https://developers.google.com/web/fundamentals/performance/critical-rendering-path/constructing-the-object-model
16. What is a CSS reset?, [tiešsaite]. [Atsauce 19.01.2017]. Pieejams: http://cssreset.com/what-is-a-css-reset/
17. Render-tree Construction, Layout, and Paint, [tiešsaite]. [Atsauce 19.01.2017]. Pieejams: https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction
18. Optimizing the Critical Rendering Path, [tiešsaite]. [Atsauce 20.01.2017]. Pieejams: https://developers.google.com/web/fundamentals/performance/critical-rendering-path/optimizing-critical-rendering-path
19. PageSpeed Rules and Recommendations, [tiešsaite]. [Atsauce 20.01.2017]. Pieejams: https://developers.google.com/web/fundamentals/performance/critical-rendering-path/page-speed-rules-and-recommendations
20. Rendering Performance, [tiešsaite]. [Atsauce 22.01.2017]. Pieejams: https://developers.google.com/web/fundamentals/performance/rendering/
21. Optimize JavaScript Execution, [tiešsaite]. [Atsauce 22.01.2017]. Pieejams: https://developers.google.com/web/fundamentals/performance/rendering/optimize-javascript-execution
22. Reduce the Scope and Complexity of Style Calculations, [tiešsaite]. [Atsauce 22.01.2017]. Pieejams: https://developers.google.com/web/fundamentals/performance/rendering/reduce-the-scope-and-complexity-of-style-calculations
23. Avoid Large, Complex Layouts and Layout Thrashing, [tiešsaite]. [Atsauce 22.01.2017]. Pieejams: https://developers.google.com/web/fundamentals/performance/rendering/avoid-large-complex-layouts-and-layout-thrashing
24. CSS Triggers, [tiešsaite]. [Atsauce 22.01.2017]. Pieejams: https://csstriggers.com/
25. Simplify Paint Complexity and Reduce Paint Areas, [tiešsaite]. [Atsauce 22.01.2017]. Pieejams: https://developers.google.com/web/fundamentals/performance/rendering/simplify-paint-complexity-and-reduce-paint-areas

26. Stick to Compositor-Only Properties and Manage Layer Count, [tiešsaite]. [Atsauce 22.01.2017]. Pieejams: https://developers.google.com/web/fundamentals/performance/rendering/stick-to-compositor-only-properties-and-manage-layer-count
27. Server-Side and Client-Side Image Maps, [tiešsaite]. [Atsauce 23.01.2017]. Pieejams: http://www.osec.doc.gov/webresources/accessibility/RuleEF.htm
28. Preconnect, Prefetch, Prerender…, [tiešsaite]. [Atsauce 23.11.2017]. Pieejams: https://docs.google.com/presentation/d/18zlAdKAxnc51y_kj-6sWLmnjl6TLnaru_WH0LJTjP-o/present?slide=id.p19
29. Prefetching, preloading, prebrowsing, [tiešsaite]. [Atsauce 23.11.2017]. Pieejams: https://css-tricks.com/prefetching-preloading-prebrowsing/
30. Intent to Deprecate and Remove: Prerender, [tiešsaite]. [Atsauce 01.12.2017]. Pieejams: https://groups.google.com/a/chromium.org/forum/#!topic/blink-dev/0nSxuuv9bBw
31. CSS @media Rule, [tiešsaite]. [Atsauce 02.12.2017]. Pieejams: https://www.w3schools.com/cssref/css3_pr_mediaquery.asp
32. Weber's Law of Just Noticeable Differences, [tiešsaite]. [Atsauce 05.12.2017]. Pieejams: http://apps.usd.edu/coglab/WebersLaw.html
33. Weber-Fechner law, [tiešsaite]. [Atsauce 05.12.2017]. Pieejams: http://psychology.wikia.com/wiki/Weber-Fechner_law
34. Steven C. Seow, *Designing and Engineering Time: The Psychology of Time Perception in Software*, Addison-Wesley Professional, 2008
35. Throwing a glance at the neural code: rapid information transmission in the visual system, [tiešsaite]. [Atsauce 06.12.2017]. Pieejams: https://www.ncbi.nlm.nih.gov/pmc/articles/PMC2689612/
36. Mihaly Csikszentmihalyi, *Flow: The Psychology of Optimal Experience*, Harper Perennial, 1990
37. How Our Brains Perceive Time: The Study of Time Perception, [tiešsaite]. [Atsauce 10.12.2017]. Pieejams: http://www.healthguidance.org/entry/17225/1/How-Our-Brains-Perceive-Time-The-Study-of-Time-Perception.html
38. Martin Heidegger, *Being and Time*, Harper & Row, 1962

39. Richard C. Larson, *Perspectives on Queues: Social Justice and the Psychology of Queueing*, INFORMS, 1987
40. Jacob Hornik*, Subjective vs. Objective Time Measures: A Note on the Perception of Time in Consumer Behavior,* Oxford University Press, 1984
41. How Tolerable is Delay? Consumers' Evaluations of Internet Web Sites after Waiting, [tiešsaite]. [Atsauce 11.12.2017]. Pieejams: http://citeseerx.ist.psu.edu/viewdoc/download?doi=10.1.1.40.6086&rep=rep1&type=pdf
42. Why Performance Matters, Part 3: Tolerance Management, [Atsauce 11.12.2017]. Pieejams: https://www.smashingmagazine.com/2015/12/performance-matters-part-3-tolerance-management/

PIELIKUMI

React komponente, saišķu izveidošanai ar Webpack, un tās izmantošanas piemērs

```
class Bundle extends React.Component<BundleProps, BundleState> {
  state = {
    mod: null,
    showLoading: false,
  };

  componentWillMount() {
    this.load(this.props);
  }

  componentWillReceiveProps(nextProps: BundleProps) {
    if (nextProps.load !== this.props.load) {
      this.load(nextProps);
    }
  }

  load(props: BundleProps) {
    this.setState({ mod: null, showLoading: false });

    setTimeout(() => this.setState({ showLoading: true }), 300);

    props.load((mod) => {
      this.setState({
        mod: mod.default ? mod.default : mod,
      });
    });
  }

  render() {
    return this.state.mod ? (this.props.children as any)(this.state.mod) : this.state.showLoading ?
      <Loader/> : null;
  }
}

const Deal360 = props => (
  <Bundle load={require('bundle-loader?lazy&name=deal360!../deal360/Deal360')}>
    {Component => <Component {...props}/>}
  </Bundle>
);

const DealsRoute = props => (
  <Bundle load={require('bundle-loader?lazy&name=deals!../deals/DealsRoute')}>
    {Component => <Component {...props}/>}
  </Bundle>
);

const InsightsRoute = props => (
  <Bundle load={require('bundle-loader?lazy&name=insights!../insights/InsightsRoute')}>
    {Component => <Component {...props}/>}
  </Bundle>
);
```

2. Pielikums

Webpack konfigurācijas fragments, koda minificēšanai

```
plugins: [
  new HtmlWebpackPlugin({
    inject: true,
    template: paths.appHtml,
    minify: {
      removeComments: true,
      collapseWhitespace: true,
      removeRedundantAttributes: true,
      useShortDoctype: true,
      removeEmptyAttributes: true,
      removeStyleLinkTypeAttributes: true,
      keepClosingSlash: true,
      minifyJS: true,
      minifyCSS: true,
      minifyURLs: true,
    },
  }),

  new webpack.optimize.UglifyJsPlugin({
    compress: {
      warnings: false,
      comparisons: false,
    },
    mangle: {
      except: ['process'],
    },
    output: {
      comments: false,
    },
    sourceMap: true,
  }),
]
```

Maģistra darbs „Tīmekļa lietotņu ātrdarbības optimizācija” izstrādāts LU Datorikas fakultātē.

Darba teksta galīgā versija izgatavota 18.01.2018.

Ar savu parakstu apliecinu, ka pētījums veikts patstāvīgi, izmantoti tikai tajā norādītie informācijas avoti un iesniegtā darba elektroniskā kopija atbilst izdrukai.

Autors: Ēriks Šarapovs _____________________ **__.01.2018**.
(Autora paraksts un datums)

Ar savu parakstu apliecinu, ka esmu lasījis augstāk minēto maģistra darbu un atzīstu to par **p i e m ē r o t u / n e p i e m ē r o t u** (nevajadzīgo svītrot) aizstāvēšanai Latvijas Universitātes datorzinātņu maģistrantūrā.

Darba vadītājs: Asociētais profesors, Dr. dat., Darja Solodovņikova **______________ __.01.2018.**
(Vadītāja paraksts un datums)

Darbs iesniegts maģistratūras sekretariātā ___________________.
(Iesniegšanas datums)

Ar šo es apliecinu, ka darba elektroniskā versija ir augšupielādēta LU informatīvajā sistēmā.

Studiju metodiķe: ___________________________
(Metodiķes paraksts)

Recenzents: Docents, Dr.dat., Leo Trukšāns
(Akad. amats, zin.grāds, vārds, uzvārds)

Darbs aizstāvēts maģistra gala pārbaudījuma komisijas sēdē

__________________________ prot. Nr. **__________**
(Darba aizstāvēšanas datums)

Komisijas sekretārs: **____**
(Sekretāra paraksts)