LATVIJAS UNIVERSITĀTE

DATORIKAS FAKULTĀTE

VIENAS LAPAS TĪMEKĻA LIETOTŅU VEIKTSPĒJA

BAKALAURA DARBS

Autors: Gustavs Helmuts Felsbergs

Studenta apliecības Nr. gf13003

Darba vadītājs: Dr. dat. Uldis Straujums

RĪGA 2018

ANOTĀCIJA

Bakalaura darbā aprakstītas veiktspējas problēmas, kas var rasties vienas lapas tīmekļa lietotnēs, un tiek apskatīti to iespējamie risinājumi. Darbā veiktspējas problēmas un to iespējamie risinājumi tiek meklēti dažādos literatūras avotos. Tiek veidoti praktiski piemēri un veikti veiktspējas mērījumi, lai analizētu to, cik lielā mērā tiek ietekmēta veiktspēja.

Ņemot vērā to, ka tīmekļa pārlūkprogrammās regulāri tiek veikti uzlabojumi, izveidotie praktiskie piemēri parādīs to, vai konkrētas veiktspējas problēmas vēl arvien ir aktuālas. Izveidotie praktiskie piemēri ir lasītājam pieejami, lai tos varētu pārbaudīt dažādās ierīcēs un pārlūkprogrammās. Darbā tiek izveidots praktiski pārbaudīto veiktspējas problēmu apkopojums un ieteikumi to novēršanai, lai dotu vienas lapas tīmekļa lietotņu izstrādātājiem vadlīnijas, kurām sekot, lai izvairītos no veiktspējas problēmām.

Atslēgas vārdi: HTML, CSS, JavaScript, vienas lapas tīmekļa lietotne, interneta pārlūkprogramma, lietotāja saskarne, animācija, tīmekļa strādnis, veiktspēja

ABSTRACT

SINGLE PAGE WEB APPLICATION PERFORMANCE

Bachelor’s thesis describes performances issues that may occur in single page web applications and their possible solutions. The performance issues and their possible solutions are gathered from various sources of literature. Practical examples and performance measurements are made to analyze the extent to which performance is affected.

Given the fact that improvements in web browsers are being made on a regular basis, the practical examples will show whether specific performance issues remain topical. Practical examples are available to the reader to be tested on various devices and web browsers. This thesis provides a compilation of practically tested performance issues and suggestions to solve them in order to give single page application developers guidelines to follow in order to avoid performance issues.

Keywords: HTML, CSS, JavaScript, single page web application, web browser, user interface, animation, web worker, performance

SATURA RĀDĪTĀJS

APZĪMĒJUMU SARAKSTS

HTML – Hiperteksta iezīmēšanas valoda

JavaScript – Programmēšanas valoda, ar ko iespējams veikt aprēķinus un realizēt dinamisku HTML saturu tīmekļa pārlūkprogrammā

CSS – Valoda, ar ko iespējams pārveidot HTML elementu vizuālo izskatu

Renderešāna – Process, kā rezultātā HTML elementi tiek grafiski attēloti tīmekļa pārlūkprogrammā

Vienas lapas tīmekļa lietotne – Šajā darbā šis apzīmējums tiek lietots kā sinonīms terminam "vienas lapas lietotne", lai būtu saprotams, ka pētījums ir saistīts ar tieši tīmekļa lietotnēm. Vienas lapas tīmekļa lietotne ir tīmekļa lietotne, kas dažādu lapu parādīšanu veic, dinamiski pārveidojot jau attēloto lapu, tā vietā, lai katru lapu ielādētu no servera, veicot lapas pārlādi

Veiktspēja – Sistēmas spēja izpildīt paredzētās funkcijas un izpildīt tās pieņemamā laikā

Tīmekļa strādnis (Web worker) – JavaScript objekts, kas spējīgs veikt aprēķinus atsevišķā pavedienā

DOM – Dokumentu objektu modelis. Tas ir tīmekļa lapas struktūras kokveida attēlojums, caur kuru iespējams piekļūt lapā esošajiem HTML elementiem

JavaScript ietvars – JavaScript bibliotēka, kas paātrina tīmekļa lietotņu izstrādi, piedāvājot jau gatavus risinājumus, un strukturē kodu, padarot to uzturamāku

Atkritumu savācējs – Automātisks mehānisms, kas atbrīvo atmiņu, kas lietotnē vairs netiek izmantota

AJAX - Metode, ar ko var sūtīt un saņemt datus no servera, neveicot pilnu lapas pārlādi. Angliski: Asynchronous JavaScript And XML

Datu piesaiste – Automātisks mehānisms, kas veic skata sinhronizēšanu ar datiem, uz kā šis skats balstās

MVC – "Model View Controller" arhitektūras šablons, kas tīmekļa vietņu izstrādē sadala kodu trīs daļās – modeļos, skatos un kontrolieros

Notikumu cikls – JavaScript operāciju virkne, ko var izraisīt kāda lietotāja darbība

Lietotāja saskarne – Grafiskā lietotnes daļa, kurā lietotājs var veikt darbības lietotnē, piemēram, veicot pogas nospiešanu

Animācija – No statiskiem objektiem veidots vizuāls kustības efekts, ko rada virkne nelielu, pakāpenisku kustību

IEVADS

Attīstoties tīmekļa pārlūkprogrammām un ierīču, kurās tās tiek izmantotas, jaudai palielinoties, vienas lapas tīmekļa lietotnes kļūst arvien plašāk izmantotas. Vienas lapas tīmekļa lietotnes ļauj sniegt lietotāja pieredzi līdzīgu kā lietotnēs, kas darbojas lokāli datorā, piemēram, Android operētājsistēmai veidotās lietotnēs. Vienas lapas tīmekļa lietotnēs ir svarīgs ne tikai lapas ielādes laiks, bet arī klienta pusē strādājošā koda veiktspēja, ņemot vērā kešdarbes attīstību un tīmekļa lietotņu veidošanu pēc progresīvās tīmekļa lietotnes principa. Dinamiska satura attēlošana notiek gan brīžos, kad lietotājs pārslēdzas starp lapām, gan brīžos, kad lietotāja darbību rezultātā jāattēlo papildus lapas saturs. Arī apjomīgu aprēķinu veikšana vienas lapas tīmekļa lietotnēs ir aktuālāka, jo ne katras izmaiņas rezultātā notiek saziņa ar serveri, lai attēlotu citu lapas saturu. Tā kā vienas lapas tīmekļa lietotnēs regulāri netiek veikta lapas pārlāde, lai attīrītu aizņemto atmiņu, atmiņas noplūdes vienas lapas tīmekļa lietotnēm ir aktuālākas, jo tās var tikt izmantotas ļoti ilgu laiku, nepārlādējot lapu, un var novest pie lēnākas lietotnes darbības, kā arī pilnīgas paredzēto funkciju izpildes nespējas.

Ņemot vērā to, ka šīs lietotnes tiek izmantotas ne tikai datoros, bet arī ierīcēs ar ierobežotiem parametriem, piemēram, viedtālruņos un planšetdatoros, ir svarīgi tas, lai šīs lietotnes spētu pildīt paredzēto funkcionalitāti un spētu to pildīt optimālā laikā, lai sniegtu pēc iespējas labāku lietotāja pieredzi.

Bakalaura darba mērķis ir teorētiski un praktiski izpētīt dažādos literatūras avotos atrodamas tīmekļa lietotņu klienta puses koda veiktspējas problēmas, kā arī papildināt iepriekšējo gadu pētījumus [53, 54], izpētot tajos neizskatīto un izskatīto papildinot. Mērķa sasniegšanai tiek veikti šādi uzdevumi:

- Izpētīt JavaScript automātisko atkritumu savākšanas mehānismu, lai noskaidrotu, vai atmiņas pārvaldīšanai vienas lapas tīmekļa lietotņu izstrādē ir nozīme. Izskatīt gadījumus, kuros iespējams neatbrīvot neizmantotu atmiņu, un pārbaudīt tos praktiski ar izveidotām testa lietotnēm, lai noskaidrotu, vai konkrētie problēmgadījumi vēl arvien ir aktuāli.
- Izpētīt iespēju veikt apjomīgus JavaScript aprēķinus, izmantojot vairākus pavedienus, un veidot testa lietotnes, lai pārbaudītu vairāku pavedienu izmantošanas efektivitāti dažādos scenārijos.
- Papildināt iepriekšējā pētījuma [53] testa lietotni dažādu DOM manipulāciju veikšanas scenāriju salīdzinājumam ar papildus scenārijiem un apjomīgākas DOM struktūras elementiem, lai sniegtu plašāku izpratni par to, cik lielā mērā atšķiras veiktspēja starp

atšķirīgiem scenārijiem, un lai dziļāk analizētu pētījumā izskatīto un papildinātu veiktos atzinumus.

- Izpētīt animāciju un lapas satura ritināšanas veiktspējas problēmgadījumus, izpētot literatūras avotus un veidojot testa lietotnes veiktspējas mērījumiem, lai pārbaudītu iespējamos problēmgadījumus, to risinājumus un to ietekmi uz veiktspēju.
- Izpētīt dažādu vienas lapas tīmekļa lietotņu izstrādes ietvaru datu piesaistes tehnikas un veidot testa lietotnes veiktspējas mērīšanai, lai noskaidrotu ar datu piesaisti saistītus veiktspēju ietekmējošus faktorus.

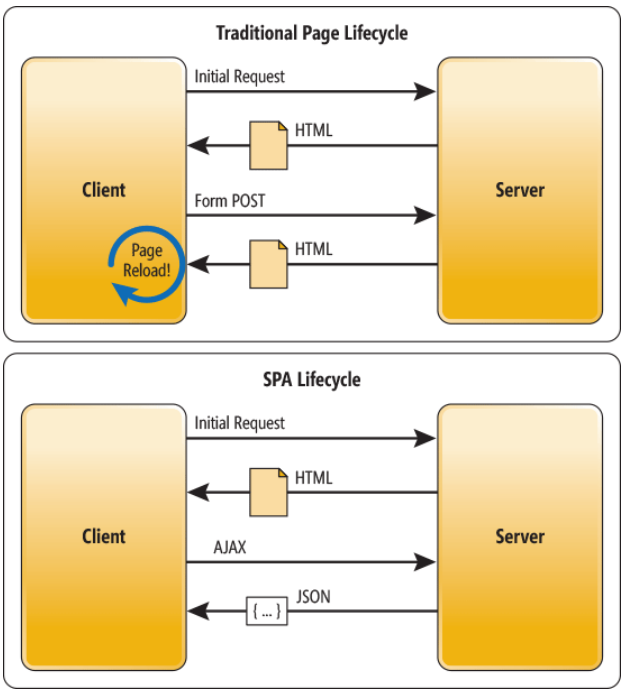
Darbā uzsvars ir uz klienta pusē strādājošo kodu sakarā ar to, ka vienas lapas tīmekļa lietotņu atšķirība no vairāku lapu tīmekļa lietotnēm ir tas, ka vairāk darba tiek veikts klienta pusē un mazāk darba tiek veikts servera pusē.

1. VIENAS LAPAS TĪMEKĻA LIETOTNES

Darbā apzīmējums “vienas lapas tīmekļa lietotne” ir sinonīms terminam “vienas lapas lietotne”. Vienas lapas tīmekļa lietotnes ir tīmekļa lietotnes, kas tā vietā, lai katru lapu ielādētu no servera un veiktu attēlotās lapas pārlādi, dažādu lapu attēlošanai dinamiski veic izmaiņas jau attēlotajā lapā. Vienas lapas tīmekļa lietotņu īpatsvars turpina pieaugt, jo klientu ierīces kļūst arvien jaudīgākas, un vienas lapas tīmekļa lietotnes rada reaģējošākas lietotnes iespaidu, jo, salīdzinot ar vairāku lapu lietotnēm, ir mazāk jāsazinās ar serveri. Nodaļā tiek apskatīts, kā vienas lapas tīmekļa lietotnes atšķiras no vairāku lapu tīmekļa lietotnēm un kā darbojas tīmekļa lietotņu klienta puses kods. Tiek apskatīti ietvari, kas palīdz vienas lapas tīmekļa lietotņu izveidē.

1.1 Atšķirība no vairāku lapu lietotnēm

Vienas lapas tīmekļa lietotnes tiek veidotas lietošanai tīmekļa pārlūkprogrammās un darbojas vienā lapā, veicot dinamisku tās satura maiņu tā vietā, lai, katru jaunu lapu pieprasot, ielādētu to no servera un veiktu pilnu lapas pārlādi ielādētā satura attēlošanai. Atšķirībā no vairāku lapu tīmekļa vietnēm vienas lapas tīmekļa lietotnes lielāko daļu nepieciešamā HTML, CSS un JavaScript koda saņem uzreiz pirmajā lapas ielādes laikā, un kods, kas nepieciešams papildus, vai dati, ko nepieciešams attēlot lapā, tiek dinamiski ielādēti, izmantojot AJAX, tādējādi veicot ielādi bez lapas pārlādes (skat. 1.1. att.). Iemesls tam, lai tīmekļa lietotni veidotu kā vienas lapas tīmekļa lietotni ir tāds, ka vienas lapas tīmekļa lietotnē nav nepieciešams gaidīt, kamēr notiek lapas pārlāde, un tas rada līdzīgu lietotāja pieredzi kā lietotnēs, kas darbojas lokāli datorā, planšetdatorā vai viedtālrunī [32, 33].



1.1. att. **Vienas lapas tīmekļa lietotņu salīdzinājums ar vairāku lapu tīmekļa lietotnēm** *[1]*

Vienas lapas tīmekļa lietotnēs jāņem vērā tas, ka lielāko daļu darba veic tīmekļa pārlūkprogrammas, kas nozīmē to, ka ne tikai ielādes laiks ir svarīgs, bet arī laiks, kas tiek patērēts HTML, JavaScript un CSS koda izpildei, dinamiski attēlojot datus, skatus u.c. Ņemot vērā to, ka ierīcēm, kādās klienti lieto vienas lapas tīmekļa lietotnes, var būt ierobežoti resursi, klienta puses koda veiktspējas optimizācijai ir liela nozīme [32, 33].

1.2 HTML5 valoda

HTML5 ir hiperteksta iezīmēšanas valodas HTML jaunākā versija. HTML ir kods, kurā tiek rakstītas tīmekļa lapas. HTML ļauj definēt tādus elementus kā paragrāfi, bloki, attēli, u.c. Kopā šie elementi veido koka struktūru (DOM), kas apraksta to, kas tiek attēlots tīmekļa lapā.

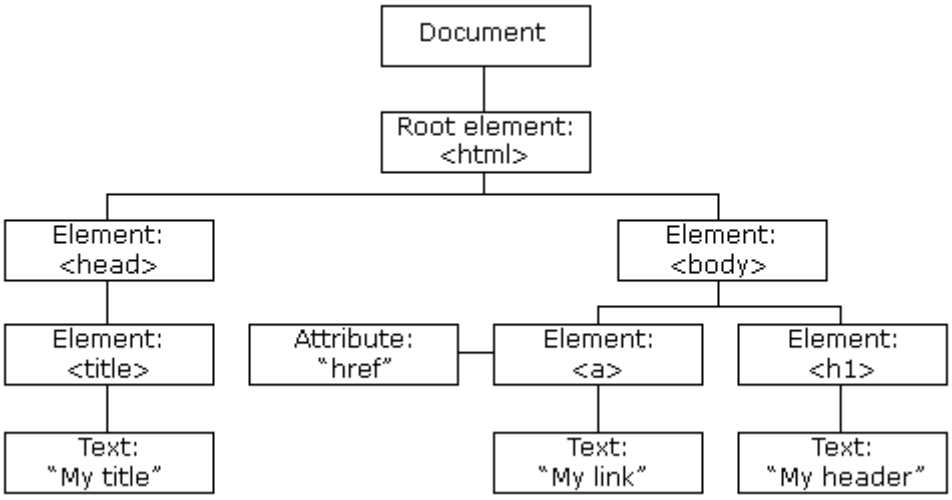
```
<select>
  <option value="volvo">Volvo</option>
  <option value="saab">Saab</option>
  <option value="mercedes">Mercedes</option>
  <option value="audi">Audi</option>
</select>
```

1.2. att. **HTML koda fragmenta piemērs** *[34]*

HTML elementi tiek pierakstīti ar atverošo un aizverošo tagu, un atverošajā tagā tiek definēti HTML elementa atribūti un CSS klases (skat. 1.2. att.). Starp atverošo un aizverošo tagu var būt elementi, kas atrodas iekšā definētajā elementā [28].

1.3 Dokumentu Objektu Modelis

DOM(Dokumentu Objektu Modelis) ir objektorientēta reprezentācija HTML un XML dokumentiem. Kad tīmekļa lapa tiek ielādēta, interneta pārlūks izveido lietotāja saskarnes DOM. HTML DOM ir standarta objektu modelis HTML valodai, un tas definē HTML elementus kā objektus, to atribūtus, tiem piešķirtās CSS klases un notikumus (*events*), kā arī metodes, kas nodrošina piekļuvi HTML elementiem [35, 36]. HTML DOM tiek veidots kā koka struktūra (skat. 1.3. att.).



1.3. att. **Dokumentu objektu modelis** *[36]*

DOM ir tīmekļa lapas reprezentācija, ko var mainīt, izmantojot JavaScript programmēšanas valodu. Izmantojot DOM, JavaScript kods var piekļūt DOM objektiem un veikt visas darbības, kas nepieciešamas dinamiska lapas satura veidošanai.

```
var paragraphs = document.getElementsByTagName('P');
```

1.4. att. **JavaScript koda fragmenta piemērs**

Piemēram, metode *getElementsByTagName()* redzamajā koda fragmentā (skat. 1.4. att.) atgriež visus paragrāfa elementus, kas atrodami tīmekļa lapas DOM, kurus pēc tam iespējams mainīt [16].

1.4 Programmēšanas valoda JavaScript

JavaScript ir programmēšanas valoda, kas tiek izmantota tīmekļa vietņu un lietotņu izstrādē. JavaScript kods izpildās klienta pārlūkprogrammā, un tas ļauj veidot dinamisku tīmekļa lapas saturu [36]:

- Mainīt lapas HTML elementus
- Mainīt lapas HTML atribūtus
- Mainīt lapas CSS stilus
- Noņemt lapas HTML elementus un atribūtus
- Pievienot HTML elementus un atribūtus lapai
- Reaģēt uz notikumiem(events)
- Izveidot HTML notikumus(*events*)

Sintaktiski JavaScript līdzinās tādām programmēšanas valodām kā *C, C++* un *Java – if, switch* zarošanās, *for, while* cikli un tādi operatori kā *&&*(un) un ||(vai) tiek rakstīti tā pat. JavaScript sintakses galvenā atšķirība ir tas, ka mainīgajiem nav noteikta tipa, kas nozīmē, ka koda izpildes laikā *string* tipa mainīgajam var tikt piešķirta, piemēram, *integer* vērtība [15, 26].

1.5 Tīmekļa pārlūkprogrammas

Vispasaules tīmeklis (*World Wide Web*) ir koplietojamu resursu sistēma internetā. Tīmekļa pārlūkprogrammas saņem tīmekļa lapu saturu no tīmekļa serveriem, izmantojot HTTP (*HyperText Transfer Protocol*) protokolu. Tīmekļa lapas tiek rakstītas hiperteksta iezīmēšanas valodā HTML. Lapas var būt papildinātas ar CSS (*Cascading Style Sheets*), kas definē HTML elementu vizuālo noformējumu, un JavaScript, kas ļauj veidot dinamisku saturu tīmekļa lapās. Liela daļa tīmekļa lapu sastāv no servera puses koda, kas tiek izpildīts serverī, un klienta puses koda, ko veido JavaScript, CSS un HTML, kas tiek izpildīts interneta pārlūkprogrammā. Galvenās tīmekļa pārlūkprogrammas komponentes ir:

- Lietotāja saskarne – adreses josla, pogas “uz priekšu” un “atpakaļ”, grāmatzīmju pārvaldība, izstrādātāja rīki un citas pārlūkprogrammas daļas, kas nav logā, kas attēlo tīmekļa lapas saturu.
- Pārlūkprogrammas dzinējs, kas reaģē uz darbībām lietotāja saskarnē, piemēram, izsauc iepriekšējās attēlotās lapas renderēšanu, kad lietotājs nospiež pogu “atpakaļ”.

- Renderēšanas dzinējs, kas veic pieprasītās tīmekļa lapas renderēšanu un attēlošanu tīmekļa pārlūkprogrammā. Tas interpretē HTML dokumentus, kas formatēti, izmantojot CSS, un ģenerē elementus konkrētā izkārtojumā. Dažādas pārlūkprogrammas izmanto dažādus renderēšanas dzinējus.
- JavaScript interpretētājs, kas veic JavaScript koda parsēšanu un izpildi. Parsēšanas rezultāti tiek nosūtīti renderēšanas dzinējam, lai tos attēlotu.
- Datu glabātava – pārlūkprogramma klienta datora lokālajā atmiņā instalācijas vietā var glabāt tādu informāciju kā kešatmiņu, sīkdatnes, grāmatzīmes u.c.[2, 53]

2018. gadā visplašāk izmantotās tīmekļa pārlūkprogrammas ir *Google Chrome*, Internet *Explorer, Firefox, Edge* un Safari [37].

1.6 JavaScript ietvari vienas lapas tīmekļa lietotņu izstrādei

Veidojot vienas lapas tīmekļa lietotnes, lielais daudzums JavaScript koda var kļūt grūti uzturams, kā arī veidošana no pamatiem, izstrādē neizmantojot jau esošus risinājumus, var būt neoptimāli laikietilpīga. JavaScript ietvari palīdz organizēt kodu, sadalot to skatos, kontrolieros, un veidnēs, kā arī tajos eksistē jau gatavi risinājumi tādām lietām kā datu piesaiste un maršrutēšana. Nodaļā tiek īsi aprakstīti daži no JavaScript ietvariem vienas lapas tīmekļa lietotņu izstrādei. Tika izvēlēti ietvari React JS, Angular JS un Knockout JS, jo tajos tiek izmantotas bakalaura darbā aprakstītās un pētītās datu piesaistes tehnikas.

1.6.1 React JS

React JS ir JavaScript valodā veidots ietvars, kas paredzēts vienas lapas tīmekļa lietotņu izstrādes paātrināšanai un uzturamāka koda veidošanai. React JS lietotnēs kods tiek veidots, izmantojot deklaratīvo programmēšanas paradigmu un uz komponentēm balstītu arhitektūru. Ietvara kontekstā tas nozīmē to, ka kods sastāv no komponentēm, kurām ir definēta renderēšanas metode, un tas, kā komponente tiks renderēta, ir atkarīgs no komponentei piesaistītajiem datiem (*state, props*) [3].

```
class ShoppingList extends React.Component {
  render() {
    return (
      <div className="shopping-list">
        <h1>Shopping List for {this.props.name}</h1>
        <ul>
          <li>Instagram</li>
          <li>WhatsApp</li>
          <li>Oculus</li>
        </ul>
      </div>
    );
  }
}
```

1.5. att. **React JS koda fragmenta piemērs** *[52]*

Redzamajā JavaScript kodā (skat. 1.5. att.) ir arī HTML5 kods – tā ir JSX sintakse, ko React JS piedāvā, lai kods būtu lasāmāks. JSX sintaksei ir nelielas atšķirības no HTML5 koda, piemēram, CSS klases definēšanai netiek izmantots *class* atribūts, jo *class* ir atslēgas vārds, kas rezervēts valodai JavaScript – JSX tā vietā lieto *className* [39]. Šīs sintakses izmantošana nav obligāta – šos HTML elementus renderēšanas metodē iespējams rakstīt arī kā JavaScript objektus. Tā kā React JS ir paredzēts vienas lapas tīmekļa lietotņu izstrādei, tas piedāvā iespēju uzstādīt maršrutēšanu kā veidu, lai pārslēgtos no viena skata uz citu [3].

1.6.2 Angular JS

AngularJS ir JavaScript valodā veidots MVC (Model-View-Controller) ietvars, kas paredzēts vienas lapas tīmekļa lietotņu izstrādes paātrināšanai un uzturamāka koda veidošanai. AngularJS ietvarā skats tiek rakstīts HTML iezīmēšanas valodā, pielietojot direktīvas un datu piesaisti no modeļa. Kontrolieris ir daļa, kurā tiek veikta biznesa loģika, kas ir atdalīta no skata ar nolūku padarīt veidojamās lietotnes kodu strukturētāku un pārskatāmāku. Kā modelis AngularJS ietvarā ir *$scope* objekts, kas glabā sevī datus, kurus skats sev var piesaistīt. AngularJS ir ietvars, kas ietver lielu daudzumu funkcionalitātes, piemēram, maršrutēšanu, datu piesaisti, kā arī iebūvētas konstrukcijas (direktīvas) [4].

```
<div ng-repeat="post in posts">
    <p>{{post.name}}</p>
</div>
```

1.6. att. **Angular JS ng-repeat direktīva**

Redzamajā kodā (skat. 1.6. att.) parādīta *ng-repeat* direktīvas izmantošanas vienkāršs piemērs. Direktīva *ng-repeat* ir paredzēta, lai DOM elementus katram kāda konkrēta masīva elementam. Piemērā masīva *posts* elementi tiek ģenerēti kā paragrāfa DOM elementi ar tekstu, kas ir masīva elementa *name* atribūtā. Masīva elementu *name* atribūts tiek iegūts, izmantojot datu piesaisti ar figūriekavu sintaksi.

1.6.3 Knockout JS

KnockoutJS ir JavaScript programmēšanas valodā veidots ietvars, kas balstīts uz MVVM (Model-View-ViewModel) arhitektūras šablona un kas paredzēts vienas lapas tīmekļa lietotņu izstrādes paātrināšanai un uzturamāka un strukturētāka koda veidošanai. Ietvarā skats ir HTML hiperteksta iezīmēšanas valodā rakstīts kods, kam piesaistīti dati no skatmodeļa (*ViewModel*), izmantojot HTML elementu *data* atribūtu. Skatmodelī tiek glabāti modeļa dati, un, lai veiktu izmaiņas skatā, ir paredzēts nevis tieši (piemēram, izmantojot getElementById) piekļūt DOM elementiem, bet gan veikt izmaiņas skatmodelī, kas automātiski attēlosies skatā [5].

```
<div data-bind="foreach: posts">
    <p data-bind="text: name"></p>
</div>
```

1.7. att. **Knockout JS masīva piesaiste**

Attēlā (skat. 1.7. att.) redzams vienkāršs piemērs masīva datu piesaistei. Atslēgas vārds *foreach* definē to, ka katram masīva *posts* elementam tiks izveidots definētais paragrāfa DOM elements, kura teksts būs *name* atribūts no atbilstošā *posts* masīva elementa. Knockout JS ir mazāka apjoma izstrādes ietvars, salīdzinot ar Angular JS un React JS, un neiekļauj, piemēram, maršrutēšanu [3].

2. VIENAS LAPAS TĪMEKĻA LIETOTŅU VEIKTSPĒJA

Bakalaura darbā veiktspēja tiek pētīta tieši klienta puses kodam, nevis ielādes ātrdarbībai, jo vienas lapas tīmekļa lietotnēs darbības ar lietotni izraisa koda darbošanos klienta datorā, lai sniegtu lietotājam paredzēto funkcionalitāti, un katras lapas attēlošanai nav nepieciešams to ielādēt no servera. Nodaļā tiek apskatīts, kas ir optimāla veiktspēja klienta puses kodam, kā arī aprakstīts, kā veiktspēja tiek mērīta šī bakalaura darba ietvaros.

2.1 Optimāla veiktspēja

Sasniegt optimālu veiktspēju vienas lapas tīmekļa lietotnē nozīmē to, ka kods, kas tiek izpildīts lietotāja darbību rezultātā, tiek izpildīts laikā, ko lietotājs uztver kā momentānu. Sasniegt pietiekamu veiktspēju vienas lapas tīmekļa lietotnē nozīmē to, ka kods, kas tiek izpildīts lietotāja darbību rezultātā, tiek izpildīts laikā, kas netraucē lietotājam turpināt koncentrēties uz izpildāmo uzdevumu.

Lietotāju uztvere laikam, kas vajadzīgs lietotāja darbību rezultātā izpildāmā koda izpildei, ir sadalāma šādos intervālos [54, 65, 64]:

- 0.1 sekundes: kodu, kas lietotāja darbības rezultātā tiek izpildīts šādā laikā, lietotājs uztver kā momentānu. Aptuveni šāds laiks cilvēkam nepieciešams, lai reaģētu uz kādu ārējo stimulu,
- 0.5 – 1 sekunde: Šādā laika intervālā aizture ir manāma, taču neizraisa gaidīšanas sajūtu un lietotāja domu plūsma paliek nepārtraukta,
- 2 – 5 sekundes: Šādā laika intervālā aizture ir ļoti manāma, taču lietotājs vēl ir spējīgs koncentrēties uz izpildāmo uzdevumu,
- 5 – 10 sekundes: Šādā laika intervālā lietotājs vēl ir spējīgs koncentrēties uz izpildāmo uzdevumu, taču palielinās iespēja, ka lietotāja uzmanība tiks novērsta,
- 10 un vairāk sekundes: Cilvēka uzmanības laika intervāls tiek pārsniegts un pastāv liela varbūtība, ka cilvēka uzmanība tiks novērsta.

No augstāk minētajiem pētījuma rezultātiem izriet, ka 2 - 5 sekundes ir laika intervāls, kādā jāsniedz atbilde uz lietotāja darbību, lai tiktu sasniegta pietiekama veiktspēja, savukārt 0.1 sekunde ir laika intervāls, kādā jāsniedz atbilde, lai tiktu sasniegta optimāla veiktspēja. Ja

lietotnē pastāv laikietilpīgi datu pieprasījumi no servera vai īpaši apjomīgu sarakstu vai tabulu renderēšana, pilnībā optimālu veiktspēju sasniegt ir grūti, taču lietotāja pieredze būs labāka, ja visu veicamo darbību, kurās tas iespējams, atbildes laiks būs līdz 0.1 sekundei.

Lai tiktu sasniegta optimāla veiktspēja, ir svarīgi arī tas, lai animācijas un satura ritināšana notiktu bez aizturēm. Tā kā mūsdienās lielākā daļa ierīču ekrānos attēlo 60 kadrus sekundē, ir svarīgi nodrošināt to, ka animācijas un satura ritināšana tiek veikta, attēlojot 60 kadrus sekundē, kas nozīmē, ka katra kadra attēlošana jāveic aptuveni 16 milisekundēs [14, 54].

Tā kā tīmekļa lietotnēs koda izpilde parasti nobloķē lietotāja saskarni, neļaujot lietotājam darboties arī ar citām lietotnes daļām, ir vēl svarīgāk sasniegt optimālu veiktspēju vai darbībām, kas ir apjomīgas un aizņem daudz laika, panākt to, ka lietotāja saskarne netiek nobloķēta. Problēmas, kas visvairāk traucē lietotāja darbību ar lietotni, ir gadījumi, kad interneta pārlūkprogramma koda izpildi veic pārāk ilgi vai tiek izmantots par daudz atmiņas un interneta pārlūkprogrammas darbība tiek pārtraukta. Pārlūkprogrammas darbības pārtraukšanas rezultātā lietotājam jāaizver tīmekļa pārlūkprogramma un no jauna jāatver lietotne interneta pārlūkprogrammā, lai atkal varētu veikt darbības tajā.

2.2 Veiktspējas mērīšana

Apakšnodaļā tiek aprakstīts, kā tiek veikti veiktspējas mērījumi šī bakalaura darba ietvaros, lai sniegtu lasītājam ieskatu tajā, kā rezultāti iegūti.

2.2.1 Ierīce un pārlūkprogramma

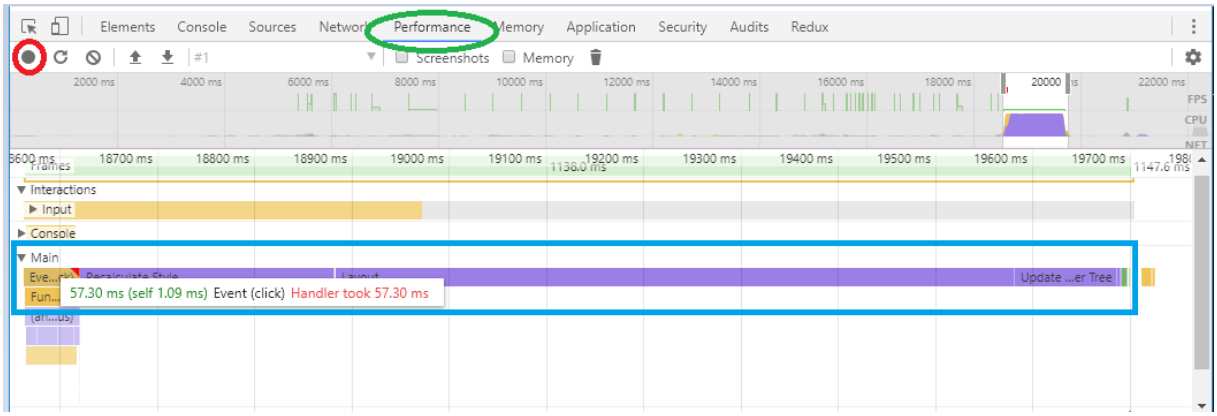
Tā kā dažādās ierīcēs veiktspējas rezultāti var atšķirties savā starpā, ir svarīgi precizēt to, kādā ierīcē tiek veikti šī bakalaura darba ietvaros veiktie veiktspējas mērījumi. Visi veiktspējas mērījumi tiek veikti datorā ar sekojošiem parametriem:

- *Intel Core i5 – 3337U* procesors: 2 kodoli, 1.80 GHz pamata frekvence, 2.70 GHz paātrinājuma frekvence.
- 6 GB operatīvās atmiņas.
- *NVIDIA GeForce GT 740M* video karte

Ņemot vērā, ka vairāk nekā puse tīmekļa lietotāju izmanto *Google Chrome* tīmekļa pārlūkprogrammu, veiktspējas mērījumi tiek veikti *Google Chrome* interneta pārlūkprogrammā. Visiem veiktspējas testiem tiek izmantota versija, kas ir aktuāla pirmo šim darbam paredzēto veiktspējas mērījumu veikšanas laikā – *Google Chrome 65.0.3325.181*.

2.2.2 Renderēšanas veiktspējas mērīšana

Renderēšanas veiktspējas mērījumi tiek veikti, izmantojot Google Chrome tīmekļa pārlūkprogrammas *Performance* sadaļu. Lai veiktu mērījumu konkrētā scenārijā, tiek nospiesta veiktspējas ierakstīšanas poga izstrādātāja rīkos un pēc tam nospiesta poga tīmekļa lietotnes lietotāja saskarnē, kas izraisa renderēšanas darbības konkrētajā testa scenārijā. Kad renderēšanas darbības ir pabeigtas, tiek nospiesta ierakstīšanas pārtraukšanas poga, un izstrādātāja rīkos ir redzams, cik laika katrs renderēšanas posms ir aizņēmis.

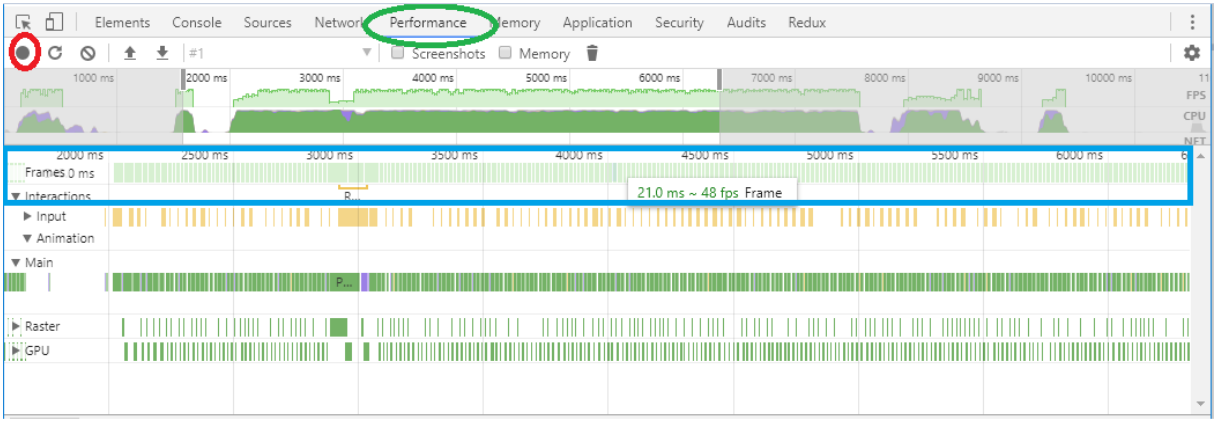


2.1. att. **Renderēšanas veiktspējas mērījums**

Attēlā (skat. 2.1. att.) redzams testa scenārijā veikts veiktspējas mērījums. Poga, lai ierakstītu/pārtrauktu ierakstīt veiktspēju, ir apvilkta sarkanā krāsā, poga, kas jānospiež, lai nokļūtu izstrādātāja rīka veiktspējas sadaļā, apvilkta zaļā krāsā, savukārt taisnstūri, kas parāda katrā renderēšanas posmā patērēto laiku ievilkti zilā krāsā. Uzliekot peli uz kāda no taisnstūriem, iespējams redzēt to, cik laika katrs no šiem renderēšanas posmiem ir patērējis. Tiek saskaitīts visu posmu patērētais laiks, sākot no funkcijas, kas izpildās lietotāja saskarnē esošas pogas nospiešanas rezultātā, un beidzot ar laiku, kas patērēts kompozitēšanas posmam. Renderēšanas procesa posmi aprakstīti 4.1 apakšnodaļā.

2.2.3 Kadru skaita sekundē mērīšana

Kadru skaita sekundē mērījumi tiek veikti, izmantojot *Google Chrome* tīmekļa pārlūkprogrammas *Performance* sadaļu. Lai veiktu kadru skaita sekundē mērījumu konkrētā scenārijā, tiek nospiesta veiktspējas ierakstīšanas poga izstrādātāja rīkos, un pēc tam tiek veikta animācija vai lapas satura ritināšana. Pēc animācijas veikšanas vai lapas satura ritināšanas tiek nospiesta poga, kas pārtrauc veiktspējas ierakstīšanu.



2.2. att. **Kadru attēlošanas veiktspējas mērījums**

Pēc veiktspējas ierakstīšanas pārtraukšanas izstrādātāja rīka *Performance* sadaļas *Frames* sadaļā redzamie zaļie taisnstūri parāda kadrus, un, turot kursoru uz kāda no šiem kadriem, redzams laiks, kas patērēts konkrētā kadra attēlošanai, kā arī tas, cik kadri sekundē tiktu attēloti, ja katra kadra attēlošanai tiktu patērēts šāds laiks (skat. 2.2. att.) [66]. Tā kā aprēķināt vidējo kadru skaitu sekundē visiem redzamajiem kadriem manuāli ir laikietilpīgs process, tam tiek izmantota cita pieeja – izstrādātāja rīki tiek atvērti nevis blakus attēlotajai lapai, bet gan atsevišķā logā. Tas, ka izstrādātāja rīki ir atvērti citā logā, dod iespēju atvērt izstrādātāja rīkus šim izstrādātāja rīku logam, nospiežot taustiņu kombināciju *Ctrl* + *Shift* + *i*. Kad tas izdarīts, izstrādātāja rīkiem atvērtajos izstrādātāja rīkos iespējams *Console* sadaļā ierakstīt attēlā (skat. 2.3. att.) redzamo formulu, kas aprēķina lietotnei atvērtajos izstrādātāja rīkos attēloto kadru vidējo kadru skaitu sekundē [24].

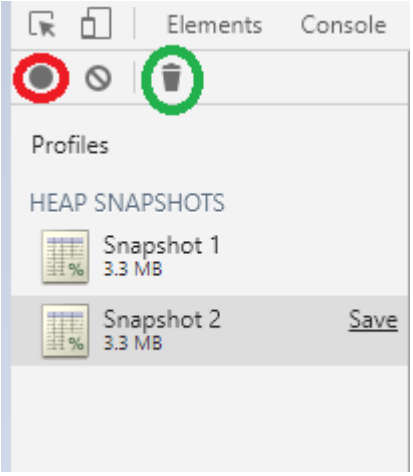
```
+UI.panels.timeline._flameChart._model._frameModel._frames.slice(1).map(f => 1000 /
f.duration).reduce((avg, fps, i) => (avg*i + fps) / (i+1), 0).toFixed(1)
```

2.3. att. **Formula vidējā kadru skaita sekundē aprēķināšanai**

Šāda pieeja ļauj neveikt manuālu katra kadra vidējā kadru skaita sekundē saskaitīšanu un dalīšanu ar kadru skaitu.

2.2.4 Aizņemtās atmiņas mērīšana

Aizņemtās atmiņas mērīšana tiek veikta, izmantojot Google Chrome izstrādātāja rīka *Memory* sadaļu [67]. Testa lietotnēs tika veiktas darbības, kas veic atmiņas aizņemšanu/atbrīvošanu, un pēc tam tika veikts atmiņas mērījums.



2.4. att. **Atmiņas mērīšana**

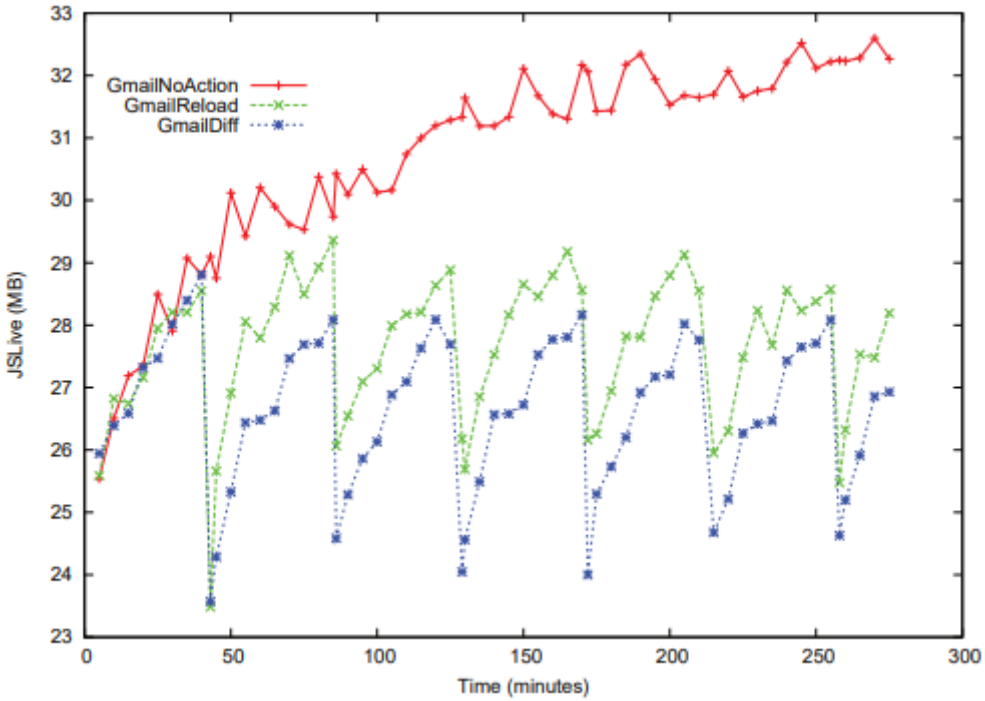
Attēlā (skat. 2.4. att.) atmiņas mērījuma veikšanas poga apvilkta ar sarkanu krāsu, savukārt piespiedu atkritumu savākšanas poga apvilkta ar zaļu krāsu. Piespiedu atkritumu savākšanas pogas darbība saistīta ar to, ka interneta pārlūkprogramma veic atkritumu savākšanu brīdī, kad tas ir izdevīgi, līdz ar to atmiņas mērījums, neveicot piespiedu atkritumu savākšanu var būt neprecīzs.

2.2.5 JavaScript koda izpildes laika mērīšana

JavaScript koda izpildes laiks šī bakalaura darba ietvaros tika mērīts, izmantojot *console.time()* un *console.timeEnd()* JavaScript funkcijas. Lai veiktu veiktspējas mērījumus, izmantojot šīs JavaScript funkcijas, izpildāmā JavaScript koda sākumā jāizsauc funkcija *console.time()*, iekavās padod konkrētu nosaukumu taimerim, piemēram, *console.time("parsing")*, savukārt izpildāmā JavaScript koda beigās jāizsauc funkcija *console.timeEnd()*, tai padodot konkrēta taimera nosaukumu, piemēram, *console.timeEnd("parsing")*. Kad kādam taimerim tiek izsaukta funkcija *console.timeEnd()*, pārlūkprogrammas izstrādātāja rīku sadaļā *Console* tiek parādīts laiks, kas konkrētajam taimerim patērēts.

3. ATMIŅAS NOPLŪŽU NOVĒRŠANA

Veidojot jebkāda veida programmatūru, ir svarīgi kontrolēt izmantotās atmiņas daudzumu un izvairīties no atmiņas noplūdēm. Vienas lapas tīmekļa lietotnēm tas ir svarīgi tāpēc, ka arvien vairāk tiek izmantotas tādas ierīces kā planšetdatori un viedtālruņi, un šajās ierīcēs operatīvās atmiņas apjoms var būt nozīmīgi mazāks, nekā datoros. Vēl viens iemesls tam, kāpēc atmiņas kontrolēšana ir svarīga vienas lapas tīmekļa lietotnēm ir tāds, ka atšķirībā no vairāku lapu tīmekļa lietotnēm, kas, piemēram, pārslēdzoties uz citu skatu, veic lapas pārlādi, tādējādi atbrīvojot atmiņu, vienas lapas tīmekļa lietotnes savā dzīves ciklā netiek pārlādētas, un lietotāji tās var izmantot ilgu laiku, veicot lielu darbību daudzumu. Ņemot vērā, ka JavaScript programmēšanas valodā strādā automātiskā atkritumu savākšana (*garbage collection*), ir viegli nepievērst uzmanību atmiņas pārvaldīšanai, taču arī programmēšanas valodās ar automātisko atkritumu savākšanu eksistē konkrētas situācijas, kurās neizmantota atmiņa netiek atbrīvota [13]. 2015. gadā veikts pētījums, kas saistīts ar atmiņas problēmām un noplūdēm, pārbaudīja atmiņas izmantošanu vairākām pazīstamām tīmekļa lietotnēm, piemēram, *Facebook* un *Gmail* [42].



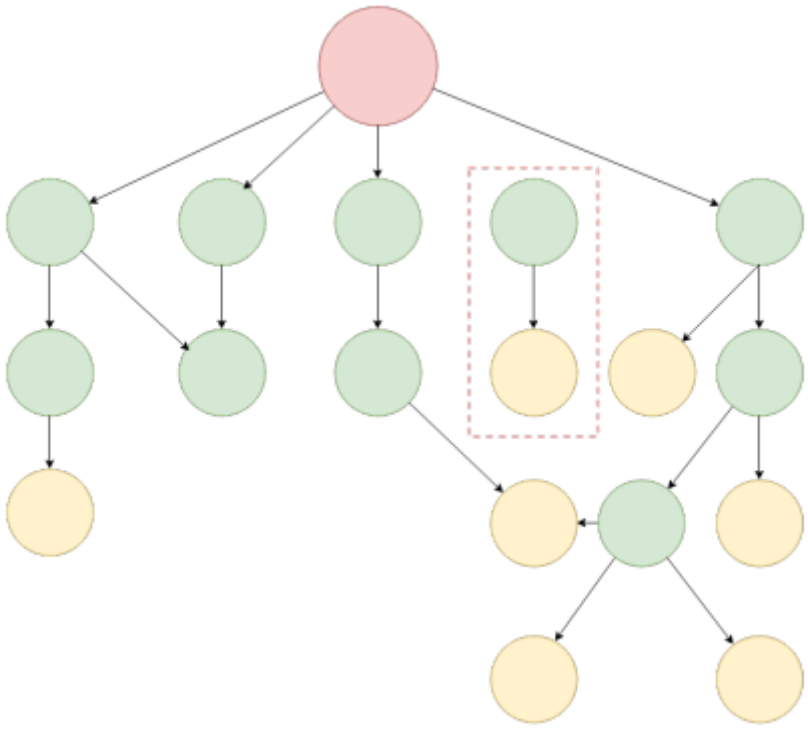
3.1. att. **Gmail lietotnes atmiņas mērījumi** *[42]*

Attēlā (skat. 3.1. att.) redzami atmiņas mērījumi laika gaitā dažādos scenārijos *Gmail* lietotnei. *GmailNoAction* scenārijā lietotājs veic darbības lietotnē, neradot papildus datus (nesūtot/nesaņemot jaunus e-pastus), *GmailReload* scenārijā lietotājs arī veic darbības, taču

periodiski pārlādē lapu. *GmailDiff* scenārijā lietotājs veic darbības, taču periodiski aizver un atver pārlūkprogrammas cilni ar Gmail lietotni. Redzams, ka scenārijā, kad netiek veiktas lapas pārlādes vai cilnes aizvēršanas, novērojams atmiņas pieaugums, lietotājam jaunus datus neradot, kas liecina par to, ka lietotnē eksistē atmiņas noplūdes. Fakts, ka lielu un populāru kompāniju plaši izmantotās vienas lapas tīmekļa lietotnēs ir konstatētas atmiņas noplūdes, norāda uz to, ka vienas lapas tīmekļa lietotnēs, kas ir maz testētas un kuru izstrādē piedalās programmētāji ar zemu pieredzi, atmiņas noplūdes var būt lielākas. Nodaļā apskatītie atmiņas noplūžu scenāriji neparāda pilnībā visas iespējamās atmiņas noplūdes, eksistē dažādi specifiski gadījumi, kad vienas lapas tīmekļa lietotnē kāds objekts netiek atbrīvots no atmiņas, jo atkritumu savācējs to uzskata par sasniedzamu, neskatoties uz to, ka programmatūras kodā tas vairs netiek izmantots.

3.1 Atkritumu savākšana

Programmatūras kodam darbojoties tīmekļa pārlūkprogrammā, visiem izveidotajiem objektiem un datu struktūrām tiek piešķirta atmiņa, lai ar tiem veiktu lasīšanas un rediģēšanas operācijas (read and write operations). Tādās programmēšanas valodās kā *C* vai *C++* atmiņas pārvaldību programmētājs veic manuāli, taču tādām valodām kā *Java*, *C#* vai *JavaScript* atmiņas pārvaldību automātiski veic atkritumu savācējs. Atkritumu savācējs ir mehānisms, kura uzdevums ir atrast piešķirto atmiņu, kas vairs nav vajadzīga, un to atbrīvot [43].



3.2. att. **JavaScript atmiņas grafs** *[43]*

JavaScript atmiņa ir kā grafs, kur sakne ir *Window* objekts (attēlā 3.2. att. atzīmēts rozā krāsā), un no saknes objekta eksistē reference uz citiem izmantotajiem objektiem, kas ir neprimitīvie datu tipi (attēlā 3.2. att. sarkanā krāsā), kam savukārt eksistē reference uz citiem objektiem vai lapām. JavaScript programmēšanas valodā funkcijas, regulārās izteiksmes, masīvi un objekti ir neprimitīvie datu tipi. Lapas grafā ir primitīvie datu tipi (attēlā dzeltenā krāsā), kam nav references uz citiem objektiem. Atkritumu savācēji var atbrīvot attēlā ar raustītu līniju apvilkto objektu un tā lapu, jo neeksistē neviena reference uz to un tas nav sasniedzams [13].

3.1.1 References atkritumu savācējs

References atkritumu savācēji nosaka to, vai objekts ir nevajadzīgs, pārbaudot, vai eksistē kāds cits objekts ar referenci uz to. Objekta aizņemtā atmiņa tiek uzskatīta par atbrīvojamu, ja neeksistē neviens cits objekts, kam uz to ir reference. Šādus atkritumu savācējus izmanto tādas pārlūkprogrammas kā *Internet Explorer 6* un *Internet Explorer 7* [44].

3.1.2 Mark and Sweep atkritumu savācējs

Atzīmēšanas un noslaucīšanas atkritumu savācējs nosaka to, vai objekts ir nevajadzīgs, pārbaudot to, vai objekts no kādas vietas kodā ir sasniedzams. Šis algoritms periodiski pārbauda, kuri objekti ir sasniedzami, sākot ar to, ka tiek noteikts, kuri objekti tiek referencēti no saknes objekta, pēc tam tiek noteikts, kuri objekti tiek referencēti no šiem objektiem, un tā tas turpinās, līdz tiek sasniegtas lapas. Ja objekts šī procesa laikā nav iezīmēts kā sasniedzams no kāda cita objekta, tas atkritumu savākšanas laikā tiek noņemts un tā aizņemtā atmiņa tiek atbrīvota [44].

3.2 Atmiņas noplūžu piemēri

Apakšnodaļā ar vienkāršiem autora izveidotiem piemēriem tiek parādīts, kā programmatūras izstrādes laikā iespējams nepareizi pārvaldīt atmiņu, neļaujot atkritumu savācējam atbrīvot atmiņu, kas vairs netiek izmantota.

3.2.1 DOM elementu noplūde

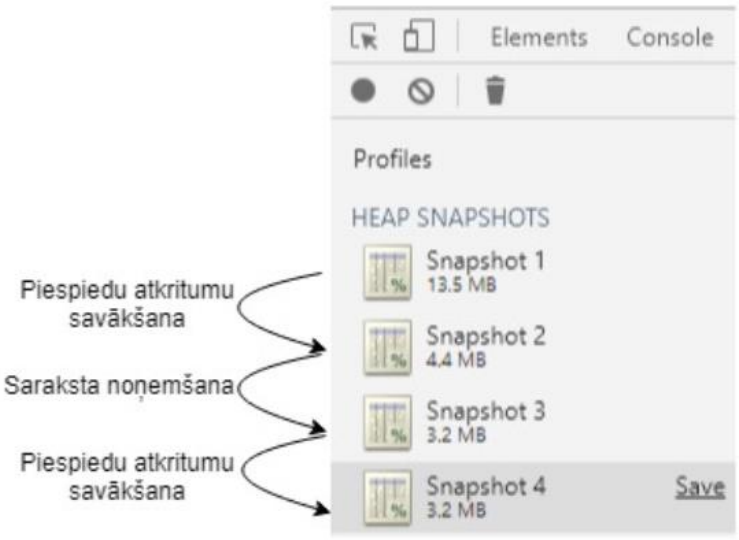
Viens no atmiņas noplūdes veidiem ir DOM elementu atmiņas noplūde (skat. 2. pielikums). Šāda atmiņas noplūde rodas, kad kādam mainīgajam tiek piešķirta kāda DOM elementa vērtība, kas nozīmē, ka eksistē reference uz konkrēto DOM elementu [40]. Tas, ka eksistē reference uz kādu DOM elementu no mainīgā, ko atkritumu savācējs nav izdzēsis, nozīmē to, ka tā aizņemto atmiņu joprojām nevar atbrīvot, jo eksistē reference uz to no mainīgā, kas joprojām ir sasniedzams.

```
document.getElementById('leakbutton').addEventListener("click", function () {
    var list = document.getElementById("posts");//Ar šo variantu netiks izraisīta atmiņas noplūde.
    list.remove();
});
```

3.3. att. **Izvairīšanās no DOM elementu noplūdes**

Attēlā (skat. 3.3. att.) redzamajā koda fragmentā redzams, ka pogai tiek pievienota notikuma apstrādes funkcija, kurā tiek izvēlēts DOM elements, kas tiek noņemts no lapas DOM. Konkrētajā scenārijā mainīgais *list*, kam ir reference uz atbilstošo DOM elementu, tiek izveidots notikuma apstrādes funkcijas vērienā, kas nozīmē, ka, funkcijas darbam beidzoties,

šo mainīgo atkritumu savācējs noņems no atmiņas, līdz ar to no DOM noņemtais elements nepaliks atmiņā, jo mainīgais ar referenci uz to arī ir noņemts no atmiņas [40].



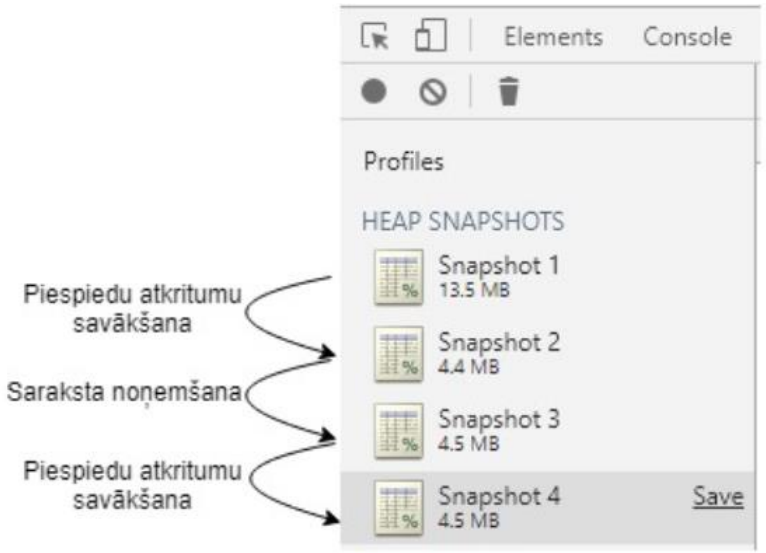
3.4. att. **Izvairīšanās no DOM elementu noplūdes atmiņas mērījumi**

Attēlā (skat. 3.4. att.) redzami pārlūkprogrammas *Google Chrome* izstrādātāja rīkā veiktie atmiņas mērījumi. Tā kā atkritumu savākšana tīmekļa lietotnes dzīves ciklā notiek periodiski, un nav iespējams pilnībā paredzēt, kad tā izpildīsies, eksperimenta nolūkos pēc lapas ielādes tiek veikta piespiedu atkritumu savākšana, nospiežot augšpusē redzamo atkritumu tvertnes pogu, lai atmiņas mērījumos nerastos kļūmes. Pēc piespiedu atkritumu savākšanas tiek veikta koda fragmentā minētās pogas nospiešana, kas veic atbilstošā DOM elementa noņemšanu no lietotāja saskarnes DOM. Attēlā (skat. 3.4. att.) redzams, ka pēc DOM elementa noņemšanas atmiņa ir atbrīvota, jo vairs neeksistē mainīgais ar referenci uz to – tas arī ir noņemts, notikuma apstrādes funkcijai beidzoties.

```
var list = document.getElementById("posts");//Ar šo variantu tiks izraisīta atmiņas noplūde.
document.getElementById('leakbutton').addEventListener("click", function () {
    list.remove();
});
```

3.5. att. **DOM elementu noplūde**

Attēlā (skat. 3.5. att.) redzamajā koda fragmentā redzams, ka pirms notikuma apstrādes funkcijas piesaistes atbilstošajai pogai tiek definēts mainīgais ar referenci uz konkrētu elementu lietotāja saskarnes DOM. Tas nozīmē, ka arī tad, kad šis DOM elements tiks noņemts no lietotāja saskarnes DOM, tas paliks atmiņā, kamēr mainīgo, kam eksistē reference uz šo DOM elementu, nebūs izdzēsis atkritumu savācējs. Šāds kods var rasties, kad programmētājs vēlas nodefinēt mainīgo ārpus citām vairākām funkcijām, kurās šis mainīgais tiks izmantots, lai nebūtu jāveic šī definēšana katrā funkcijā.



3.6. att. **DOM elementu noplūdes atmiņas mērījumi**

Attēlā (skat. 3.6. att.) redzami tādā pašā scenārijā veikti atmiņas mērījumi šim koda fragmentam. Šeit redzams, ka pēc DOM elementa noņemšanas no lietotāja saskarnes DOM, tas vēl joprojām tiek turēts atmiņā, jo uz to eksistē reference no sasniedzama mainīgā. Piespiedu atkritumu tīrīšana pēc saraksta noņemšanas parāda to, ka aizņemtās atmiņas daudzums nemainās, kas apstiprina to, ka šie rezultāti nav saistīti ar atkritumu savācēja periodisko darbību – nav saistīti ar to, ka atkritumu savācējs vēl nav veicis nesasniedzamo objektu atmiņas atbrīvošanu.

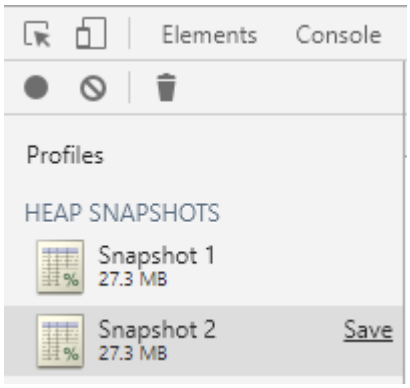
3.2.2 Referencējot daļu no objekta, atmiņā paliek viss objekts

Ir iespējams izmantot vairāk atmiņas, nekā nepieciešams, ja kādam mainīgajam kā vērtība tiek piešķirts kāda objekta apakšobjekts [40]. Piemērs šādai situācijai var būt gadījums, kad kādā funkcijā tiek apstrādāts kāds no servera saņemts objekts, un kādam DOM elementam tiek pievienota nospiešanas notikuma apstrādes funkcija, kurā tiek izmantota daļa no servera saņemtā objekta (skat. 3. pielikums).

```
var attachTheListener = function(){
    var a1=[];
    var a2=[];
    for(var i=0; i<300000; i++){
        a1.push({name:"test"});
    }
    for(var i=0; i<300000; i++){
        a2.push({name:"test"});
    }
var oObject={
    p1:a1,
    p2:a2
}
document.getElementById('leakbutton').addEventListener("click", function () {
    alert(oObject.p1[0].name);
});
}
attachTheListener();
```

3.7. att. **Nevajadzīgi liels objekts turēts atmiņā**

Attēlā (skat. 3.7. att.) redzamajā koda fragmentā objekts *oObject* ir kā piemērs ar vairākiem atribūtiem (piemērā p1 un p2), un tālāk DOM elementam ar identifikatoru *leakbutton* tiek pievienota nospiešanas notikuma apstrādes funkcija, kas izmanto *p1* atribūtu no objekta *oObject*. Neskatoties uz to, ka pēc *attachTheListener* funkcijas izpildes atkritumu savācējs iztīra atmiņu no visiem šīs funkcijas lokālajiem mainīgajiem, viss oObject objekts paliek atmiņā, jo tas tiek referencēts notikuma apstrādes funkcijai DOM elementam.



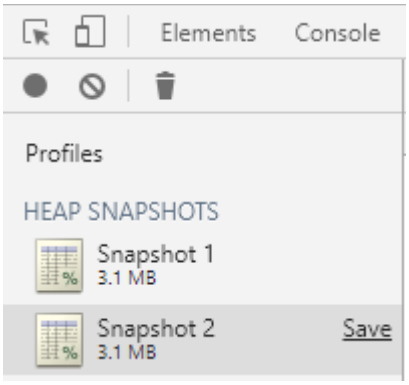
3.8. att. **Atmiņas mērījums nevajadzīgi liela objekta turēšanai atmiņā**

Veicot atmiņas mērījumus pēc šī koda izpildes (skat. 3.8. att.) redzams, ka tiek aizņemti 27,3 megabaiti atmiņas. Otrais redzamais mērījums veikts pēc piespiedu atkritumu savākšanas, un ir redzams, ka atkritumu savākšana nostrādājusi jau pirms pirmā atmiņas mērījuma, kas nozīmē, ka šis rezultāts nav blakus efekts atkritumu savācēja periodiskajai darbībai.

```
var attachTheListener = function(){
    var a1=[];
    var a2=[];
    for(var i=0; i<300000; i++){
        a1.push({name:"test"});
    }
    for(var i=0; i<300000; i++){
        a2.push({name:"test"});
    }
var oObject={
    p1:a1,
    p2:a2
}
var sName = oObject.p1[0].name;
document.getElementById('leakbutton').addEventListener("click", function () {
    alert(sName);
});
}
attachTheListener();
```

3.9. att. **Izvairīšanās no nevajadzīgi iela objekta turēšanas atmiņā**

Attēlā (skat. 3.9. att.) redzamais koda fragments parāda iepriekšējā koda fragmenta optimizētu variantu, kurā tā vietā, lai veiktu referenci uz *oObject* objektu notikuma apstrādes funkcijā, ārpus notikuma apstrādes funkcijas tiek definēts mainīgais ar tajā izmantojamo vērtību no *oObject* objekta.



3.10. att. **Atmiņas mērījumi, izvairoties no nevajadzīgi liela objekta turēšanas atmiņā**

Atmiņas mērījumi gan pirms, gan pēc piespiedu atkritumu savākšanas uzrāda ievērojami mazāku aizņemtās atmiņas daudzumu, jo atmiņā netiek glabāts viss *oObject* objekts (skat. 3.10. att.).

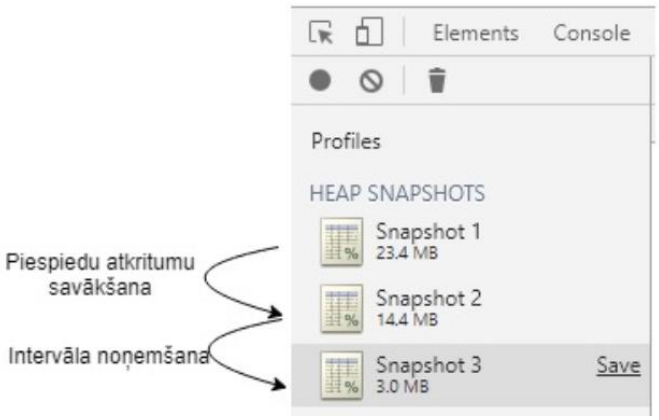
3.2.3 Nenoņemtie setInterval un setTimeout objekti

Taimeru un intervālu nenoņemšana var izraisīt neizmantotas atmiņas palikšanu atmiņā (skat. 4. pielikums). Piemēram, ja intervālā tiek veikta reference uz kādu objektu vai daļu no tā, tas tiks atstāts atmiņā, kamēr intervāls netiks noņemts [41].

```
var leakExample=function(){
    var posts = getData();
    setInterval(function() {
        var element = document.getElementById('leakbutton');
        //setInterval atstās atmiņā posts masīvu arī tad, kad izmantotais DOM elements ar id 'leakbutton' būs noņemts
        if(element) {
            element.innerHTML = posts[Math.floor(Math.random() * 10)].id;
        }
    }, 1000);
}
leakExample();
```

3.11. att. **setInterval atmiņas noplūde**

Attēlā (skat. 3.11. att.) redzamajā koda fragmentā tiek definēts intervāls, kas ik pēc sekundes veic kādu darbību ar DOM elementu ar id *leakbutton,* un tiek izmantots *leakExample* funkcijā definētais mainīgais *posts*, kas potenciāli satur lielu daudzumu datu. Tā kā intervālā ir veikta reference uz šo mainīgo, tas tiks paturēts atmiņā arī tad, kad funkcijas *leakExample* darbība būs beigusies, kā arī tad, ja DOM elements ar id *leakbutton* būs noņemts no lietotāja saskarnes.



3.12. att. **Atmiņas mērījumi setInterval atmiņas noplūdei**

Attēlā (skat. 3.12. att.) redzami atmiņas mērījumi koda fragmentā redzamajam scenārijam. Pirmais mērījums ir veikts, ielādējot lapu un noņemot DOM elementu ar id leakbutton, otrais mērījums ir veikts pēc piespiedu atkritumu savākšanas, lai nerastos mērījumu kļūdas, un pēdējais mērījums veikts pēc manuālas intervāla noņemšanas parauga programmas kodā. Ir redzams, ka mainīgais *posts* tiek paturēts atmiņā tik ilgi, kamēr intervāls nav noņemts.

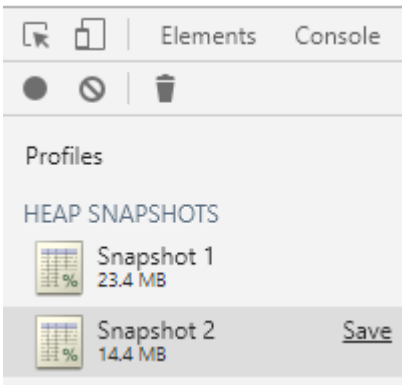
3.2.4 Nejauši globāli mainīgie

Ja, definējot mainīgos valodā JavaScript, netiek izmantoti atslēgas vārdi *var* vai *let*, tad tā vietā, lai tiktu izveidots lokāls mainīgais konkrētai funkcijai, tiek izveidots globāls mainīgais, kura aizņemtā atmiņa atkritumu savākšanas procesā netiks atbrīvota (skat. 5. pielikums) [41]. Šī JavaScript programmēšanas valodas īpatnība programmētājam ar pieredzi var šķist pašsaprotama, taču, ja pie apjomīgas vienas lapas tīmekļa lietotnes projekta strādā programmētāji, kas vēl nav guvuši pietiekamu pieredzi ar JavaScript programmēšanas valodu, programmatūras pirmkodā var rasties funkcijas, kurās *var* un *let* atslēgas vārdi netiek lietoti, un tiek aizņemta atmiņa, kas nav nepieciešama.

```
var leakExample=function(){
        posts = getData();//Šis kods izveido nevis lokālu mainīgo, bet window.posts
        var element = document.getElementById('leakbutton');
        if(element) {
            element.innerHTML = posts[Math.floor(Math.random() * 10)].id;
        }
}
leakExample();
```

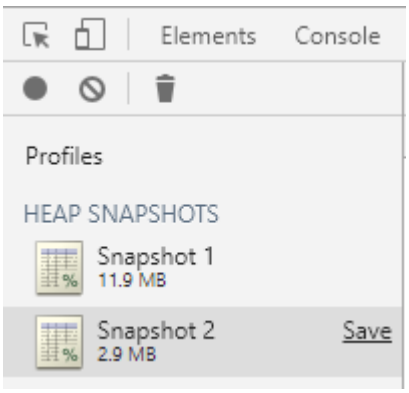
3.13. att. **Nejaušs globālais mainīgais**

Attēlā (skat. 3.13. att.) redzamajā koda fragmentā, definējot mainīgo *posts*, tiek izveidots globāls mainīgais, jo nav lietoti atslēgas vārdi *var* vai *let*. Funkcijas *leakExample* darbam beidzoties, šis mainīgais paliks aizņemtajā atmiņā – atkritumu savācējs to nevar atbrīvot.



3.14. att. **Atmiņas mērījumi nejaušam globālam mainīgajam**

Attēlā (skat. 3.14. att.) redzami aizņemtās atmiņas mērījumi pēc lapas ielādes un pēc piespiedu atkritumu savākšanas. Ir redzams, ka piespiedu atkritumu savākšana ir atbrīvojusi daļu no aizņemtās atmiņas.



3.15. att. **Atmiņās mērījumi, izvairoties no nejauša globāla mainīgā**

Attēlā (skat. 3.15. att.) redzami aizņemtās atmiņas mērījumi tam pašam scenārijam, kas minēts augstāk, izmantojot atslēgas vārdu *var* mainīgā *posts* definēšanai. Redzams, ka, salīdzinot iepriekšējiem mērījumiem gan ielādes brīdī, gan pēc piespiedu atkritumu iztīrīšanas tiek izmantots 11.5 MB mazāk atmiņas, kas apstiprina to, ka pirmā scenārija globālais mainīgais netiek iztīrīts no atmiņas.

Šādas kļūdas pieļaušanu var novērst, izmantojot funkcijas vai skripta augšpusē lietojot atslēgas vārdus *"use strict";* - ja tiek lietoti šie atslēgas vārdi, koda izpilde tiks apturēta vietā, kur mainīgais tiek definēts bez atslēgas vārda *var* vai *let*, un tiks parādīts kļūdas paziņojums.

3.2.5 Notikuma apstrādes funkcijas pārnešana augstākā līmenī

Veidojot sarakstus gadījumā, kad katram saraksta elementam nepieciešama kāda notikuma, piemēram, nospiešanas apstrādes funkcija, šo apstrādes funkciju iespējams pievienot pašam sarakstam tā vietā, lai to pievienotu katram elementam atsevišķi [21]. Tas samazina izmantotās atmiņas daudzumu. Šādu pieeju notikuma apstrādes funkciju pievienošanai sauc par notikumu delegāciju (*event delegation*). Notikumu delegācijas neizmantošana nav atmiņas noplūde, taču tas var palielināt aizņemtās atmiņas daudzumu, kad tam nav nepieciešamības.

A

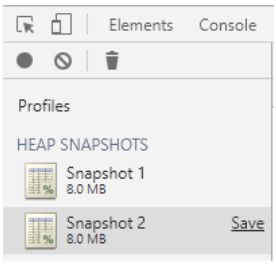
```
for(var i=0; i<30000; i++){
    var listElement=document.createElement("li");
    listElement.innerHTML = Math.floor(Math.random()*10)+1;
    books.appendChild(listElement);
    listElement.addEventListener('click', function(e) {
    if (e.target.innerHTML==="10") {
        alert("10 clicked.");
    }
});
}
```

B

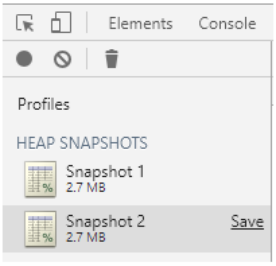
```
books.addEventListener('click', function(e) {
    if (e.target.innerHTML==="10") {
        alert("10 clicked.");
    }
}, false);


for(var i=0; i<30000; i++){
    var listElement=document.createElement("li");
    listElement.innerHTML = Math.floor(Math.random()*10)+1;
    books.appendChild(listElement);
}
```

A



B



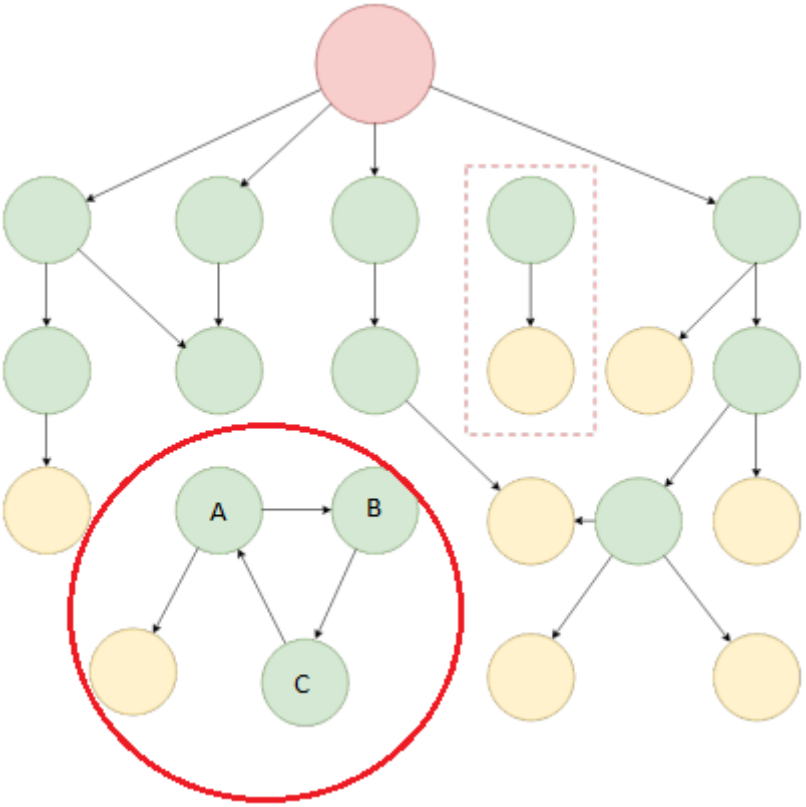
3.16. att. **Atmiņas mērījumi, pārnesot un nepārnesot notikuma apstrādes funkcijas**

Attēlā (skat. 3.16. att.) redzamais scenārijs A neizmanto notikumu delegāciju, tas katram saraksta elementam atsevišķi pievieno notikuma apstrādes funkciju. Attēlā redzamais scenārijs B izmanto notikumu delegāciju, notikuma apstrādes funkcija tiek pievienota nevis katram saraksta elementam atsevišķi, bet gan pašam saraksta DOM elementam, kas glabājas mainīgajā *books* (skat. **7.** pielikums). Ja notikuma apstrādes funkcijā nepieciešams izmantot pašu nospiesto saraksta elementu, to iespējams iegūt, izmantojot *target* atribūtu no objekta, kas tiek padots notikuma apstrādes funkcijai. Redzams, ka gadījumā ar 30000 elementu ir novērojama atšķirība atmiņas izmantošanā, kad tiek pievienota ļoti vienkārša apstrādes funkcija. Praksē notikuma apstrādes funkcijas mēdz būt ievērojami apjomīgākas, kas nozīmē to, ka atšķirība var būt lielāka.

3.2.6 Apļveida reference

Šāda veida atmiņas noplūde iespējama tikai ar references atkritumu savācējiem. Visi mūsdienu interneta pārlūki izmanto *Mark and Sweep*, tāpēc šī atmiņas noplūde attiecas tikai uz salīdzinoši senām interneta pārlūkprogrammām, piemēram, *Internet Explorer 6* un *Internet Explorer 7*. Iemesls, kāpēc references atkritumu savācēji nevar atbrīvot atmiņu, kas aizņemta ar apļveida referencēm, ir tāds, ka, izmantojot šāda veida atkritumu savācējus, atmiņa tiek atbrīvota tikai tiem objektiem, uz kuriem nenorāda neviena reference. Tas nozīmē, ka gadījumā,

ja eksistē objekts A, kam ir reference uz objektu B, un ja tajā pašā laikā objektam B eksistē reference uz objektu A, tad ne A, ne B objekta aizņemto atmiņu nevar atbrīvot (skat. 7. pielikums).



3.17. att. **Apļveida references vizuāls attēlojums** *[43]*

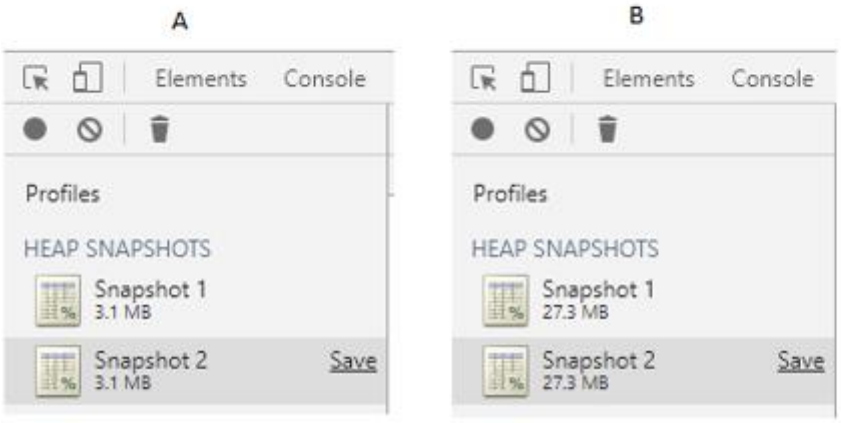
Attēlā (skat. 3.17. att.) redzams vizuāls apļveida references attēlojums – ar sarkano krāsu apvilkto objektu aizņemtā atmiņa nevar tikt atbrīvota, jo tiem veidojas apļveida reference.

A

```
function leakExample() {
    var a1={
        name:"",
        list:[]
    };
    var a2={
        name:"",
        list:[]
    };
    for(var i=0; i<300000; i++){
        a1.list.push({name:"test"});
    }
    for(var i=0; i<300000; i++){
        a2.list.push({name:"test"});
    }
    a1.name = a2.list[0].name;
    a2.name = a1.list[0].name;
}
leakExample();
```

B

```
function leakExample() {
    window.a1={
        name:"",
        list:[]
    };
    window.a2={
        name:"",
        list:[]
    };
    for(var i=0; i<300000; i++){
        a1.list.push({name:"test"});
    }
    for(var i=0; i<300000; i++){
        a2.list.push({name:"test"});
    }
    a1.name = a2.list[0].name;
    a2.name = a1.list[0].name;
}
leakExample();
```



3.18. att. **Merījumi parāda, ka apļveida references vairs nav aktuālas**

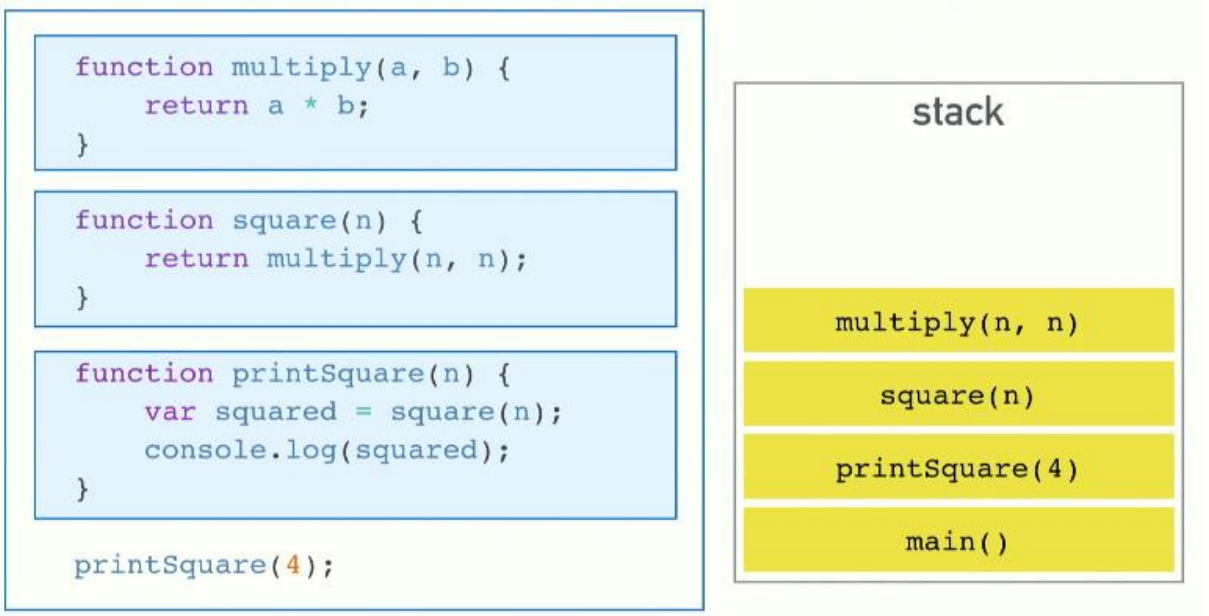
Attēlā (skat. 3.18. att.) redzami koda fragmenti, kuros parādīts piemēra scenārijs, kurā rodas apļveida reference – šajā gadījumā objektam *a1* ir reference uz objektu *a2* un otrādi. Scenārijiem veiktajos atbilstošajos aizņemtās atmiņas mērījumos redzams, ka gadījumā, kad objekti tiek izveidoti kā lokāli mainīgie funkcijai *leakExample*, atmiņa netiek aizņemta, savukārt gadījumā, kad šie objekti ir definēti kā globāli mainīgie, tiem aizņemtā atmiņa netiek atbrīvota, kas arī ir sagaidāmais rezultāts – tas parāda to, ka mūsdienu tīmekļa pārlūkprogrammās apļveida references vairs nav problēma.

4. VAIRĀKI PAVEDIENI VALODĀ JAVASCRIPT

JavaScript kods pēc noklusējuma darbojas vienā pavedienā, un, kamēr tajā notiek JavaScript koda izpilde, lietotāja saskarne ir bloķēta, un lietotājs nevar veikt darbības ar lietotāja saskarni. Nodaļā tiek izskatītas iespējas izmantot vairākus pavedienus valodā JavaScript, un tiek novērtēta šādas pieejas efektivitāte.

4.1 JavaScript valodas galvenais pavediens

Ņemot vērā faktu, ka vienas lapas tīmekļa lietotnēs liela daļa koda izpildes notiek interneta pārlūkprogrammā, var secināt, ka var rasties nepieciešamība interneta pārlūkprogrammā izpildīt kodu, kas aizņem ievērojamu laiku. Šāda koda izpildē ir jāņem vērā tas, ka JavaScript kods tiek izpildīts vienā pavedienā [30].



4.1. att. **Funkciju izsaukumu steks** *[30]*

JavaScript koda izpildes laikā funkciju izsaukumi tiek glabāti stekā. Attēlā (skat. 4.1. att.) redzams koda fragments un tam blakus redzams tam atbilstošs steks, kurā tiek glabāti funkciju izsaukumi. Steka apakšā ir *main* funkcijas izsaukums, kas šajā gadījumā ir kā funkcija, kurā iekļauts attēlā redzamais koda fragments, tālāk šajā funkcijā notiek *printSquare* funkcijas izsaukums, kas tiek ielikts stekā, *printSquare* funkcija savukārt izsauc *square* funkciju un tā šis process turpinās. Kad augšējā funkcija, kurā vairs nav citu izsaukumu, beidz darbu, tās

izsaukums tiek izņemts no steka, pēc tam, *square* funkcijai beidzot darbu, tā tiek izņemta no steka, un tā tas turpinās līdz steks tiek atbrīvots pilnībā – šajā gadījumā tas nozīmē tad, kad funkcija *main* beidz darbu [29]. Kamēr šis process notiek, jebkādas citas veicamās darbības, kas rodas, piemēram, šajā procesā veikta AJAX izsaukuma funkcija, kas tiek izsaukta, kad tiek saņemta šī AJAX izsaukuma atbilde, tiks izpildītas tikai tad, kad šis steks ir atbrīvots no visiem funkciju izsaukumiem. Šī procesa laikā arī lietotāja saskarne ir nereaģējoša – tas nozīmē, ka, ja, piemēram, kods, kas tiek izsaukts, nospiežot pogu, aizņem daudz laika, poga vizuāli paliks nospiesta un neatgriezīsies sākotnējā stāvoklī, līdz šis ilgstošais kods nebeigs izpildi. Tāpat šī koda izpildes laikā jebkuri citi lietotāja saskarnes elementi nereaģēs uz darbībām ar tiem, kā arī notikumu apstrādes funkcijas vairumā pārlūkprogrammu netiek reģistrētas kā veicamā darbība [25]. Veids, kā veikt ilgstoša koda izpildi bez iepriekš minētajām blakus parādībām, ir tīmekļa strādņu (Web worker) izmantošana.

4.2 Tīmekļa strādņi vairāku pavedienu izmantošanai

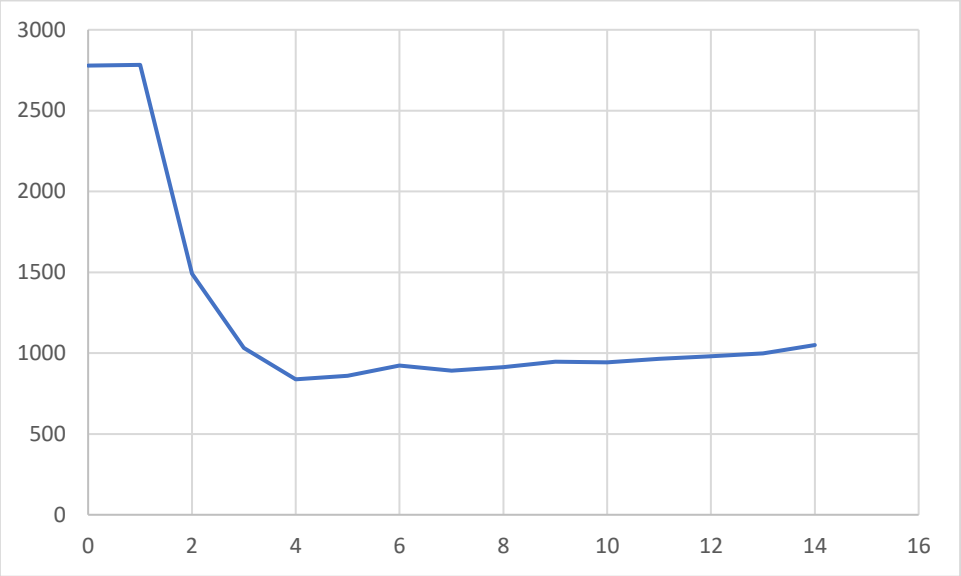
Tīmekļa strādņi ir objekti, kas tiek veidoti, izmantojot *Worker()* konstruktoru, un veic koda izpildi, kas definēts atsevišķā failā [6]. Šī koda izpilde notiek atsevišķi no galvenā pavediena, kas nozīmē, ka tā nerada iepriekš aprakstīto procesu – tā nepiepilda galvenā pavediena steku ar funkciju izsaukumiem, līdz ar to izvairās no šī steka piepildīšanas sekām – nereaģējošas lietotāja saskarnes un citu darbību bloķēšanas. Tīmekļa strādņiem eksistē ļoti būtiski ierobežojumi, kas nozīmē to, ka tīmekļa strādņu izmantošana ir derīga tikai konkrētiem gadījumiem. Tīmekļa strādņu galvenie ierobežojumi ir šādi:

- Nav piekļuves globālajam *Window* objektam
- Nav piekļuves lietotāja saskarnes dokumentu objektu modelim
- Nav piekļuves galvenā pavediena mainīgajiem

Sakarā ar to, ka tīmekļa strādņiem nav piekļuves galvenā pavediena mainīgajiem, veids, kā galvenais pavediens sazinās (padodot mainīgos) ar tīmekļa strādņiem, ir caur *postMessage()* metodi, kurā kā parametri tiek padoti sūtāmie mainīgie. Tīmekļa strādņi un galvenais pavediens viens no otra saņem mainīgos caur *onmessage()* metodi [6].

4.3 Vairāku pavedienu efektivitātes praktiska izvērtēšana

Darba ietvaros tika izveidoti piemēri, kas simulē ilga koda izpildi, un piemēri tika izpildīti ar dažādu daudzumu tīmekļa strādņu ar mērķi noskaidrot, kā tīmekļa strādņu daudzums ietekmē ilgā koda izpildes laiku – vai vairāk tīmekļa strādņu nozīmē ātrāku izpildi, vai arī konkrētiem gadījumiem eksistē optimāls tīmekļa strādņu daudzums (skat. 8. pielikums).

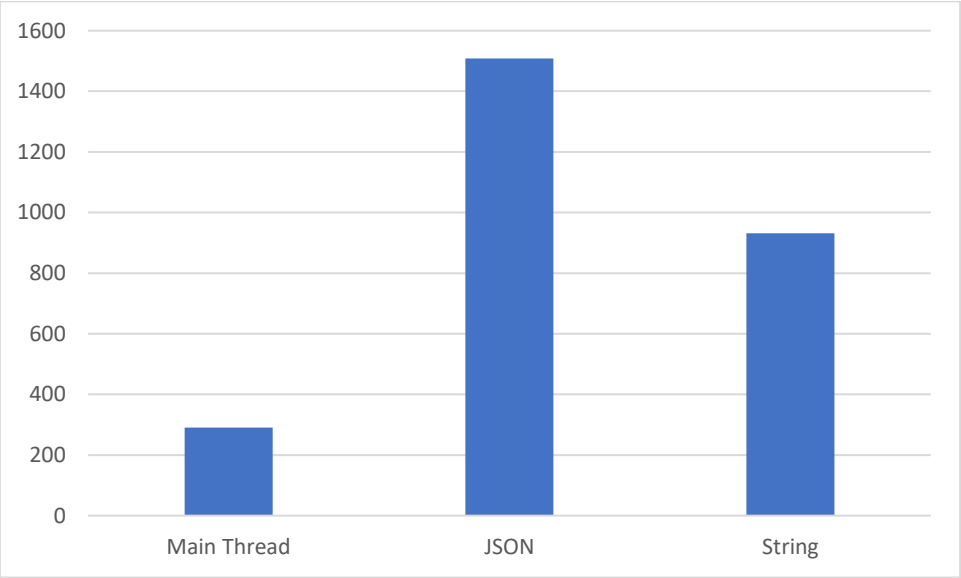


4.2 att. **Patērētais laiks atkarībā no tīmekļa strādņu daudzuma**

Attēlā (skat. 4.2 att.) uz x ass redzams tīmekļa strādņu, kam tika sadalīts uzdevums, skaits, un uz y ass redzams laiks milisekundēs, kas patērēts uzdevuma izpildei. Konkrētajā scenārijā tīmekļa strādņiem tika padots masīvs ar 10000 skaitļu, un katram skaitlim 10000 reižu tika veiktas 10 reizināšanas operācijas un 10 dalīšanas operācijas ar mērķi simulētu sarežģītus, laikietilpīgus aprēķinus, aprēķinu beigās tiek iegūts konkrēts skaitlis. Reizināšanas un dalīšanas operācijas tiek veiktas ar skaitļiem 1 un 2. Diagrammā redzams, ka nav būtiskas atšķirības starp izpildi galvenajā pavedienā un izpildi tīmekļa strādnī, izņemot to, ka, izpildot šos aprēķinus tīmekļa strādnī, lietotāja saskarne paliek reaģējoša. Redzams, ka, palielinot tīmekļa strādņu skaitu līdz 4, veiktspēja ir pakāpeniski uzlabojusies. Taču, sasniedzot lielāku tīmekļa strādņu skaitu par 4, ir redzama salīdzinoši neliela pakāpeniska veiktspējas samazināšanās, ko var izskaidrot ar laiku, kas tiek patērēts tīmekļa strādņu izveidošanā un ziņu apmaiņai starp galveno pavedienu un tīmekļa strādņiem.

4.4 Liela apjoma datu apstrādes efektivitāte tīmekļa strādņos

Tā kā tīmekļa strādņu galvenais mērķis ir ļaut veikt laikietilpīga koda izpildi, neietekmējot galvenā pavediena veiktspēju, var secināt, ka liela apjoma datu apstrāde var būt viens no gadījumiem, kur lietot tīmekļa strādņus. Sakarā ar šo faktu tika izveidots testpiemērs, kas nosūta tīmekļa strādnim JSON objektu, kas aizņem 12 megabaitus, tālākai apstrādei, un tika salīdzināts ar šī JSON objekta izpildi galvenajā pavedienā (skat. 9. pielikums). Sakarā ar to, ka tīmekļa resursos atrodama informācija par to, ka JSON objekta pārveidošana par simbolu virkni galvenajā pavedienā un pārveidošana par JSON objektu tīmekļa strādnī var radīt veiktspējas ieguvumus, tika pārbaudīts arī tas, cik lielu ieguvumu dod šī pieeja [31].



4.3. att. **Uzdevuma izpildei patērētā laika salīdzinājums**

Attēlā (skat. 4.3. att.) redzams, cik laika patērēts, veicot datu apstrādi katrā no iepriekš aprakstītajām pieejām. No šiem rezultātiem var secināt, ka tīmekļa strādņu izmantošana ne vienmēr rada veiktspējas ieguvumu – laiks, kas tiek patērēts saziņai starp tīmekļa strādni un galveno pavedienu, ir būtisks faktors, kas jāņem vērā. Var secināt, ka tas, vai tīmekļa strādņiem ir jāsūta apstrādājamie dati, ir būtisks faktors, kas jāizvērtē, pirms tīmekļa strādņi tiek izmantoti vienas lapas tīmekļa lietotnē. Rezultāti parāda arī to, ka JSON objekta pārveidošana par simbolu virkni pirms nosūtīšanas tīmekļa strādnim un pārveidošana atkal par JSON objektu rada manāmus veiktspējas uzlabojumus. Tā kā tīmekļa strādņi ir spējīgi veikt AJAX izsaukumus [6, 53], var secināt, ka no AJAX pieprasījumiem iegūtu datu apstrāde tīmekļa strādnī var palīdzēt novērs nereaģējošu lietotāja saskarni apstrādes laikā, jo AJAX izsaukums atgriezīs datus galvenajam pavedienam un tīmekļa strādnim vienādā laikā.

5. RENDERĒŠANAS VEIKTSPĒJA

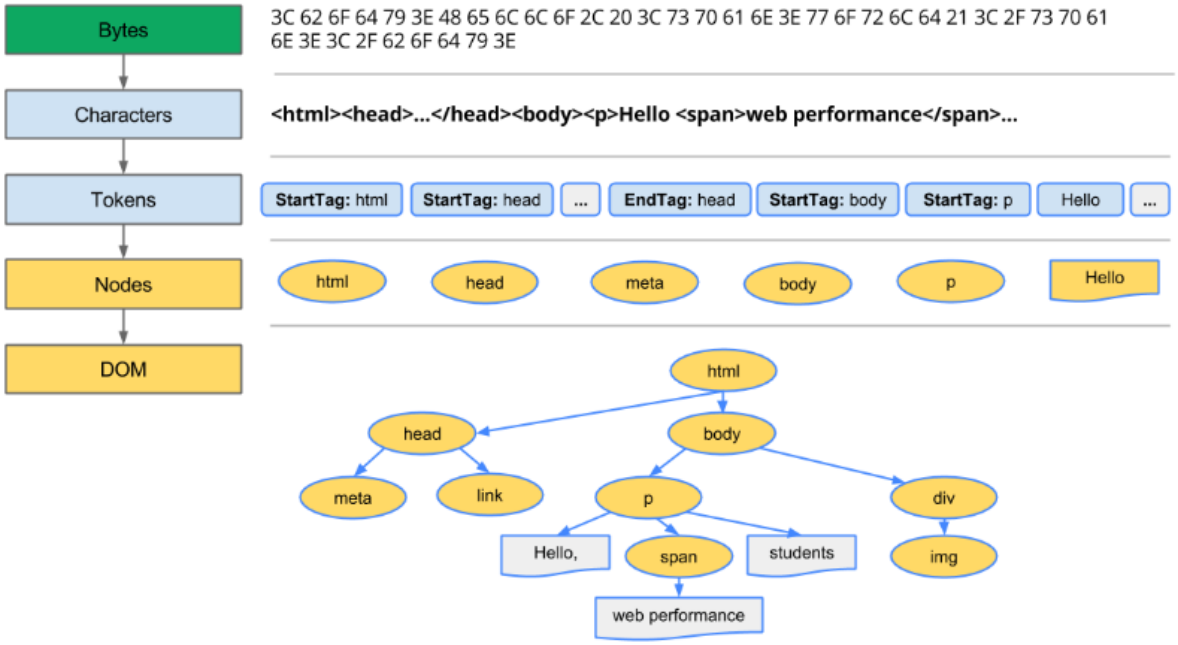
Nodaļā tiek aprakstīta veiktspēja sarakstu ģenerēšanai ar dažādām DOM metodēm dažādos scenārijos. Tiek aprakstīts DOM elementu renderēšanas process, un tiek veikti praktiski eksperimenti, lai salīdzinātu veiktspēju sarakstu ģenerēšanai, izmantojot dažādas DOM metodes, kā arī tiek veikti eksperimenti ar dažādas DOM struktūras saraksta elementiem, lai varētu analizēt DOM struktūras apjoma ietekmi uz veiktspēju dažādos scenārijos. Nodaļā izmantota 2017. gadā veikta pētījuma [53] ietvaros izveidota DOM metožu veiktspējas salīdzināšanas lietotne, kas šī bakalaura darba ietvaros papildināta ar apjomīgākas DOM struktūras saraksta elementiem un papildus scenārijiem tālākai veiktspējas analīzei un pētījumā [53] veikto atzinumu papildināšanai un apstiprināšanai.

5.1 HTML un CSS attēlošanas process interneta pārlūkprogrammās

Pirms HTML un CSS kods tiek attēlots interneta pārlūkprogrammā kā lietotāja saskarne, kuru lietotājs var izmantot, tiek veiktas vairākas secīgas darbības (skat. 5.1. att.).

5.1. att. **HTML un CSS attēlošanas process** *[2]*

Vispirms notiek HTML koda parsēšana, lai izveidotu lietotāja saskarnē attēlojamo dokumentu objektu modeli (DOM).

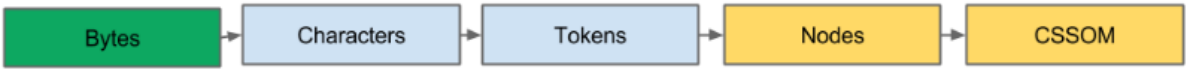


5.2. att. **Koda pārveidošana par dokumentu objektu modeli** *[11]*

Attēlā (skat. 5.2. att.) redzams, ka HTML koda parsēšana sākas ar šī koda baitu pārveidošanu par simbolu virkni, tālāk no šīs simbolu virknes tiek izveidoti marķējumi, kas ir, piemēram, atverošie vai aizverošie DOM elementu tagi vai teksts, kas atrodas starp tagiem, tālāk no šiem marķējumiem tiek izveidoti DOM elementu mezgli, kas tālāk tiek pārveidoti par dokumentu objektu modeli (DOM) [54, 11].

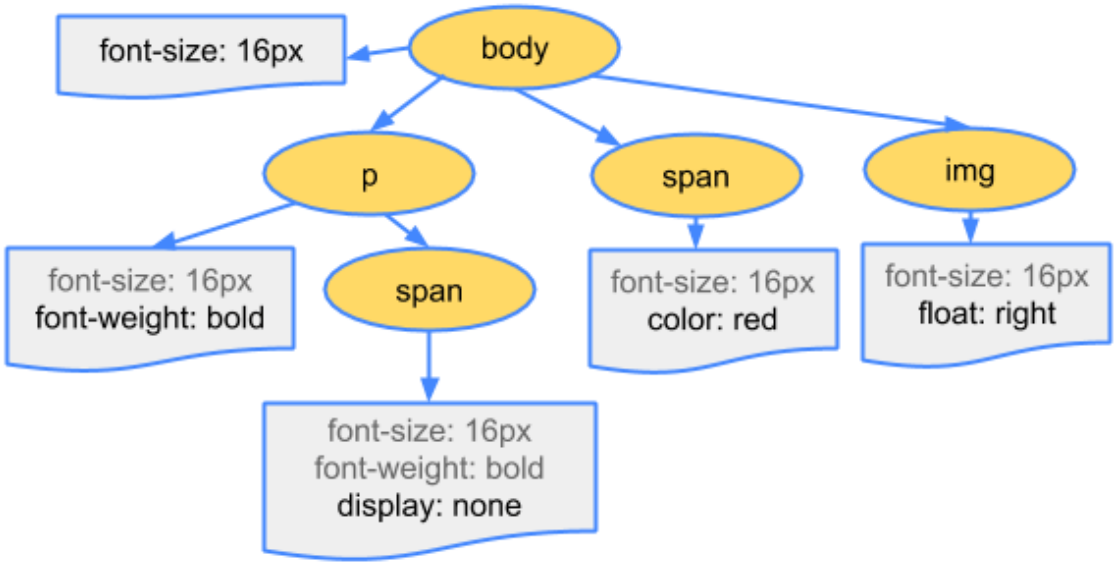
Laikā, kas patērēts HTML koda parsēšanā ietilpst arī laiks, kas patērēts kļūdaina HTML koda labošanā [53], sakarā ar to, ka HTML pieļauj kodu ar nepilnībām, piemēram, *td* elementam var tikt izlaistas beigas (*</td>*), un tiek uzskatīts, ka tas beidzas nākamā *td* elementa taga sākumā (*<td>*) [10], tāpēc, lai HTML koda parsēšanas laiks būtu optimāls, HTML kodam jābūt korektam.

Pēc DOM izveides tiek izveidots CSSOM (CSS Object Model). Tas tiek izveidots līdzīgi kā DOM, taču tas apraksta CSS kodā definētos stilus.



5.3. att. **CSSOM izveides process** *[11]*

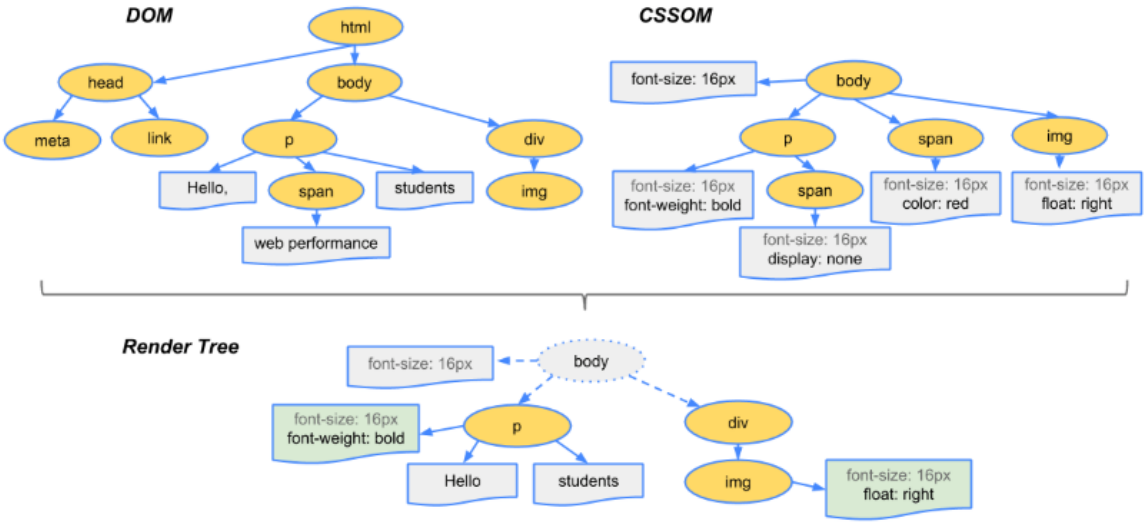
Attēlā (skat. 5.3. att.) redzams, ka CSSOM izveides soļi ir līdzīgi kā DOM izveides soļi. Šo soļu rezultātā tiek izveidota CSSOM koka struktūra, kas apraksta konkrētus CSS stilus konkrētiem DOM mezgliem.



5.4. att. **CSSOM struktūras piemērs** *[11]*

Attēlā (skat. 5.4. att.) redzams vienkāršs piemērs CSSOM struktūrai. Ar šo konkrēto piemēru var aprakstīt vienkāršu gadījumu, kā CSSOM palīdz tīmekļa pārlūkprogrammām piešķirt konkrētus stilus konkrētiem DOM mezgliem – piemērā *span* elements, kas atrodas *body* elementā tiks attēlots sarkans, taču, ja *span* elements atrodas p elementā, tas netiks attēlots, jo *span* elementiem, kas atrodas p elementos piešķirts stils *display: none* [11].

Kad izveidotas DOM un CSSOM koku struktūras, tās izmantojot, tiek izveidots renderēšanas koks, kas tiek izmantots tam, lai noteiktu visu redzamo DOM elementu izkārtojumu un veiktu to zīmēšanu uz ekrāna [12].

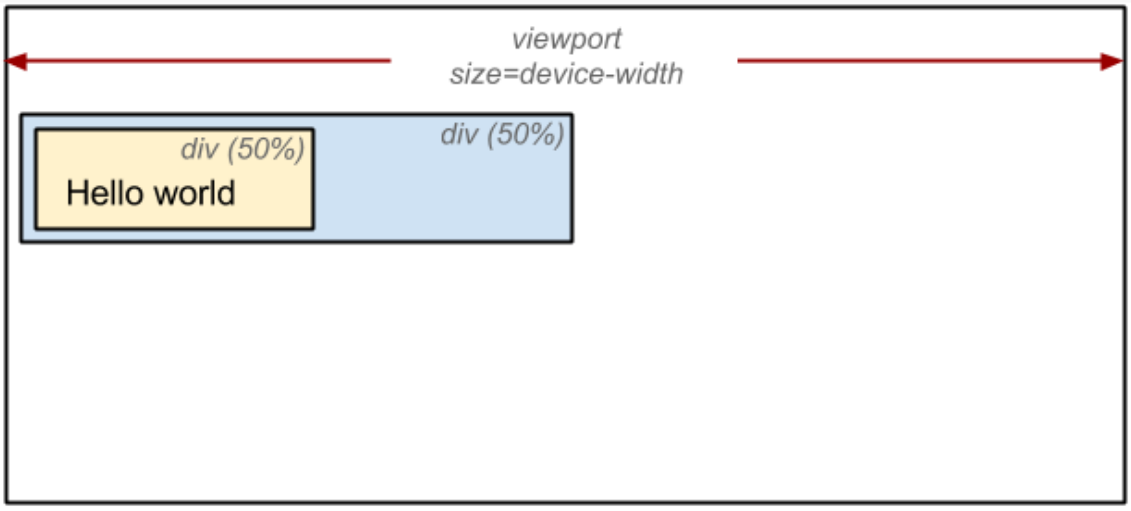


5.5. att. **Renderēšanas koks** *[12]*

Lai izveidotu renderēšanas koku, sākot no DOM koka saknes (parasti *body* elementa) tiek apstrādāts katrs redzamais DOM mezgls. Daļa DOM mezglu var nebūt redzami, piemēram, *script* vai meta mezgli netiek apstrādāti, jo tie nav redzami lietotāja saskarnē, kā arī DOM mezgli, kam piešķirti CSS stili, kas tos paslēpj, netiek iekļauti renderēšanas kokā, piemēram, attēlā redzamais *p* elementā esošais *span* mezgls netiek iekļauts renderēšanas kokā, jo tam piešķirta *display: none* īpašība. Katram redzamajam DOM mezglam tiek piešķirti tam atbilstošie stili no CSSOM (skat. 5.5. att.) [12, 53, 54].

Kad ir izveidots renderēšanas koks, tiek veikts izkārtošanas process, kurā tiek aprēķināts katra DOM elementa izmērs un atrašanās uz ekrāna.

```
<html>
  <head>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <title>Critial Path: Hello world!</title>
  </head>
  <body>
    <div style="width: 50%">
      <div style="width: 50%">Hello world!</div>
    </div>
  </body>
</html>
```



5.6. att. **HTML elementu izkārtojums** *[12]*

Attēlā (skat. 5.6. att.) redzams vienkāršs HTML kods un tā rezultāts uz ekrāna. Redzams, ka lapas saturs sastāv no *div* elementa, kas aizņem 50% no ekrāna platuma un tajā ir vēl viens *div* elements, kas aizņem 50% no vecāka elementa. Izkārtošanas procesa rezultāts ir "kastes modelis", kas precīzi zina katra elementa atrašanos un izmēru pikseļos uz ekrāna. Var secināt, ka vecāka izmēri ietekmē visu tā bērna elementu izmēru aprēķināšanu, līdz ar to tas var būt process, kas aizņem laiku [12, 53, 54].

Kad ir zināms, kuri lietotāja saskarnes DOM elementi ir redzami, zināmi to stili, atrašanās vieta un izmēri, notiek zīmēšanas process. Šajā procesā katra elementa pikseļi tiek aizpildīti ar tā vizuālo saturu – tekstu, ko elements satur, elementa krāsu, attēliem, apmalēm, ēnām u.c [14]. Piemēram, attēlā redzamajā *div* elementā tiek iezīmēts teksts "Hello world".

Lietotāja saskarnes DOM vecāku un bērnu elementi var būt zīmēti dažādos slāņos un pārklāt viens otru. Lai šiem vairākos slāņos zīmētajiem DOM elementiem nerastos nevēlamas pārklāšanās savā starpā, tiek veikts kompozitēšanas posms, kas sakārto šos slāņus pareizā secībā – lai pareizie elementi tiktu parādīti virspusē atbilstošajiem elementiem [14, 53, 54].

5.2 Apskatāmie saraksta ģenerēšanas scenāriji un to salīdzinājums

Nodaļā tiek aprakstīti apskatāmie saraksta ģenerēšanas scenāriji un tiek veikta to veiktspējas salīdzināšana (skat. 10. pielikums un 11. pielikums), lai atrastu problēmgadījumus un varētu veikt tālāku šo atrasto problēmgadījumu analīzi. Minētā pētījuma [53] ietvaros izveidotās lietotnes scenāriji ir A1 – A5. Pārējie scenāriji ir papildinājums, kas veikts šī bakalaura darba ietvaros. Minētajā pētījumā nav aprakstīta praktiska A1 – A5 scenāriju analīze, tā kā arī tas ir papildinājums minētajam pētījumam. Apskatāmie scenāriji ir sekojoši:

A1: Scenārijā *div* elementam ar cikla palīdzību katrā cikla iterācijā tiek pievienots saraksta elements, mainot tā *innerHTML* atribūtu, kā arī tiek nolasīts div elementa *offsetWidth* atribūts (skat. 5.7. att.). *offsetWidth* atribūta nolasīšanai konkrētajā scenārijā nav praktiska pielietojuma – tas tiek darīts tikai tāpēc, lai noskaidrotu šī atribūta nolasīšanas ietekmi uz veiktspēju.

```
for (var i = 0; i < iterations; i++) {
div.innerHTML += '<li>List item ' + i + '</li>';
w = div.offsetWidth;
}
console.timeEnd("Div - append with reflow " + iterations + "x");
}
```

5.7. att. **Scenārijs A1**

A2: Scenārijā *div* elementam ar cikla palīdzību katrā cikla iterācijā tiek pievienots saraksta elements, mainot tā *innerHTML* atribūtu, taču atšķirībā no A1 scenārija netiek nolasīts *offsetWidth* atribūts (skat. 5.8. att.).

```
for (var i = 0; i < iterations; i++) {
div.innerHTML += '<li>List item ' + i + '</li>';
}
console.timeEnd("Div - append " + iterations + "x");
}
```

5.8. att. **Scenārijs A2**

A3: Scenārijā *div* elementam ar cikla palīdzību katrā cikla iterācijā tiek pievienots saraksta elements, to konstruējot ar *document.createElement* metodi, un pievienošana tiek panākta ar *appendChild* metodi (skat. 5.9. att.).

```
for (var i = 0; i < iterations; i++) {
var el = document.createElement('li');
el.innerText = 'List item ' + i;
div.appendChild(el);
}
console.timeEnd("Div - appendChild " + iterations + "x");
}
```

5.9. att. **Scenārijs A3**

A4: Scenārijā tiek izveidots DOM fragments, kuram ar cikla palīdzību tiek pievienoti saraksta elementi, izmantojot *appendChild* metodi, un pēc tam izveidotais DOM fragments tiek ievietots lietotāja saskarnes DOM esošā div elementā (skat. 5.10. att.).

```
var fragment = document.createDocumentFragment();
for (var i = 0; i < iterations; i++) {
var el = document.createElement('li');
el.innerText = 'List append fragment ' + i;
fragment.appendChild(el);
}
div.appendChild(fragment);
```

5.10. att. **Scenārijs A4**

A5: Scenārijā ar cikla palīdzību tiek izveidota simbolu virkne, kas satur visu pievienojamo saraksta elementu HTML kodu, un pēc tam šī simbolu virkne tiek uzstādīta kā *innerHTML* atribūts lietotāja saskarnes DOM esošā div elementā (skat. 5.11. att.).

```
var html = '';
console.time("Div - append join " + iterations + "x");
for (var i = 0; i < iterations; i++) {
html += '<li>List item ' + i + '</li>';
}
div.innerHTML = html;
console.timeEnd("Div - append join " + iterations + "x");
```

5.11. att. **Scenārijs A5**

A6: Šis scenārijs ir identisks scenārijam A3, taču katrā cikla iterācijā tiek veikta *offsetWidth* īpašības nolasīšana (skat. 5.12. att.).

```
for (var i = 0; i < iterations; i++) {
var el = document.createElement('li');
el.innerText = 'List item ' + i;
div.appendChild(el);
w=div.offsetWidth;
}
```

5.12. att. **Scenārijs A6**

B1, B2, B3, B4, B5, B6: Šie scenāriji atbilst attiecīgajiem iepriekš aprakstītajiem A scenārijiem, taču B scenārijos ir papildināta DOM struktūra (skat. 5.14. att.), lai testpiemēri būtu pietuvināti reālai tīmekļa lietotnes situācijai, kā arī tāpēc, lai redzētu to, cik ļoti laiks, kas vajadzīgs elementu ģenerēšanai, ir atkarīgs no šādas DOM struktūras papildināšanas. B scenāriju saraksta elementu struktūra veidota līdzīgi kā *Twitter* sociālā tīkla ierakstu saraksta elementi (skat. 11. pielikums).

Attēlā (skat. 5.13. att.) redzama scenārijos A izmantotā saraksta elementu DOM struktūra.

```
<li>List item 0</li>
```

5.13. att. ***Scenāriju A DOM struktūra***

Attēlā (skat. 5.14. att.) redzama scenārijos B izmantotā saraksta elementu DOM struktūra.

```
▼<div class="row input-wrap">
    ::before
  ▼<div class="media">
    ▼<div class="body-container">
      ▼<div class="row">
          ::before
        ▼<div class="media-body">
          ▼<h4 class="post-heading">
              <span>Jake Smith0</span>
              <a>@jsmith0</a>
              <span>0 minutes ago</span>
            </h4>
          ▼<p class="post-body">
              "Lorem Ipsum has been the 0 industry's standard dummy text ever since the 1500s, when an
              unknown printer took a galley of type and scrambled it to make a type specimen book."
            </p>
            <p class="post-rating">Rating: 0</p>
          </div>
          ::after
        </div>
      ▼<div class="row">
          ::before
        ▼<div class="bottom-links">
            <a class="post-property" href="#">Expand</a>
            <a class="post-property" href="#">Reply</a>
            <a class="post-property" href="#">Repost (0)</a>
            <a class="post-property" href="#">Star (0)</a>
            <a class="post-property" href="#">More</a>
          </div>
          ::after
        </div>
      </div>
    </div>
    ::after
  </div>
```

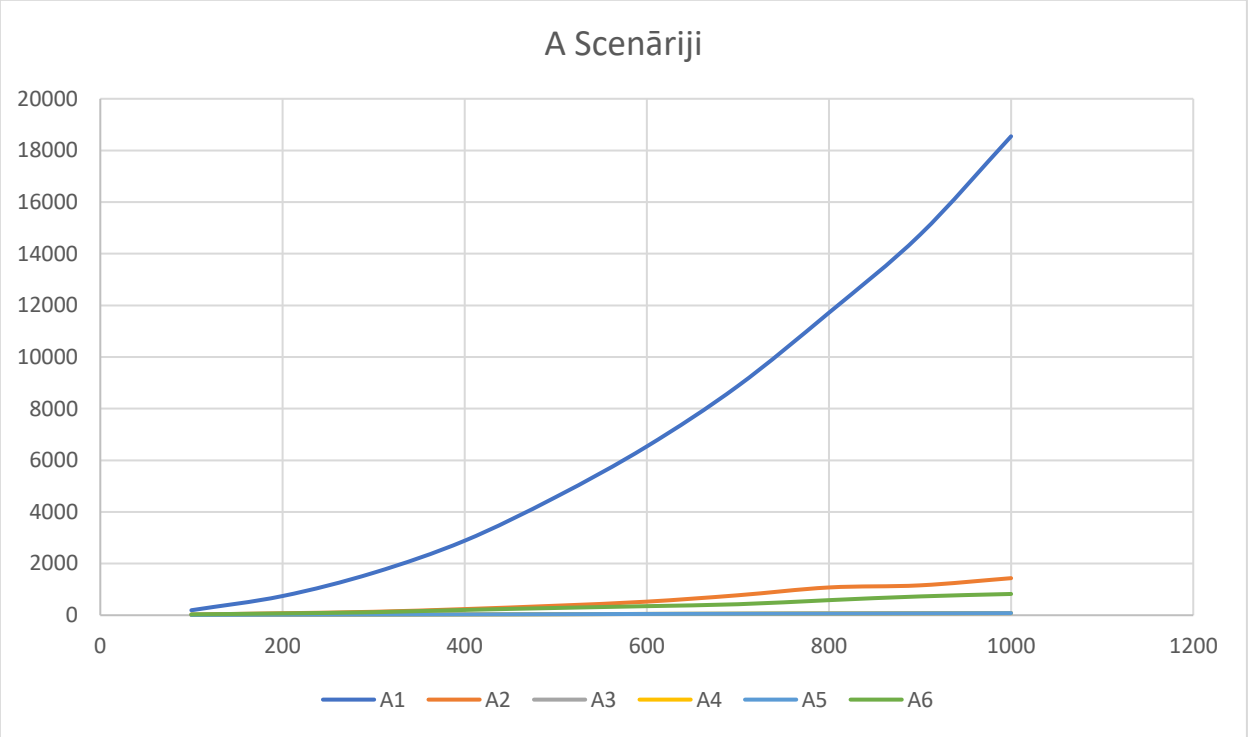
5.14. att. **Scenāriju B DOM struktūra**

Ātrdarbības mērījumi katram scenārijam tika veikti ar dažādiem saraksta elementu skaitiem no 100 līdz 1000 ar intervālu 100, lai katram scenārijam būtu redzams elementu skaits, pie kura iestājas lietotājam manāmas veiktspējas problēmas.

Scenārijos A1, B1, A6 un B6 tiek izmantota funkcija, kas izraisa piespiedu izkārtošanas procesu. Funkcijas, kas izraisa piespiedu izkārtošanas procesu, ir šādas [61, 53]:

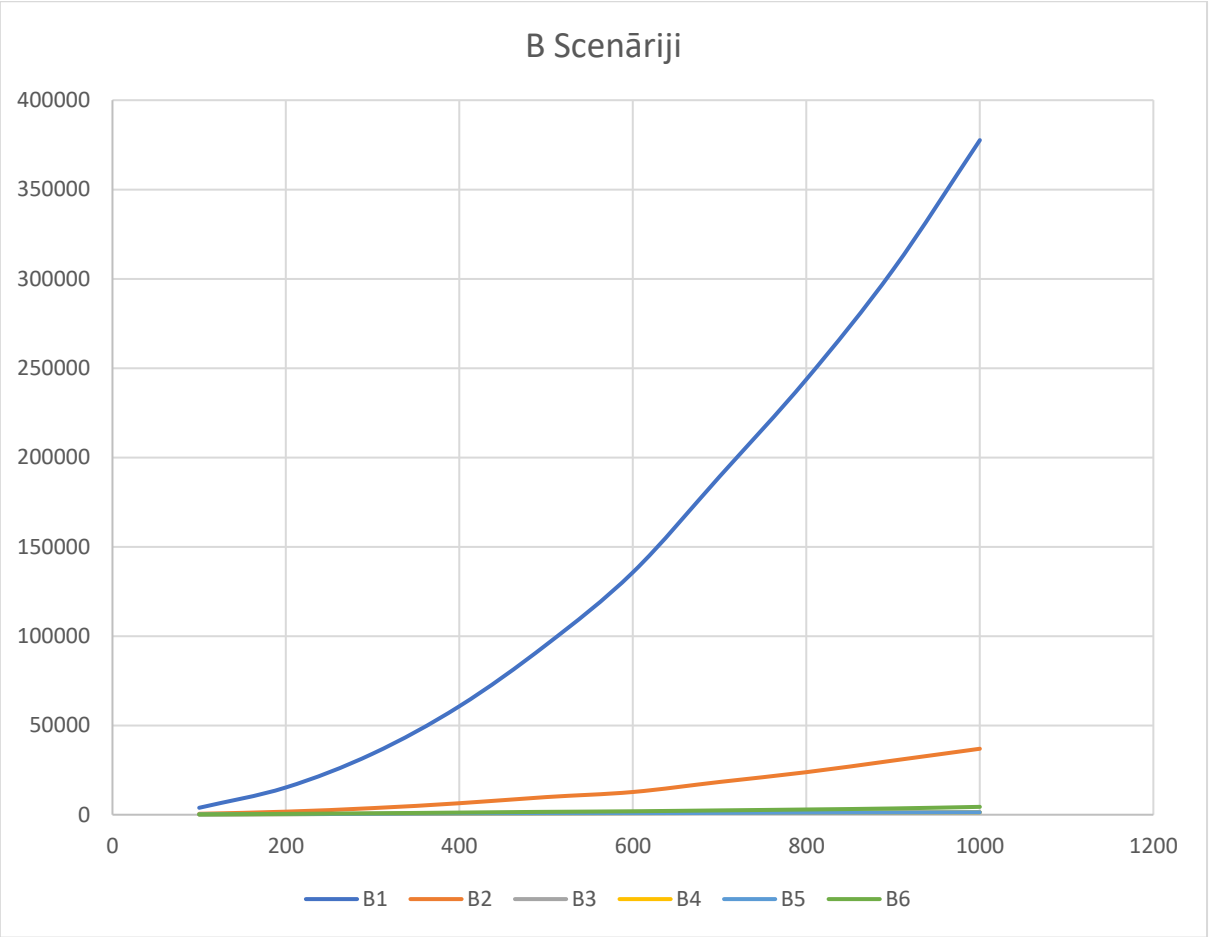
- elem.offsetLeft, elem.offsetTop, elem.offsetWidth, elem.offsetHeight, elem.offsetParent
- elem.clientLeft, elem.clientTop, elem.clientWidth, elem.clientHeight
- elem.getClientRects(), elem.getBoundingClientRect()
- elem.scrollBy(), elem.scrollTo()
- elem.scrollIntoView(), elem.scrollIntoViewIfNeeded()
- elem.scrollWidth, elem.scrollHeight
- elem.scrollLeft, elem.scrollTop
- elem.focus()
- elem.computedRole, elem.computedName
- elem.innerText
- window.getComputedStyle()

- window.scrollX, window.scrollY
- window.innerHeight, window.innerWidth
- window.getMatchedCSSRules()
- inputElem.focus()
- inputElem.select(), textareaElem.select()
- mouseEvt.layerX, mouseEvt.layerY, mouseEvt.offsetX, mouseEvt.offsetY
- range.getClientRects(), range.getBoundingClientRect()



5.15. att. **Patērētais laiks atkarībā no saraksta elementu skaita**

Attēlā (skat. 5.15. att.) attēloti ātrdarbības mērījumi A scenārijiem ar saraksta elementu skaitu uz x ass un patērēto laiku milisekundēs uz y ass. Redzams, ka scenārijiem A6, A1 un A2 tiek patērēts ievērojami lielāks laiks, nekā scenārijiem A3, A4 un A5.



5.16. att. **Patērētais laiks atkarībā no saraksta elementu skaita**

Attēlā (skat. 5.16. att.) attēloti ātrdarbības mērījumi B scenārijiem, un tāpat kā iepriekšējā grafikā – uz x ass ir ģenerēto elementu skaits, un uz y ass ir tam patērētais laiks. Tāpat kā A scenārijos – B6, B1 un B2 scenāriju ātrdarbība ir ievērojami sliktāka, nekā scenārijiem B3, B4 un B5. Sakarā ar veikto ātrdarbības mērījumu rezultātiem tālāk analizējamie scenāriji ir A6, A1, A2, B6, B1 un B2. Tā kā ātrdarbības mērījumi parāda, ka scenāriju A3, A4 un A5 izpildes laiki ir savstarpēji aptuveni vienādi, kā arī B3, B4 un B5 izpildes laiki ir savstarpēji aptuveni vienādi, kā korekti un ātri izpildāmi scenāriji salīdzināšanai tiek izvēlēti A3 un B3.

5.3 DOM elementu pārzīmēšanu izsaucošas darbības

Savstarpēji līdzīgi scenāriji, kas salīdzinājumā ar alternatīvajiem scenārijiem rādīja vājākus rezultātus, ir scenāriji A1 un B1. Šajos scenārijos ar cikla palīdzību katrā cikla iterācijā vecāka *div* elementam tiek pievienots saraksta elements, nolasot vecāka *div* elementa *innerHTML* vērtību, tai pievienojot jauno saraksta elementu. Šie scenāriji sakrīt ar scenārijiem attiecīgi A2 un B2, taču scenārijos A1 un B1 *div* elementam tiek izsaukta funkcija *offsetWidth*. Šī funkcija

atgriež lietotāja saskarnes DOM esoša elementa fiziski aizņemto platumu uz ekrāna, ieskaitot apmales (*border*), polsterējumu (*padding*), un ritināšanas joslu (*scroll bar*), ja elementam tāda pastāv [8]. Apakšnodaļā tiek izskatīts arī scenāriju A6 un B6 salīdzinājums ar scenārijiem attiecīgi A3 un B3, jo šie ir scenāriji, kas ir vienādi un arī atšķiras tikai ar to, ka scenārijos A6 un B6 tiek izsaukta funkcija *offsetWidth*.

5.1. tabula. **Scenārijos patērētais laiks atkarībā no elementu skaita**

	A1	A2	A3	A6
100	193	25.7	10.6	31.5
200	742.2	76.1	18.4	70.1
300	1642.3	132.5	24.3	117.3
400	2884.1	237.3	31.8	205.3
500	4574.2	367.9	39.2	280.9
600	6534.8	525.4	49.5	351.4
700	8876.3	772	58.3	425.3
800	11713.3	1075.2	66.5	583.4
900	14733.2	1156.7	73.8	728.4
1000	18546.6	1434.6	82.1	823.2

Tabulā (skat. 5.1. tabula) redzams, ka scenārijā A1 ģenerējot pat tikai 100 saraksta elementus ar ļoti vienkāršu DOM struktūru, tiek sasniegts atbildes laiks, kas nav momentāns. Salīdzinājumā ar līdzīgo A2 scenāriju A1 scenārijā 100 saraksta elementu ģenerēšana aizņem 7,51 reizes lielāku laiku, savukārt 1000 elementu ģenerēšana aizņem 12,93 reizes lielāku laiku, no kā var secināt, ka, pielietojot *offsetWidth* katrā iterācijā šajā scenārijā, veiktspēja pasliktinās ar paātrinājumu, saraksta elementu skaitam pieaugot. Scenārijā A6 atbildes laiks, kas nav momentāns, novērojams jau pie 300 vienkāršas struktūras elementiem. Salīdzinājumā ar līdzīgo A3 scenāriju A6 scenārijā 100 saraksta elementu ģenerēšana aizņem 2,97 reizes lielāku laiku, savukārt 1000 saraksta elementu ģenerēšana aizņem 10,02 reizes lielāku laiku, kas liecina, ka arī šajos scenārijos *offsetWidth* izmantošana katrā cikla iterācijā rada paātrinātu veiktspējas pasliktināšanos, saraksta elementu skaitam pieaugot.

5.2. tabula. **Scenārijos patērētais laiks atkarībā no elementu skaita**

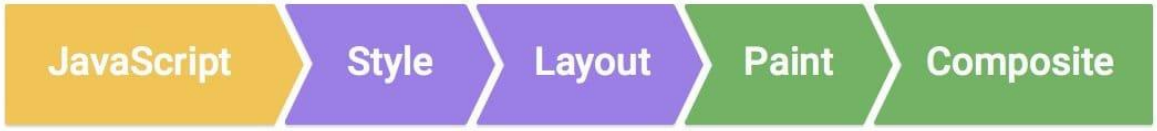
	B1	B2	B3	B6
100	3872.5	510.3	151.9	196.1
200	15270.7	1712.7	310.6	461.9
300	34160.5	3696.5	493.3	785.3
400	60739.5	6425.1	686.5	1201.8
500	95095.3	9864.3	803.4	1612.4
600	135703.1	12725.8	891.9	1927.1
700	189347.4	18340.4	979.8	2415.4
800	243743.9	23808	1118.4	2923.9
900	305114.1	30311.5	1237.3	3498.2
1000	377636.4	36929.7	1343.6	4406.3

Tabulā (skat. 5.2. tabula) redzams, ka ar palielinātu DOM struktūru starpība starp atšķirīgajiem scenārijiem ir ievērojami lielāka. B3 scenārijā ģenerējot 200 saraksta elementus, sasniegts atbildes laiks, kas nav momentāns, taču neliela aizkave ir sagaidāms rezultāts, ģenerējot lielu daudzumu neprimitīvas struktūras saraksta elementu, to darot datorā, kam piemīt testos izmantotā datora jauda. B1 scenārijā redzams, ka pat ģenerējot 200 saraksta elementu rodas ievērojama aizkave, kas pārsniedz cilvēka uzmanības intervālu. B1 scenārijā ģenerējot 100 saraksta elementus, tiek patērēts 7,59 reizes ilgāks laiks, nekā scenārijā B2, savukārt 1000 saraksta elementu ģenerēšanā atšķirība ir 10,22 reizes. B6 scenārijā ģenerējot 100 saraksta elementus, tiek patērēts 1.29 reizes ilgāks laiks, nekā scenārijā B3, savukārt 1000 saraksta elementu ģenerēšanā atšķirība ir 3.28 reizes. Arī B scenārijos var secināt, ka, elementu skaitam pieaugot, *offsetWidth* izmantošana pasliktina veiktspēju paātrināti.

Var secināt, ka visos scenārijos piespiedu izkārtošanas process izraisa nevis lineāru, bet paātrinātu veiktspējas pasliktināšanos, saraksta elementu skaitam pieaugot, salīdzinot ar scenārijiem, kur piespiedu izkārtošanas process netiek izsaukts. Tomēr vienkāršas DOM struktūras elementu veiktspējas pasliktināšanās kāpums ir straujāks, saraksta elementu skaitam pieaugot. Neskatoties uz to, gan aizkavi, kas ir pamanāma, gan aizkavi, kas pārsniedz cilvēka uzmanības intervālu (skat. apakšnodaļu 2.1), ir iespējams sasniegt ar ievērojami mazāku saraksta elementu skaitu, ja tiek izmantota apjomīgāka saraksta elementa DOM struktūra.

A1, B1, A6 un B6 scenāriju vājā veiktspēja ir saistīta ar nodaļā 4.1 izskatīto renderēšanas procesu, kā arī ar pārlūku JavaScript koda izpildes un lietotāja saskarnes renderēšanas optimizāciju ierobežojumiem. Process, kā notiek lietotāja saskarnē attēlotā satura izmaiņas parasti ir sekojošs (skat. 5.17. att.):

- JavaScript kods ir veicis aprēķinus un atbilstoši veicis manipulācijas ar lietotāja saskarnes DOM,
- Tiek aprēķināts, kuri stili ir piešķirti kuriem lietotāja saskarnes DOM elementiem pēc JavaScript veiktajām izmaiņām,
- Notiek izkārtošanas process elementu vietas un izmēru aprēķināšanai,
- Tiek veikta aprēķināto elementu zīmēšana,
- Tiek veikta uzzīmēto elementu kompozitēšana [14, 54].



5.17. att. **Attēlotā satura izmaiņu process** *[14]*

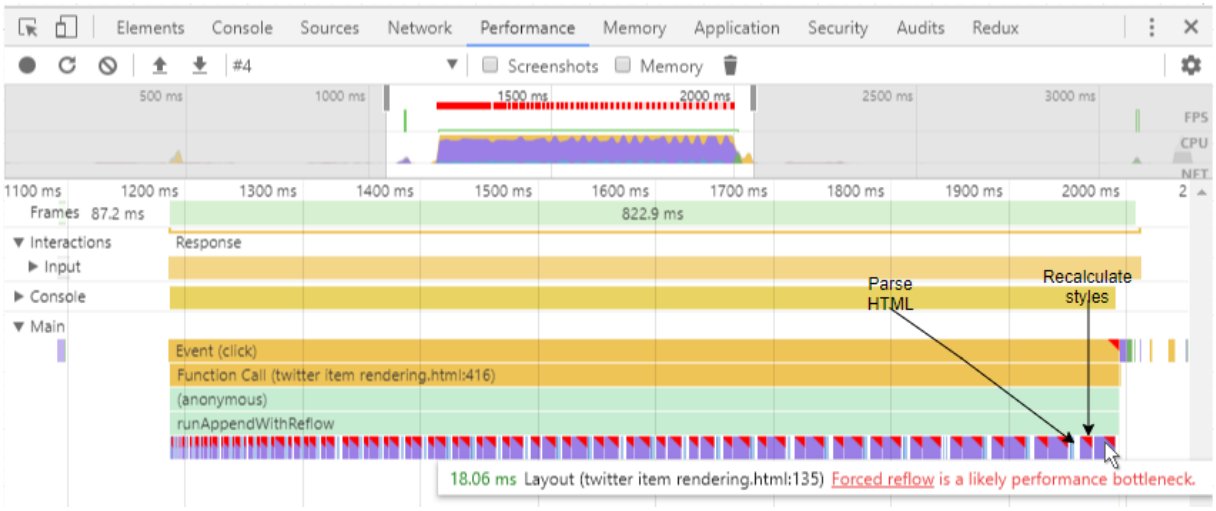
Apskatītajos scenārijos katrā sarakstu veidojošā cikla iterācijā notiek JavaScript koda izpilde, kas izmaina lietotāja saskarnē attēloto saturu. Tā kā lietotāja saskarnē esošā satura izmainīšana ir laikietilpīgs process, tīmekļa pārlūkprogrammas šo procesu optimizē – lietotāja saskarnes elementi, kuriem veiktas izmaiņas, tiek atzīmēti kā "netīri", un to stilu aprēķināšana, izkārtošana, zīmēšana un kompozitēšana notiks tikai pēc JavaScript koda izpildes beigšanās [2, 53].

Taču eksistē HTML elementu īpašību nolasīšanas, kas izraisīs piespiedu izkārtošanas procesu. Tas nozīmē, ka visur JavaScript kodā, kur tiek nolasīta kāda no šīm HTML elementu īpašībām, koda izpildei nebeidzoties, tiek veikts izkārtošanas process, kā arī pārējie procesi, kas redzami attēlā pa labi no izkārtošanas procesa. Jāpiemin, ka šāds piespiedu izkārtošanas process notiks tikai tad, ja līdz īpašības nolasīšanas brīdim ir veiktas tādas izmaiņas, kas var būt ietekmējušas lietotāja saskarnē attēloto saturu, piemēram, kāda elementa pievienošana vai stila nomainīšana. Tāpēc iespēju robežās būtu nepieciešams koda izpildes laikā sākumā veikt izmaiņas lietotāja saskarnē attēlotajā saturā un tikai tad nolasīt nepieciešamās īpašības, lai samazinātu piespiedu izkārtošanas procesu skaitu [53, 25].

A1 un B1 scenārijos katrā cikla iterācijā, kad tiek pievienots HTML elements, tiek nolasīta saraksta elementu vecāka div elementa *offsetWidth* īpašība, kas nozīmē, ka interneta pārlūkprogrammai ir fiziski jāveic izkārtošanas process, lai varētu nolasīt šo īpašību, kas izskaidro ievērojamo veiktspējas trūkumu šajos scenārijos [9, 53].

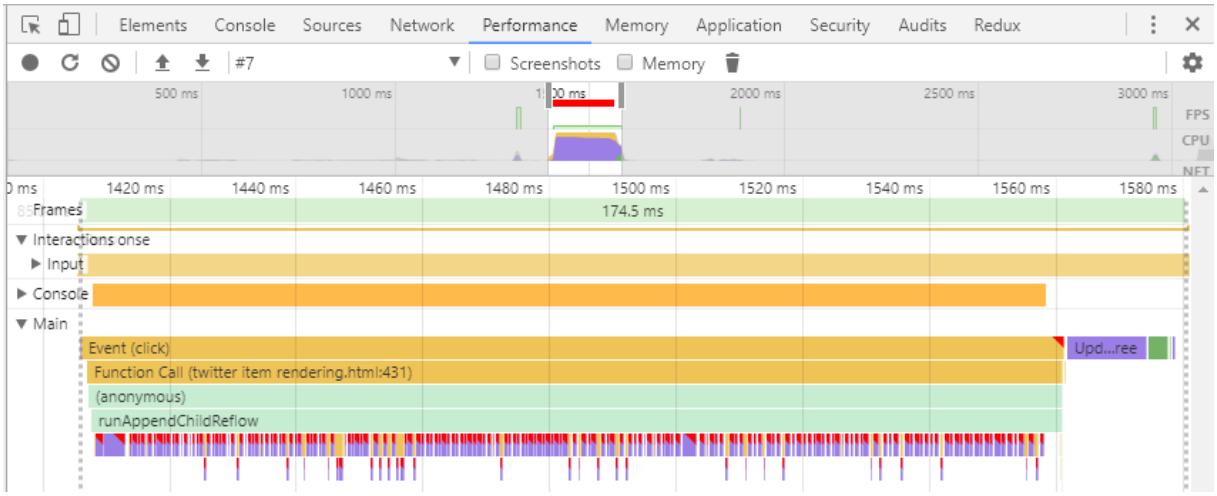
Iemesls tam, ka A1, B1 scenāriju veiktspēja ir ievērojami sliktāka, nekā scenāriju A6 un B6 veiktspēja ir tas, ka *innerHTML* īpašība katru reizi no jauna parsē tai padoto kodu, izveido

dokumenta fragmentu un aizvieto elementa, kam *innerHTML* pielietots, saturu ar jaunajiem elementiem. Tas nozīmē, ka katrā cikla iterācijā interneta pārlūkprogramma veic nevis tikai konkrētajā iterācijā pievienotā elementa izkārtošanas procesu, bet gan katru reizi veic visu saraksta elementu izkārtošanas procesu.



5.18. att. **B1 scenārija 30 saraksta elementu renderēšanas mērījums**

Attēlā (skat. 5.18. att.) redzams veiktspējas ieraksts 30 saraksta elementu ģenerēšanai scenārijā B1. Redzams, ka izstrādātāja rīkos violetie taisnstūri parāda laiku, kas aizņemts, veicot stilu aprēķināšanas un izkārtošanas procesus, pirms kuriem ir salīdzinoši neliels laiks, kas patērēts HTML koda parsēšanai. Ir redzams, ka kreisajā pusē violetie trijstūri ir ievērojami šaurāki, un tie kļūst pakāpeniski platāki ar katru nākamo izkārtošanas procesu. Tas skaidrojams ar to, ka saraksta ģenerēšanas sākumā sarakstā ir tikai daži elementi, kam jāveic izkārtošanas process, savukārt ar katru nākamo cikla iterāciju elementu, kam jāveic izkārtošana, skaits pieaug.



5.19. att. **B6 saraksta elementu renderēšanas mērījums**

Attēlā (skat. 5.19. att.) redzams līdzīgs veiktspējas ieraksts scenārijam B6. Tāpat kā scenārijā B1 ir redzams, ka tiek veikts izkārtošanas process katra elementa pievienošanas laikā, taču nav redzama tendence izkārtošanas procesā pavadītā laika pieaugumam – tas saistīts ar to, ka, izmantojot *appendChild()* metodi, viss elementa saturs netiek aizvietots ar citu kodu, bet gan esošajam kodam tiek pievienots tikai jaunais elements, kas nozīmē to, ka pārlūkprogrammai ir jāveic tikai pievienotā elementa izkārtošanas process, lai varētu nolasīt vecāka elementa *offsetWidth* īpašību.

5.4 Saraksta papildināšana ar innerHTML

Savstarpēji līdzīgi scenāriji, kas salīdzinājumā ar alternatīvajiem scenārijiem radīja ievērojami sliktākus rezultātus, bija A2 un B2. Šajos scenārijos ar cikla palīdzību katrā cikla iterācijā div elementam tiek pievienots saraksta elements, papildinot div elementa *innerHTML* ar pievienojamā elementa HTML kodu. Tā kā *innerHTML* ir tīmekļa lietotņu izstrādē plaši izmantota metode, var šķist, ka tās darbība neatšķiras no, piemēram, *appendChild* metodes.

5.3. tabula. **Scenārijos patērētais laiks atkarībā no elementu skaita**

	A2	A3
100	25.7	10.6
200	76.1	18.4
300	132.5	24.3
400	237.3	31.8
500	367.9	39.2
600	525.4	49.5
700	772	58.3
800	1075.2	66.5
900	1156.7	73.8
1000	1434.6	82.1

Tabulā (skat. 5.3. tabula) redzams, ka ātrdarbība scenārijiem ievērojami atšķiras. Redzams, ka scenārijā A2 atbildes laiks, kas nav momentāns, ir manāms jau sākot no 300 vienkāršas DOM struktūras saraksta elementu ģenerēšanas.

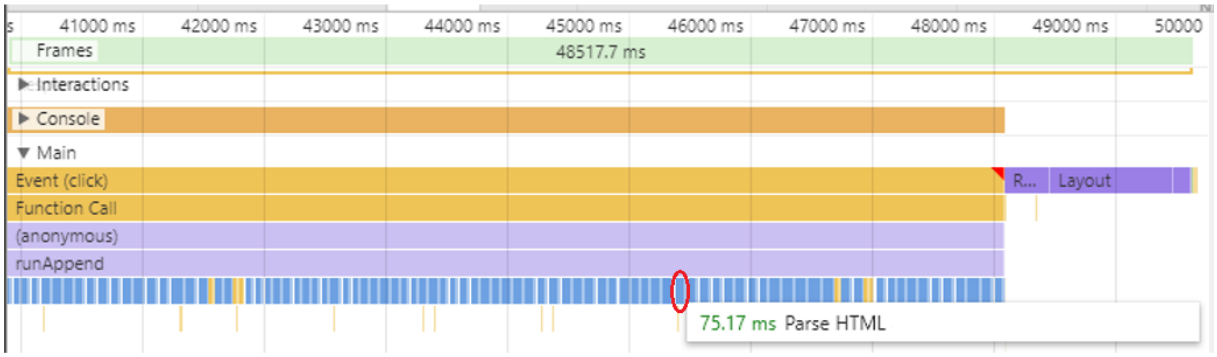
5.4. tabula. **Scenārijos patērētais laiks atkarībā no elementu skaita**

	B2	B3
100	510.3	151.9
200	1712.7	310.6
300	3696.5	493.3
400	6425.1	686.5
500	9864.3	803.4
600	12725.8	891.9
700	18340.4	979.8
800	23808	1118.4
900	30311.5	1237.3
1000	36929.7	1343.6

Tabulā (skat. 5.4. tabula) redzams, ka, palielinot DOM struktūru līdz tādas lietotnes kā *Twitter* saraksta elementiem, laiks, kas patērēts saraksta ģenerēšanai, ir ievērojami palielinājies B2 un B3 scenārijos, salīdzinot ar attiecīgi scenārijiem A2 un A3. B2 scenārijā jau 100 elementu ģenerēšana rada atbildes laiku, kas lietotājam ir pamanāms. No B2 un B3 rezultātiem var secināt, ka optimālai saraksta vai tabulas elementu ģenerēšanai jāpievērš sevišķi liela nozīme, ņemot vērā ievērojamo patērētā laika kāpumu, salīdzinot ar mazākas DOM struktūras elementiem. Vēl jāņem vērā, ka B scenārijos izmantotā DOM struktūra nav piemērs sarežģītas struktūras saraksta vai tabulas elementiem – vienas lapas tīmekļa lietotnēs var tikt izmantotas tabulas ar rindām, kuru DOM struktūra ievērojami pārsniedz B scenārijos izmantotās DOM

struktūras piemēru, līdz ar to šādās tabulās neoptimālas tabulas rindu ģenerēšanas gadījumā negatīvā ietekme uz ātrdarbību var būt vēl vairākas reizes ievērojamāka.

Scenārijā A2 100 saraksta elementu renderēšana aizņem 2,42 reizes vairāk laika, nekā A3 scenārijā, savukārt 1000 saraksta elementu renderēšana aizņem 17,47 reizes vairāk laika. Scenārijā B2 100 saraksta elementu renderēšana aizņem 3,36 reizes vairāk laika, nekā B3 scenārijā, savukārt 1000 saraksta elementu renderēšana aizņem 27,49 reizes vairāk laika. No tā var secināt, ka šajā gadījumā lielākas DOM struktūras elementiem veiktspēja pasliktinās nedaudz straujāk, saraksta elementu skaitam pieaugot, kā arī gan atbildes laiku, kas lietotājam ir pamanāms, gan atbildes laiku, kas pārsniedz cilvēka uzmanības intervālu (skat. 2.1 nodaļu), ir iespējams sasniegt ar ievērojami mazāku elementu skaitu.



5.20. att. **B2 scenārija veiktspējas mērījums**

Attēlā (skat. 5.20. att.) ir redzams Google Chrome pārlūkprogrammā veikts ātrdarbības mērījums scenārijam B2, kas parāda detaļas par to, cik laika aizņemts, veicot konkrētas darbības. Redzams, ka zilās iedaļas atspoguļo laiku, kas patērēts HTML koda parsēšanai, kas arī ir atšķirība no scenārija B3, kurā saraksta elementi tiek pievienoti ar *appendChild* metodi. Tas skaidrojams ar *innerHTML* izpildes procesa detaļām. Katru reizi, kad elementam tiek mainīta *innerHTML* vērtība, jaunā vērtība, kas ir simbolu virkne, tiek parsēta par HTML kodu, kas tiek pārveidots par *DocumentFragment* objektu, kas satur visus no parsētā HTML koda izveidotos DOM elementus, un pēc tam elementam, kam mainīta *innerHTML* vērtība, visi DOM elementi, ko tas satur, tiek aizvietoti ar jaunizveidotajiem DOM elementiem [7]. Tā kā mūsdienu pārlūki ir optimizēti sinhrona JavaScript koda izpildē radušās DOM izmaiņas veikt pēc koda izpildes, tad praktiski elementa, kam mainīta *innerHTML* vērtība, A2 un B2 scenārijos lietotāja saskarnē esošais saturs tiek mainīts tikai vienu reizi pēc koda izpildes, taču redzams, ka ar to, ka katra elementa pievienošanas brīdī tiek veikta pilna saraksta simbolu virknes parsēšana par HTML un *DocumentFragment* izveidošana, ir pietiekami, lai radītu lietotājam

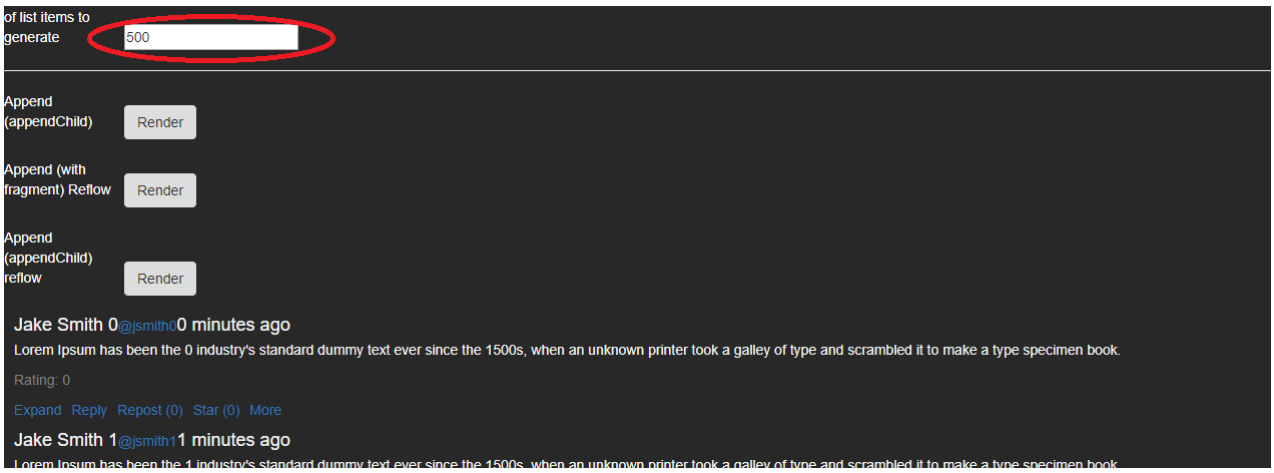
manāmu aizkavēšanos, salīdzinot ar A3 un B3 scenārijiem, kur tiek izmantota *appendChild* metode.

5.5 Piespiedu izkārtošanas novēršana

Nodaļā 5.3 tika izpētītas dažādas pieejas HTML lapas satura papildināšanai, un no praktiskas pārbaudes tika secināts, ka, papildinot sarakstu ar *innerHTML* potenciāli rada ievērojami lielākas veiktspējas problēmas, nekā *appendChild* metode, ja nepieciešams veikt darbību, kas izraisa izkārtošanas procesu, katrā papildināšanas reizē. Taču dažādos literatūras avotos ir aprakstītas metodes, kas palīdz pilnībā izvairīties no izkārtošanas procesa veikšanas katrā papildināšanas reizē [53, 25]:

- Papildināmo elementu atzīmēt ar CSS īpašību *display: none*.
- Veikt papildināmā elementa klonēšanu, veikt visas darbības un aizvietot oriģinālo elementu.
- Izmantot dokumenta fragmentu (document.createDocumentFragment()) un tam pievienot papildināmo saturu, un beigās fragmentu ievietot lietotāja saskarnes DOM

Nodaļā tiek pārbaudīta dokumenta fragmenta izmantošana piespiedu izkārtošanas procesa novēršanai. Pirms pārbaudes tika pārbaudīts, vai fragmenta izmantošana atļauj nolasīt nepieciešamo informāciju, kas izraisa izkārtošanu. Scenārijā B4 *offsetWidth* īpašība tika nomainīta pret *offsetHeight* īpašību, jo tādā veidā katrā iterācijā, kad tiek nolasīta *offsetHeight* īpašība, sagaidāmais rezultāts ir cits saraksta vecāka elementa augstums sakarā ar to, ka katrā nākamajā iterācijā sarakstā tiek ievietots jauns elements. Tika konstatēts, ka šādā veidā nav iespējams nolasīt aktuālo augstumu ar nevienu no literatūras avotos pieejamajiem risinājumiem piespiedu izkārtošanas procesa novēršanai.



5.21. att. **Ar sarkanu krāsu iezīmēts elements, kura augstums tiek nolasīts**

Sakarā ar to tā vietā, lai katrā iterācijā nolasītu saraksta vecāka elementa augstumu, tika nolasīts no saraksta neatkarīga ievades lauka augstums (skat. 5.21. att.), kas arī izraisa piespiedu izkārtošanas procesu katrā saraksta elementa pievienošanas reizē. Scenāriju koda fragmentu attēlos nav parādīts pilnais kods – daudzpunktes vietā kods iztrūkst. Pilnais kods atrodams Github repozitorijā (skat. 11. pielikums). Tiek definēti šādi scenāriji:

C1: Scenārijs atbilst B3 scenārijam (skat. 5.22. att.).

```
for (var i = 0; i < iterations; i++) {
var divInputWrap = document.createElement('div');
.......
aPostPropertyStar.innerText='Star ('+i+')';
div.appendChild(divInputWrap);
}
```

5.22. att. **C1 scenārija koda fragments**

C2: Scenārijs atbilst B4 scenārijam, taču katrā iterācijā tiek nolasīts ievades lauka augstums (skat. 5.23. att.).

```
for (var i = 0; i < iterations; i++) {
var divInputWrap = document.createElement('div');
.......
aPostPropertyStar.innerText='Star ('+i+')';
fragment.appendChild(divInputWrap);
h=document.getElementById("listElementCount").offsetHeight;
}
```

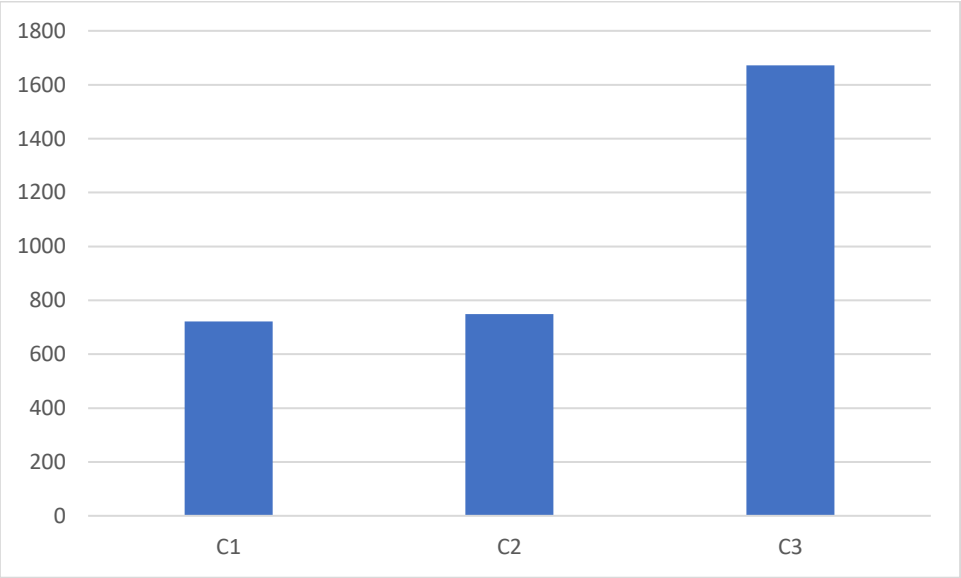
5.23. att. **C2 scenārija koda fragments**

C3: Scenārijs atbilst B3 scenārijam, taču katrā iterācijā tiek nolasīts ievades lauka augstums (skat. 5.24. att.).

```
for (var i = 0; i < iterations; i++) {
var divInputWrap = document.createElement('div');
.......
aPostPropertyStar.innerText='Star ('+i+')';
div.appendChild(divInputWrap);
h=document.getElementById("listElementCount").offsetHeight;
}
```

5.24. att. **C3 scenārija koda fragments**

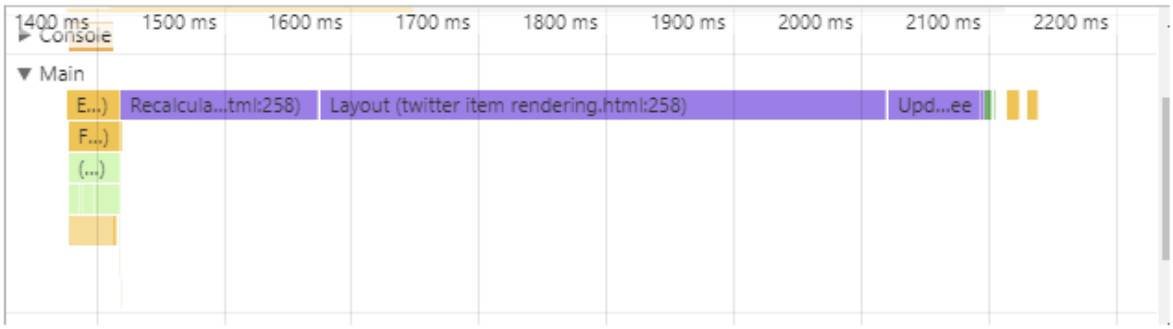
Tā kā šo scenāriju mērķis ir noteikt to, vai notiek izvairīšanās no piespiedu izkārtošanas procesa, mērījumi tiek veikti 500 elementiem, un tālāki eksperimenti netiek veikti, jo nav nepieciešams noteikt sakarību starp elementu skaitu.



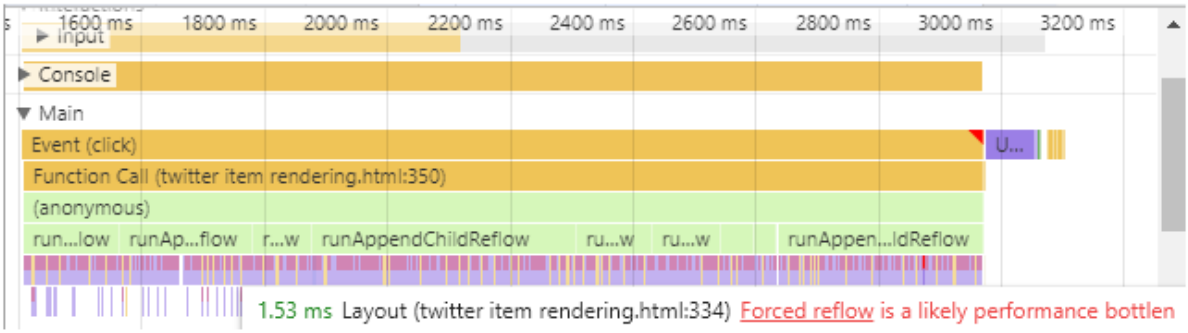
5.25. att. **Saraksta elementu renderēšanai patērētais laiks atkarībā no elementu skaita scenārijos**

Redzams, ka scenārija C2 rezultāts ir tuvs C1 rezultātam, kas liecina par to, ka dokumenta fragmenta veidošana šajā gadījumā ir nodrošinājusi to, ka netiek izsaukta piespiedu izkārtošana, nolasot ievades lauka augstumu.

C2



C3



5.26. att. **Scenāriju veiktspējas ieraksts Google Chrome izstrādātāja rīkos**

Attēlā C2 scenārijam zem *Event (click)* izsaukuma nav redzami violeti taisnstūri, kas apstiprina to, ka piespiedu izkārtošana nav veikta. C3 scenārijā zem *Event (click)* izsaukuma redzams liels daudzums violetu taisnstūru, kas nozīmē, ka ir vairākkārt veikts izkārtošanas process pat gadījumā, kad elements, kam nolasīts augstums ar *offsetHeight*, ir nesaistīts ar sarakstu.

Šādā veidā tika pārbaudīta arī lietotāja saskarnē esoša elementa paslēpšana ar display:none pirms papildināšanas ar elementiem, kā arī lietotāja saskarnē esoša elementa klonēšana, klonētā elementa papildināšana un oriģinālā elementa aizstāšana. Visas šīs pieejas rāda līdzīgus rezultātus.

Pēc izpētes var secināt, ka ar literatūras avotos atrodamajām metodēm ir iespējams izvairīties no piespiedu izkārtošanas procesa, taču tas noder tikai gadījumos, kad ir jānolasa kāda vērtība, kas nav saistīta ar veiktajām manipulācijām. Piemēram, gadījumā, ja ar cikla palīdzību tiek veidots saraksts, un ir nepieciešams aktuālais augstums katrā iterācijā kopā ar jaunizveidoto elementu, to nav iespējams nolasīt, jo lietotāja saskarnē esošie elementi netiek fiziski izmainīti. Tāpēc šādos gadījumos ir svarīgi izvairīties no *innerHTML* metodes un izmantot, piemēram, *appendChild* vai *insertAdjacentHTML*, kas neaizstāj visu elementa saturu, katru reizi to papildinot.

6. ANIMĀCIJU VEIKTSPĒJA

Animācijas ir no statiskiem objektiem veidots vizuāls kustības efekts, ko rada virkne nelielu, pakāpenisku kustību. Vienas lapas tīmekļa lietotnēs tās izmanto dažādiem nolūkiem – kāda ģeometriska objekta, ar ko veic vizuālas manipulācijas, konkrēta efekta iegūšanai, piemēram, kāda lietotāja saskarnes HTML elementa aizslīdēšanai uz sānu, līdz lietotājam tas vairs nav redzams. Pastāv dažādi veidi, kā veidot animācijas, un darbā daļa no tiem tiek izskatīta, kā arī salīdzināta savā starpā ar veiktspējas mērījumiem.

6.1 Elementu ģeometriju neietekmējošas CSS īpašības

Ir dažādas CSS īpašības, ar ko veikt animācijas tīmekļa lietotnēs. Pastāv dažādas animācijām pielietojamas CSS īpašības, kas ietekmē objekta ģeometriju, līdz ar to izsauc renderēšanas izkārtošanas, zīmēšanas un kompozitēšanas posmus (skat. *6.1.* att.) (posmu aprakstu skat. 4.1 apakšnodaļā).

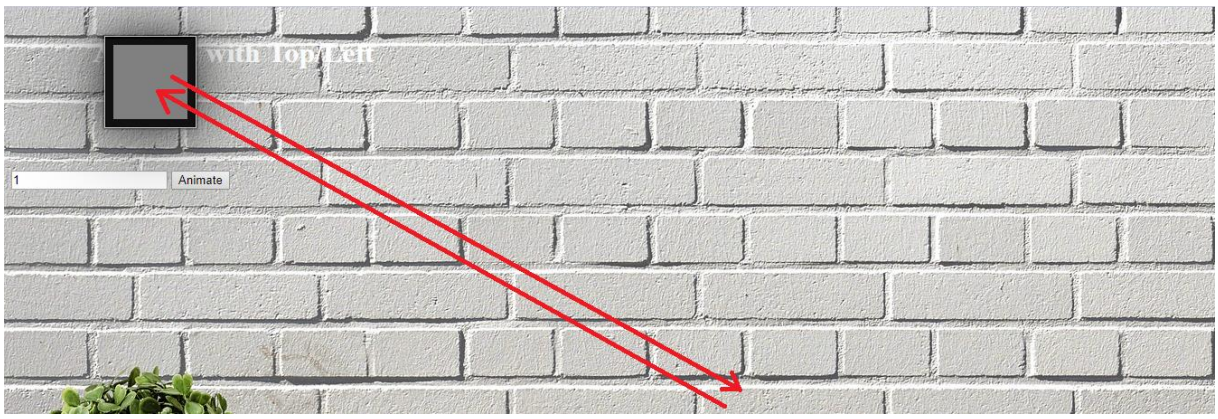
6.1. att. **Ģeometriju ietekmējošu īpašību saskarnes izmaiņu process** [57]

Kā piemērs šādai īpašībai, ko var izmantot animācijā, ir *top, left, right, bottom* CSS īpašības, kas saistītas ar *position: absolute* [56].

Taču eksistē arī tādas animācijām pielietojamas CSS īpašības, kas neizsauc izkārtošanas un zīmēšanas procesus – tikai kompozitēšanas procesu (skat. 6.2. att.) [55]. Piemērs šādai CSS īpašībai ir *transform: translate*, kas animāciju veiks citā slānī, izmantojot video kartes jaudu [62]. Tas, ka animācija tiek veikta citā slānī, nozīmē to, ka animējamā objekta pozīcija netiks aprēķināta attiecībā pret citiem objektiem, līdz ar to tiek veikts tikai kompozitēšanas posms [53]. Tas nozīmē, ka, veicot sarežģītas animācijas, šīs īpašības izmantojot, pastāv iespēja, ka šo animāciju veiktspēja būs redzami labāka.

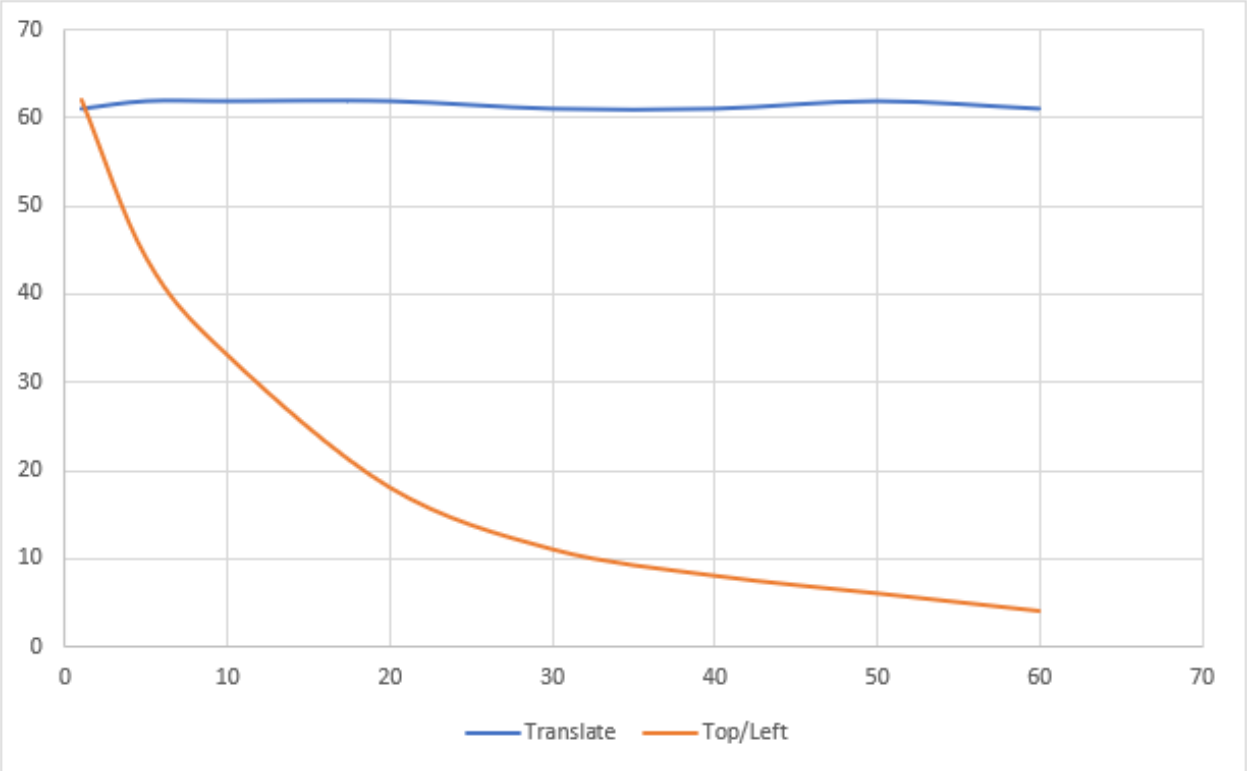
6.2. att. **Tikai kompozitēšanu izsaucošu īpašību saskarnes izmaiņu process** *[55]*

Lai noskaidrotu atšķirību starp šīm dažādajām pieejām animāciju veidošanā, tika izvēlēta tīmeklī atrodama lietotne šo pieeju salīdzināšanai [38], un tā tika vienkāršota, lai būtu skaidrāk redzamas atšķirības katrai pieejai, kā arī kods būtu pārskatāmāks un būtu redzams, ka cits kods šo pieeju darbību neietekmē (skat. 12. pielikums un 13. pielikums).



6.3. att. **Animācijas testa lietotne**

Testa lietotnē realizēta animācija, kas *div* elementu ar ēnojumu pakāpeniski pārvieto slīpi no lapas augšpuses uz apakšpusi un atpakaļ (skat. 6.3. att.). Tā kā šī animācija ir salīdzinoši vienkārša, tika veikti testi arī ar vairākiem šādiem pārvietojamiem *div* elementiem vienlaikus, lai redzētu, kā tas ietekmē pārlūkprogrammas attēlotos kadrus sekundē.



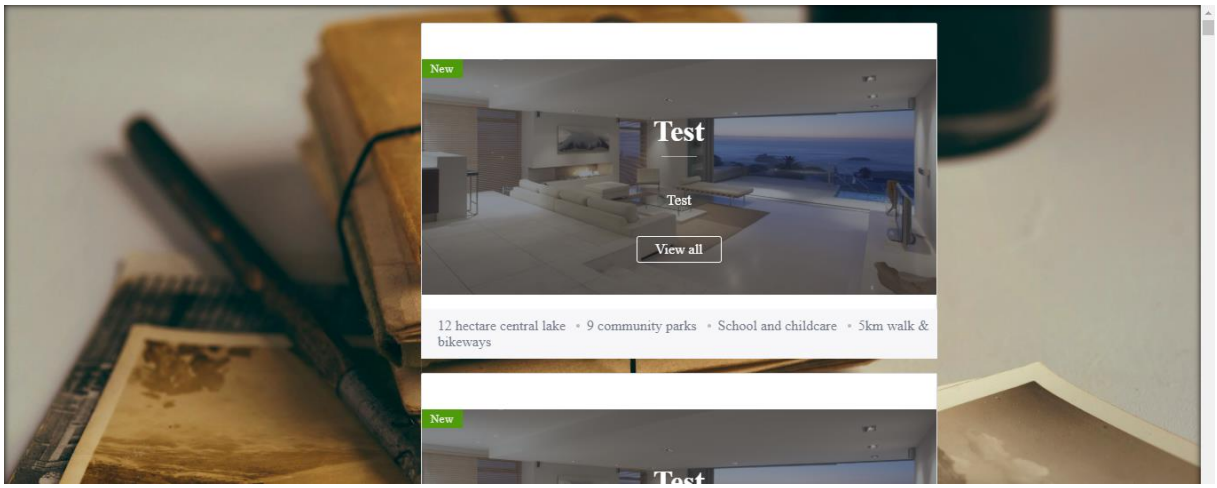
6.4. att. **Kadru skaits sekundē atkarībā no animēto elementu skaita**

Attēlā (skat. 6.4. att.) uz x ass attēlots pārvietojamo div elementu skaits, taču uz y ass – attēloto kadru skaits sekundē. Attēloto kadru mērījumi tika veikti, izmantojot pārlūkprogrammas *Google Chrome* izstrādātāja rīkus, kas attēlo laiku, cik patērēts katra kadra attēlošanai, kā arī to, cik kadrus sekundē var attēlot ar konkrēto kadra attēlošanai patērēto laiku. No grafikā redzamajiem datiem var secināt, ka vienkāršu animāciju, piemēram, sāna navigācijas izslīdēšanai, var lietot animāciju, kas veidota ar *top, left, bottom, right* CSS īpašībām, bez kadru skaita sekundē samazināšanās. Savukārt animācijām, kas ietver vairāku elementu pakāpenisku pārvietošanu, vai cita veida sarežģītām animācijām vēlams izmantot CSS īpašības, kas neizsauc renderēšanas izkārtošanas un zīmēšanas procesus.

6.2 Izvietošana atsevišķos slāņos

Tādas darbības kā animācijas vai satura ritināšana var radīt situācijas, kad katrā kadrā jāveic kāda elementa pārzīmēšana attiecībā pret citiem [59]. Piemēram, ja tīmekļa lietotnei eksistē galvene ar CSS īpašību *position: fixed*, tad, veicot lapas satura ritināšanu, šī galvene tiek pārzīmēta, jo ritināšanas rezultātā galvene atrodas citā vietā attiecībā pret pārējiem lapas elementiem, kas nav fiksēti, un galvene atrodas vienā slānī ar pārējiem elementiem [60]. No tā var secināt, ka eksistē situācijas, kurās jauna slāņa izveidošana var uzlabot ritināšanas procesā attēloto kadru skaitu sekundē. Jaunu slāni iespējams izveidot, izmantojot CSS *will-change* īpašību [58, 54]. Pārlūkiem, kas neatbalsta *will-change* īpašību, iespējams izmantot *transform: translateZ(0)* [58, 54].

Jauna slāņa izveides ietekmes pārbaudei tika izveidota testa lietotne (skat. 14. pielikums un 15. pielikums), kas attēlo sarakstu, un šai lietotnei kā fons ir attēls ar CSS *position:fixed* īpašību un iekšēju ēnojumu (skat. 6.5. att.).

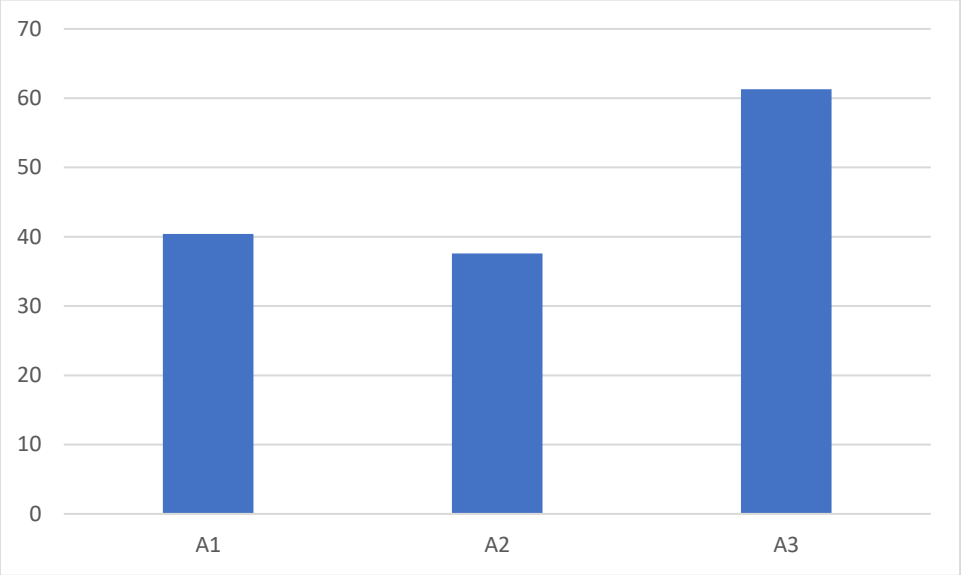


6.5. att. **Ritināšanas testa lietotne**

Tika veikti ritināšanas ātrdarbības mērījumi. Lai ritinātu saturu, tika turēts nospiests klaviatūras taustiņš ar bultiņu uz leju. Mērījumi tika veikti šādiem scenārijiem:

- A1: Fons veidots kā fiksēts fona attēls *div* elementam, kas ir vecāks visam lapas saturam un aizņem visu ekrāna platumu un visu ekrāna garumu.
- A2: Fons veidots kā fiksēts pseido elements (*::before*), kas aizņem visu ekrāna platumu un garumu, lapā esošam DOM elementam.

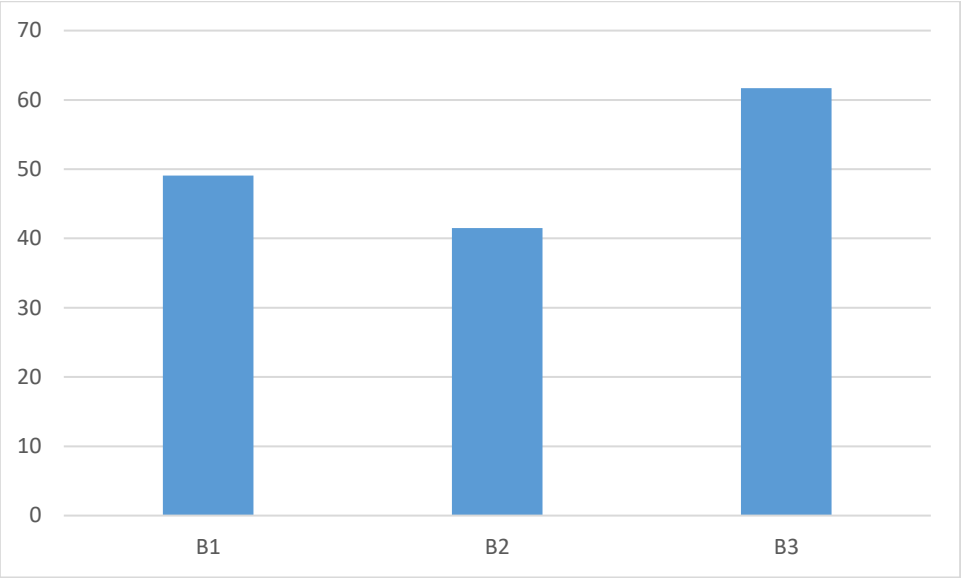
- A3: Fons veidots kā fiksēts pseido elements (*::before*), kas aizņem visu ekrāna platumu un garumu, lapā esošam DOM elementam, un šis pseido elements tiek izvietots jaunā slānī, izmantojot *will-change:transform*.



6.6. att. **Kadru skaits sekundē dažādos scenārijos**

Rezultātos (skat. 6.6. att.) redzams, ka šādā gadījumā iespējams optimizēt ritināšanas veiktspēju. Gan A1, gan A2 gadījumos ir vizuāli redzams, ka tiek attēlots mazāk kadru sekundē. Tā kā A1 gadījumā fona attēls ir uzstādīts lapas vecāka elementam, A3 scenārijā bija nepieciešams šo fonu uzstādīt pseido elementam, lai būtu atsevišķs elements, ko ievietot jaunā slānī. Scenārijs A2 tika izveidots, lai redzētu tiešo uzlabojumu identiskiem scenārijiem, kuru vienīgā atšķirība ir *will-change* īpašības izmantošana.

Tika izveidoti arī scenāriji B1, B2 un B3, kas atbilst attiecīgajiem A scenārijiem, taču šajos scenārijos fona attēlam netika izmantota iekšējā ēna – šo testa scenāriju mērķis ir pārbaudīt, kā CSS īpašības, kas veic vizuālas izmaiņas elementā, ietekmē ritināšanas veiktspēju.



6.7. att. **Kadru skaits sekundē dažādos scenārijos bez ēnojuma**

Redzams, ka, noņemot ēnu no fiksētā fona attēla, ir uzlabojies kadru skaits sekundē scenārijos B1 un B2 (skat. 6.7. att.). Scenārijā B3 novērojams minimāls uzlabojums, taču tam ir maza nozīme, jo gan A3, gan B3 scenārijos tiek pārsniegti 60 kadri sekundē.

Neskatoties uz šiem rezultātiem, ir jāpiemin tas, ka *will-change* CSS īpašību vēlams izmantot tikai gadījumos, kad ir manāmas veiktspējas problēmas. Ja šo īpašību pielieto visiem ar animācijām un zīmēšanas procesu saistītajiem elementiem, tiek radīts liels daudzums slāņu, kas jāielādē videokartē kā tekstūras, un tas aizņem atmiņu [58, 62, 54].

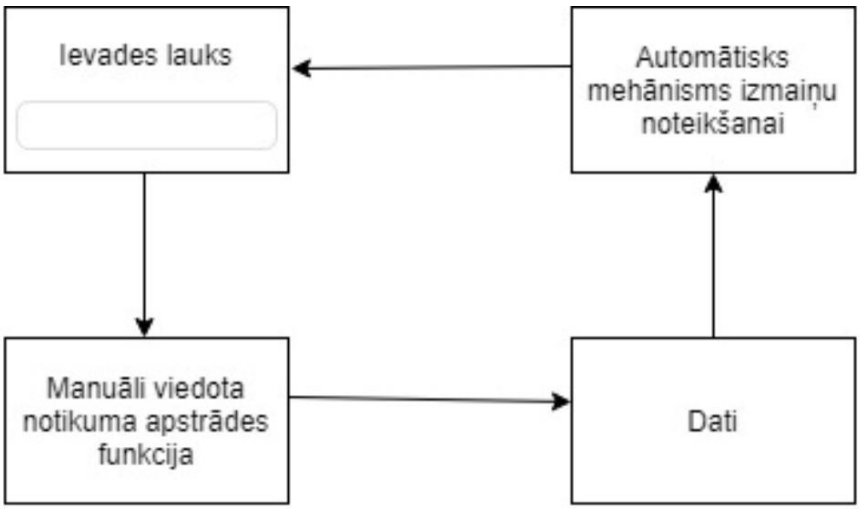
7. DATU PIESAISTE

Parasti vienas lapas tīmekļa lietotnēs ir dati, kas tiek attēloti lietotāja saskarnē DOM elementos. Dati parasti tiek glabāti JSON formātā. Datu piesaistes mērķis ir veikt šo datu atspoguļošanu lietotāja saskarnē tādā formā, kā programmētājs to ir definējis. Piemēram, ja dati glabā JSON masīvu ar objektiem, kas satur informāciju, un šie dati ir piesaistīti sarakstam, tad saraksts ģenerē saraksta elementus, kas attēlo daļu katra šī masīva elementa informācijas (skat. 7.1. att.).

7.1. att. **Datu attēlošana lietotāja saskarnē**

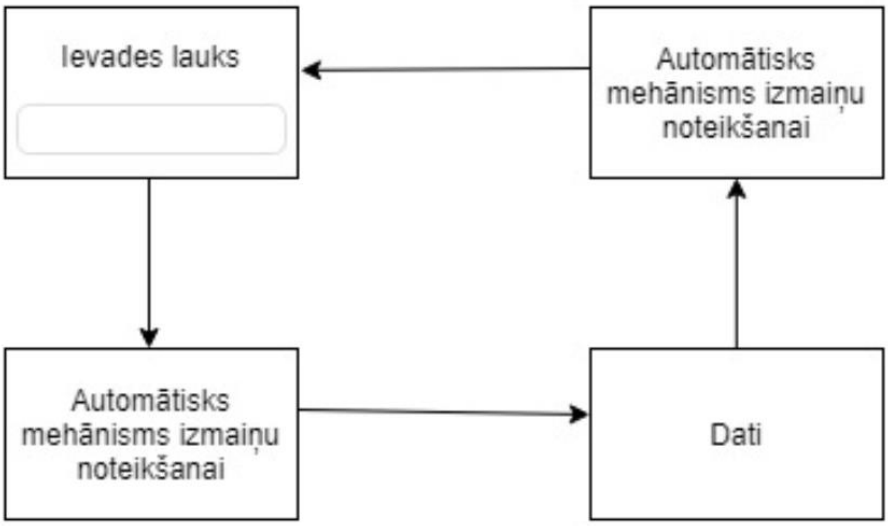
Ja datos tiek nomainīts kāds atribūts, piemēram, attēlā (skat. 7.1. att.) dotajā piemērā tiek nomainīts pirmā elementa *favorites* atribūts, datu piesaiste nodrošina šīs izmaiņas atspoguļošanu lietotāja saskarnē – atbilstošā saraksta elementa atbilstošais teksts tiek nomainīts atbilstoši datiem. Datu piesaisti var implementēt manuāli, izmantojot JavaScript programmēšanas valodu vai *jQuery* bibliotēku, vai arī izmantot kādu JavaScript ietvaru, kas ietver jau iekļautu datu piesaistes mehānismu. Manuāli (neizmantojot ietvarus) veicot lietotāja saskarnes manipulācijas, ir svarīgi tas, ka nepieciešams veikt minimālās nepieciešamās izmaiņas lietotāja saskarnē, jo lietotāja saskarnes elementu izmainīšana ir laikietilpīgs process, ja, piemēram, daļai datu izmainoties, tiek vienkārši pārģenerēts viss saraksts, balstoties uz izmainītajiem datiem.

Datu piesaiste var būt viena virziena vai divu virzienu, kā arī vienas reizes. Viena virziena datu piesaiste nozīmē to, ka, datiem izmainoties, tie tiks automātiski izmainīti lietotāja saskarnē, taču, ja vajadzīga datu izmainīšana no lietotāja saskarnes, tad tas nenotiek automātiski – tam nepieciešams realizēt atsevišķu funkciju, piemēram, definēt teksta ievadlaukam teksta ievades notikuma apstrādes funkciju, kas manuāli izmaina datus (skat. 7.2. att.) [17].



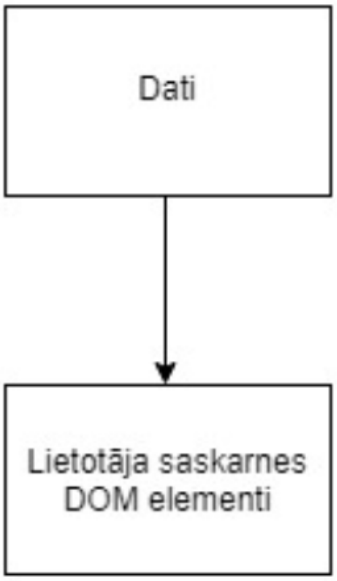
7.2. att. **Viena virziena datu piesaiste**

Divu virzienu datu piesaiste nozīmē to, ka, datiem izmainoties, tie tiks automātiski izmainīti lietotāja saskarnē, kā arī, piemēram, teksta ievades lauks, kas piesaistīts datiem ar divu virzienu datu piesaisti, automātiski izmaina atbilstošos datus, kad tajā tiek rakstīts teksts (skat. 7.3. att.) [17].



7.3. att. **Divu virzienu datu piesaiste**

Vienas reizes datu piesaiste nozīmē to, ka lietotāja saskarnes DOM elementi tiek ģenerēti tikai vienu reizi no modeļa programmētāja definētajā veidā (skat. 7.4. att.).

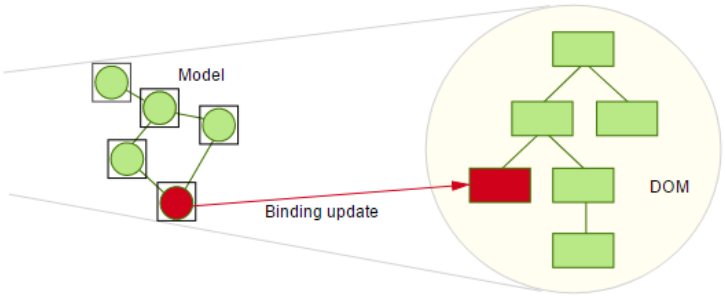


7.4. att. **Vienas reizes datu piesaiste**

Šādas datu piesaistes gadījumā izmaiņas datu modelī netiek automātiski attēlotas lietotāja saskarnē, kā arī izmaiņas lietotāja saskarnes elementos, piemēram, ievades lauka teksta izmaiņas neizmainīs datu modeli.

7.1 Novērojamie objekti

Viena no datu piesaistes tehnikām, ko izmanto vienas lapas tīmekļa lietotņu izstrādes ietvaros, ir novērojamie objekti [23]. Šāda datu piesaistes metode tiek izmantota Knockout JS ietvarā, un kā vēl vienu piemēru šādai datu piesaistes tehnikai var minēt Ember JS ietvaru. Šī datu piesaistes tehnika atbalsta divu virzienu datu piesaisti. Šī datu piesaistes tehnika balstās uz to, ka novērojamie objekti var paziņot citiem objektiem par to, ka ir izmainījušies, un kas ir izmainījies (skat. 7.5. att.) [48]. Tas nodrošina to, ka, piemēram, teksts, kas piesaistīts HTML *p* elementam skatā, var veikt attiecīgās izmaiņas, lai skats un dati būtu nosinhronizēti [18, 27].



7.5. att. **Novērojamie objekti** *[45]*

Tas realizēts ar ziņošanas mehānismu, kas nozīmē, ka konkrēti objekti var sūtīt ziņas objektiem, kas ir pieteikušies tās saņemt. Šis ir mehānisms, kas Knockout JS ietvarā ļauj skatam automātiski atjaunoties atkarībā no izmaiņām skatmodelī, kas sastāv no novērojamajiem objektiem.

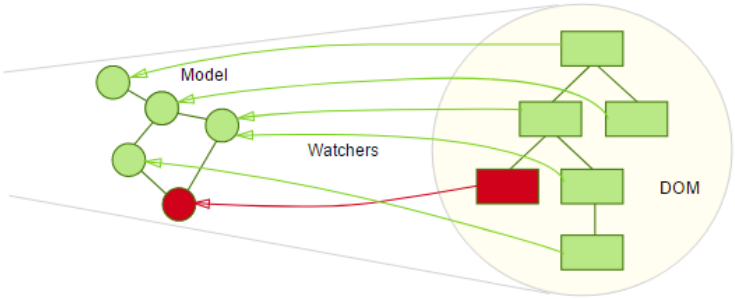
```
The name is <span data-bind="text: personName"></span>
```

7.6. att. **Span elementam piesaistīti modeļa dati**

Šis mehānisms paredz, ka dati, kas tiek piesaistīti skatam, tiek pārveidoti no parastiem JavaScript objektiem par novērojamajiem objektiem, lai tie būtu spējīgi paziņot lietotāja saskarnes piesaistītajiem elementiem par to, ka ir notikušas izmaiņas. Attēlā (skat. 7.6. att.) redzamajā kodā *personName* atribūts ir piesaistīts atbilstošam novērojamajam objektam, kas nozīmē, ka šajā *span* elementā tiks parādīts teksts, kas datos ir *personName* atribūtā, kā arī šis *span* elements ir reģistrējis sevi tam, lai saņemtu paziņojumus par to, ka *personName* atribūts ir izmainījies [18].

7.2 Netīro datu pārbaude

Vēl viena datu piesaistes metode ir netīro datu pārbaude. Šāda datu piesaistes tehnika tiek izmantota Angular JS ietvarā un kā vēl vienu ietvara piemēru, kam datu piesaiste darbojas līdzīgi, var minēt Angular ietvaru [63]. Šī datu piesaistes tehnika atbalsta divu virzienu datu piesaisti. Modeļa vērtībām, kas tiek piesaistītas kādam lapā attēlotajam HTML elementam, tiek izveidoti vērotāji (*watcher*). Kad lietotnē notiek kāda darbība, kas var izsaukt izmaiņas datos, tiek veikta netīro datu pārbaude, kas salīdzina katra vērotāja iepriekšējās un tagadējās vērtības (skat. 7.7. att.). Ja vērotāja iepriekšējā un tagadējā vērtība atšķiras, tiek veikta konkrētajam HTML elementam piesaistītās vērtības atjaunināšana lietotāja saskarnē. Šo procesu mēdz dēvēt arī par sagremošanas ciklu (*digest loop*), jo Angular JS ietvarā to var izsaukt arī manuāli, izmantojot funkciju *$scope.$digest()* [49, 19, 45].



7.7. att. **Netīro datu pārbaude** *[45]*

Attēlā (skat. 7.8.att.) redzamajā kodā parādīts Angular JS skata daļas piemērs, kurā vairāki atribūti no modeļa piesaistīti HTML DOM elementiem, izmantojot dubultas figūriekavas.

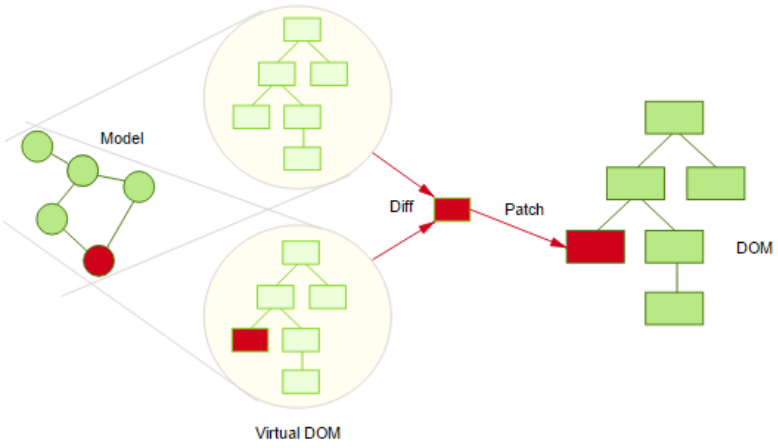
```
<div class="media-body">
   <h4 class="post-heading">{{firstName}} {{lastName}}  <a>@{{firstName}}</a> 4 minutes ago</h4>
   <p class = "post-body" >{{text}}</p>
   <p class = "post-rating" >Rating {{favorites}}</p>
</div>
```

7.8.att. **Skatam piesaistīti dati** *[18]*

Katram atribūtam, kas skatā piesaistīts ar figūriekavām, tiek izveidots vērotājs, lai netīro datu pārbaudes ciklā pārbaudītu to, kuras vērtības izmainītas. Pateicoties šim mehānismam, Angular JS modeļi ir tīri JavaScript objekti, kas nav jāpārveido par novērojamajiem objektiem vai kādā citā veidā [49].

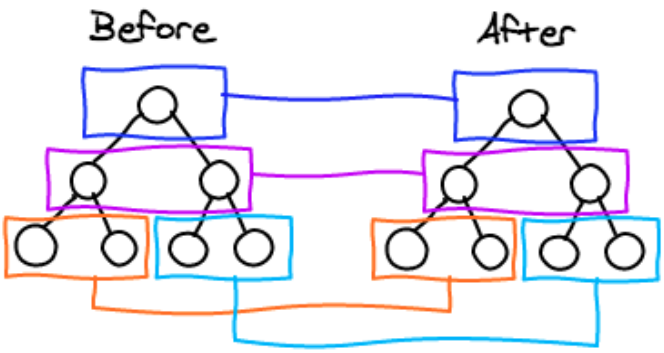
7.3 DOM salīdzināšana

DOM salīdzināšanas datu piesaistes mehānisms strādā, vispirms izveidojot DOM JavaScript objektu reprezentāciju, un tikai pēc tam ģenerējot DOM attēlojamajā lapā. Šāds datu piesaistes mehānisms darbojas React JS ietvarā, un tas atbalsta viena virziena datu piesaisti. Kad kādai komponentei tiek piesaistīti jauni dati vai tiek veiktas izmaiņas tās datos, tiek veikta šīs komponentes ģenerēšana JavaScript DOM reprezentācijā. Atmiņā tiek glabāta šīs komponentes iepriekšējā JavaScript DOM reprezentācija, un tā tiek salīdzināta ar JavaScript DOM reprezentāciju, kas ģenerēta izmaiņu rezultātā. Pēc šo JavaScript DOM reprezentāciju salīdzināšanas tiek konstatēts, kas lapā attēlotajā DOM ir jāizmaina, lai skats un dati būtu nosinhronizēti (skat. 7.9. att.) [48, 45, 20].



7.9. att. **DOM salīdzināšana** *[45]*

Veicot iepriekšējās un šī brīža komponentes JavaScript DOM reprezentāciju salīdzināšanu, šīs reprezentācijas tiek salīdzinātas pa līmeņiem (skat. 7.10. att.), nevis meklēta absolūti minimālā atšķirība starp abām koka struktūrām [46].



7.10. att. **Salīdzināšana pa līmeņiem** *[46]*

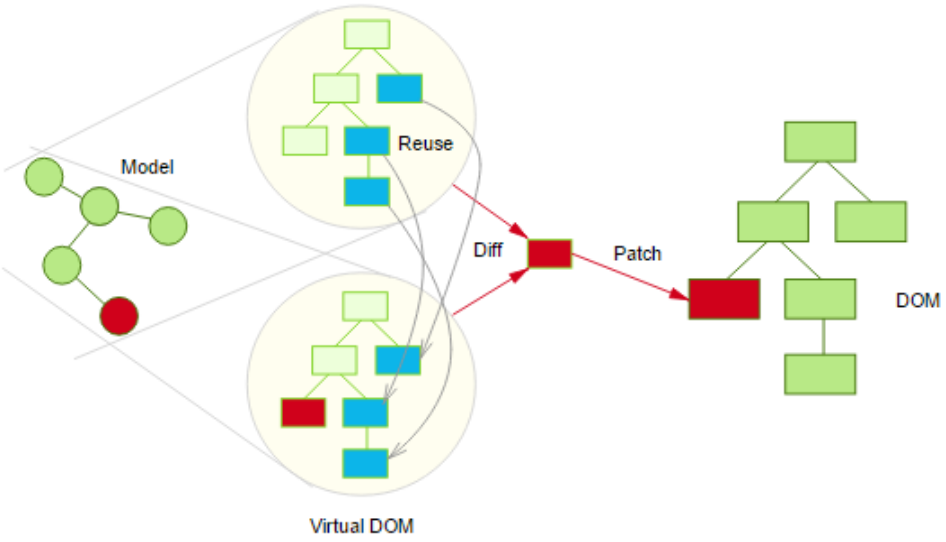
Salīdzinot JavaScript DOM reprezentācijas šādā veidā algoritma sarežģītība ir mazāka, un izmaiņas tiek noteiktas pietiekami precīzi, jo gadījumi, kad nepieciešams pārvietot kādu komponenti vai HTML elementu citā līmenī DOM kokā, rodas ļoti reti [46].

8. VEIKTSPĒJAS UZLABOŠANA UN ĪPATNĪBAS VIENAS LAPAS TĪMEKĻA LIETOTŅU IZSTRĀDES IETVAROS

Liela apjoma vienas lapas tīmekļa lietotņu izstrādē tiek plaši izmantoti JavaScript izstrādes ietvari, lai izstrādes laiks būtu mazāks un pirmkods būtu organizētāks un uzturamāks. Ir svarīgi aprakstīt, kā iespējams panākt veiktspējas uzlabošanu vienas lapas tīmekļa lietotnēs, kas izstrādātas, tos izmantojot. Viens no faktoriem, kas ietekmē ar JavaScript ietvaru izstrādātas apjomīgas vienas lapas tīmekļa lietotnes veiktspēju, ir tas, kā ir realizēta datu piesaiste un tās izmaiņu noteikšanas mehānisms – tas, kā ietvari apstrādā apjomīgas komponentes, skatus, lielas tabulas un sarakstus, kam piesaistīti modeļa dati. Testiem tika izmantotas pētīto JavaScript ietvaru produkcijas versijas, nevis versijas ar nesaīsinātu kodu un kļūdu paziņojumu paskaidrojumiem, jo tas uzlabo veiktspēju. Testu mērķis nav salīdzināt to, kura datu piesaistes tehnika darbojas visātrāk, bet gan vērst uzmanību uz vērā ņemamajiem faktoriem, izstrādājot vienas lapas tīmekļa lietotnes šajos ietvaros. Ierobežotā laika dēļ šajā nodaļā veiktspējas uzlabošana tiek izskatīta ļoti virspusēji un iekļauj tikai pamatus, kas katrā ietvarā jāņem vērā, lai vērstu uzmanību uz to, cik svarīgi ir izprast ietvara, kurā lietotne tiek veidota, datu piesaistes metodi. Katrā ietvarā eksistē liels skaits iespējamo veiktspējas uzlabojumu, un visu iespējamo uzlabojumu izpētei būtu nepieciešams ļoti apjomīgs ieguldījums.

8.1 React JS

React JS ietvarā ir iespējams izvairīties no liekas JavaScript DOM reprezentāciju ģenerēšanas un salīdzināšanas (skat. 8.1. att.), izmantojot *shouldComponentUpdate* metodi (skat. 16. pielikums).



8.1. att. **Izvairīšanās no JavaScript reprezentāciju ģenerēšanas** *[46]*

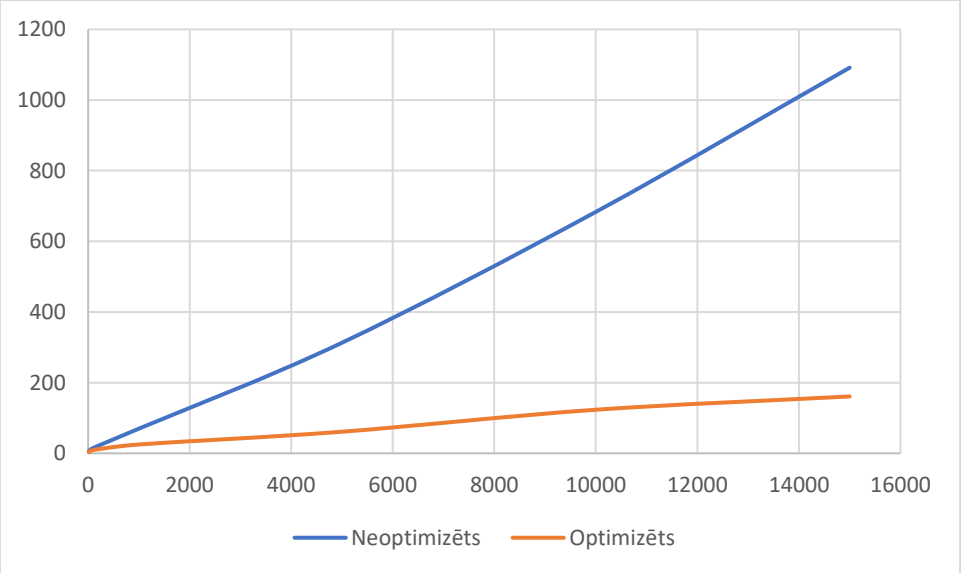
Pielietojot šo metodi, ir iespējams veikt komponentes, kuras datos notikušas izmaiņas, iepriekšējo datu salīdzināšanu ar tagadējiem datiem. Gadījumā, kad netiek konstatētas izmaiņas komponentei piesaistītajos datos, netiek ģenerēta jauna komponentes JavaScript DOM reprezentācija, lai salīdzinātu ar iepriekšējo [47, 46].

Šī nodaļa paredzēta tam, lai ar praktiskiem eksperimentiem parādītu to, cik svarīga var būt šīs metodes ieviešana ar React JS veidotu vienas lapas tīmekļa lietotņu komponentēs.

```
▼<div class="row input-wrap">
  ::before
 ▼<div class="media">
  ▼<div class="body-container">
   ▼<div class="row">
     ::before
    ▼<div class="media-body">
     ▼<h4 class="post-heading">
        <span>Jake Smith0</span>
        <a>@jsmith0</a>
        <span>0 minutes ago</span>
      </h4>
     ▼<p class="post-body">
        "Lorem Ipsum has been the 0 industry's standard dummy text ever since the 1500s, when an
        unknown printer took a galley of type and scrambled it to make a type specimen book."
      </p>
      <p class="post-rating">Rating: 0</p>
     </div>
     ::after
    </div>
   ▼<div class="row">
     ::before
    ▼<div class="bottom-links">
      <a class="post-property" href="#">Expand</a>
      <a class="post-property" href="#">Reply</a>
      <a class="post-property" href="#">Repost (0)</a>
      <a class="post-property" href="#">Star (0)</a>
      <a class="post-property" href="#">More</a>
     </div>
     ::after
    </div>
   </div>
  </div>
  ::after
</div>
```

8.2. att. **Saraksta elementu struktūra**

Tika izveidota testa lietotne ar elementu sarakstu, kura elementu struktūra ir redzama attēlā (skat. 8.2. att.) un veikti ātrdarbības mērījumi viena saraksta izmainīšanai. Tika veikti testi ar 10, 100, 1000, 5000, 10000, 15000 saraksta elementiem. Šāds tests tika izvēlēts tāpēc, ka šāds scenārijs React JS datu piesaistei intuitīvi var radīt problēmas, jo, nomainot vienu saraksta elementu, jāpārģenerē viss saraksts JavaScript reprezentācijā un jāsalīdzina ar iepriekšējo tikai tāpēc, lai atrastu vienu izmaiņu vienā saraksta elementā.



8.3. att. **Patērētais laiks atkarībā no elementu skaita**

Attēlā (skat. 8.3. att.) uz x ass attēlots saraksta elementu skaits un uz y ass attēlots laiks milisekundēs viena masīva elementa izmainīšanas rezultātā izsauktajai modeļa un skata sinhronizācijai. Redzams, ka portatīvajā datorā ar minētajiem parametriem viena elementa nomainīšanas laiks aug ievērojami straujāk, saraksta elementu skaitam pieaugot, tāpēc ir ļoti svarīgi lietot *shouldComponentUpdate* metodi veidojamās vienas lapas tīmekļa lietotnes sarakstos un apjomīgās komponentēs. Šajā gadījumā *shouldComponentUpdate* metode ļauj katram saraksta elementam pārbaudīt, vai tā dati ir izmainījušies, pirms tiek veikta JavaScript DOM reprezentāciju salīdzināšana.

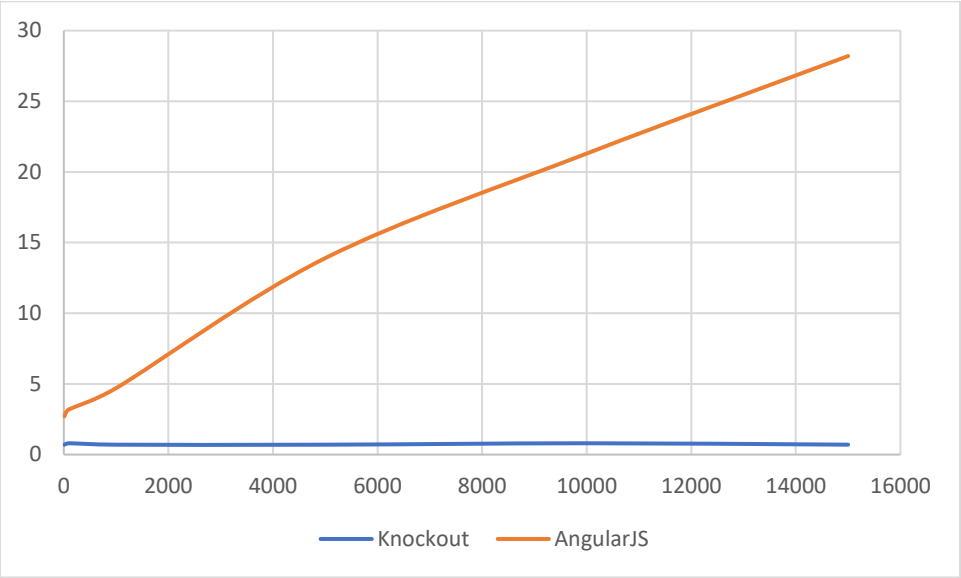
8.2 Angular JS

Pēc teorijas apskates dažādiem JavaScript ietvaru datu piesaistes mehānismiem, var secināt, ka, pievienojot viena skata *$scope* objektam lielu skaitu vērotāju, katra modeļa un skata sinhronizācija aizņems ievērojamu daudzumu laika [19]. Lai redzētu to, cik lielā mērā šis faktors ietekmē veiktspēju, tika veikti testi sarakstam ar dažādiem skaitiem elementu (skat. 18. pielikums). Ņemot vērā to, ka Knockout JS lietotāja saskarnes elementi un modeļa elementi ir sasaistīti ar ziņošanas mehānismu, intuitīvi viena elementa izmainīšanai vajadzētu aizņemt vienādu laika daudzumu neatkarīgi no tā, cik elementu ir sarakstā vai skatā, tāpēc salīdzināšanai tika izvēlēts šis ietvars. Šim testam tika veidots saraksts ar vienkāršu DOM struktūru, piesaistot astoņus atribūtus no objektiem, no kuriem sastāv masīvs.

```
<div class="row input-wrap">
<p>{{post.firstName}}</p><p>{{post.lastName}}</p><p>{{post.text}}</p><p>({{post.favorites}})</p>
<p>{{post.reposts}}</p><p >{{post.image}}</p><p >{{post.age}}</p><p >{{post.id}}</p>
</div>
```

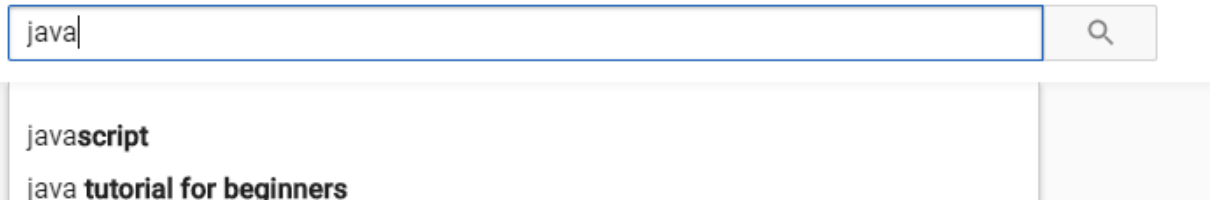
8.4. att. **Saraksta elementu struktūra**

Attēlā (skat. 8.4. att.) redzamajā kodā redzams, ka katram saraksta elementam piesaistītas astoņas vērtības, kas nozīmē, ka katrs masīva elements ģenerē astoņus vērotājus, kuru iepriekšējās un esošās vērtības jāsalīdzina pie katras izmaiņas.



8.5. att. **Patērētais laiks atkarībā no elementu skaita**

Attēlā (skat. 8.5. att.) redzamā diagramma parāda, ka Angular JS katrai veicamajai modeļa-skata sinhronizācijas darbībai patērētais laiks palielinās, pievienojot *$scope* objektam vairāk vērotāju. Diagrammā uz x ass attēlots saraksta elementu skaits un uz y ass attēlots laiks, kas patērēts viena elementa *id* izmaiņas noteikšanai milisekundēs. Šis laiks iekļauj tikai izmaiņas noteikšanai aizņemto laiku, un neiekļauj izmaiņas renderēšanas laiku, jo tas ir aptuveni vienāds, un šādā veidā ir skaidrāk redzama atšķirība. Neskatoties uz to, ka, intuitīvi spriežot, katra vērotāja iepriekšējās un esošās vērtības salīdzināšana 15000 saraksta elementu gadījumā ir laikietilpīgs process, rezultāti parāda to, ka tas aizņem maznozīmīgu daudzumu laika. Taču jāņem vērā, ka ātrdarbības mērījumi ir veikti salīdzinoši jaudīgā datorā, un, piemēram, planšetdatorā vai viedtālrunī šis laiks var būt lielāks.



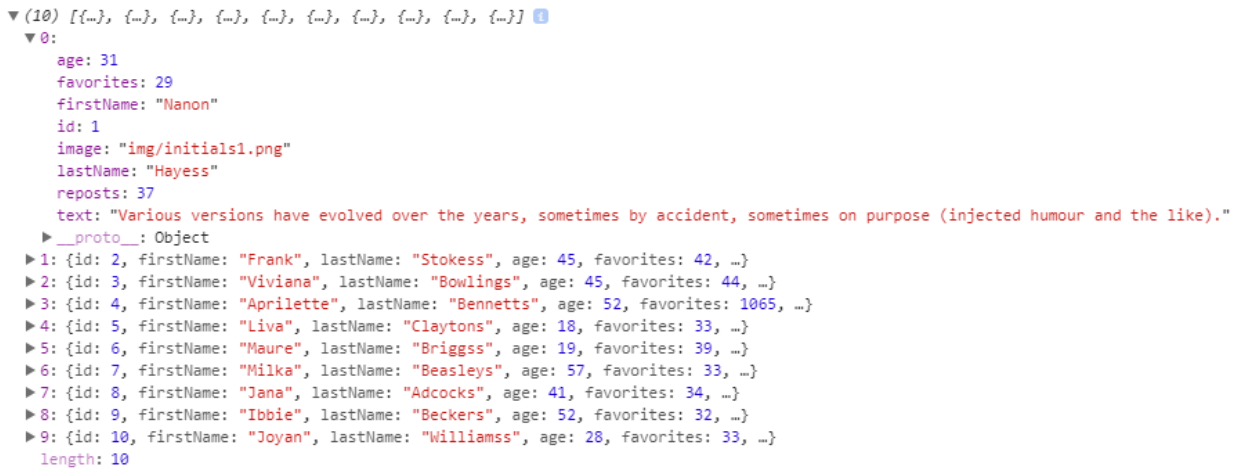
8.6. att. **Ievades lauks, kas piedāvā iespējas katrā taustiņa nospiešanas *reizē*** *[50]*

Situācija, kurā šādam laika daudzumam var būt nozīme, ir gadījums, kad modelim piesaistīts ievades lauks, kas pie katras lietotāja taustiņa nospiešanas veic izmaiņas datos, izmantojot divu virzienu datu piesaisti – lietotājs var pieredzēt situāciju, kad, nospiežot taustiņu, tas neparādās ievades laukā, kamēr notiek netīro datu pārbaude (skat. 8.6. att.). Šie rezultāti parāda to, ka var būt svarīgi veikt vienas reizes datu piesaisti lietotnes daļās, kurās tas iespējams, jo tādējādi netiek izveidoti vērotāji, kas katras izmaiņas rezultātā jāpārbauda. Svarīgi, ka netīro datu pārbaudes process notiek ne tikai brīžos, kad notiek izmaiņas sarakstam piesaistītajos datos, bet

gan jebkuriem skatam piesaistīto datu izmainīšanas brīdī. Gadījumā, ja lietotāja saskarnē ir piemērā izmantotais saraksts ar 15000 elementu, kam ir vajadzīga tikai vienas reizes datu piesaiste, taču ir izmantota noklusētā divu virzienu datu piesaiste, katrai modeļa-skata sinhronizācijai jāsalīdzina daudz vairāk vērotāju, salīdzinot ar scenāriju, kad šim sarakstam izmantota vienas reizes datu piesaiste [51]. Konkrētajā scenārijā divu virzienu datu piesaiste izraisa niecīgus veiktspējas zudumus, un pamanāmākai atšķirībai būtu bijis nepieciešams sarežģītāks piemērs un dziļāka izpēte.

8.3 Knockout JS

Nodaļas, kurā aprakstītas Angular JS datu piesaistes īpatnības, diagrammā redzams, ka saraksta un lietotāja saskarnes DOM elementu skaitam pieaugot, Knockout JS ietvarā viena elementa nomainīšanai nepieciešamais laiks ir konstants un neaug, jo skats ar modeli ir saistīti ar ziņošanas mehānismu, kas nozīmē, ka, izmaiņai notiekot, attiecīgajam DOM elementam tiek paziņots par izmaiņām, un tas var veikt atbilstošo darbību.



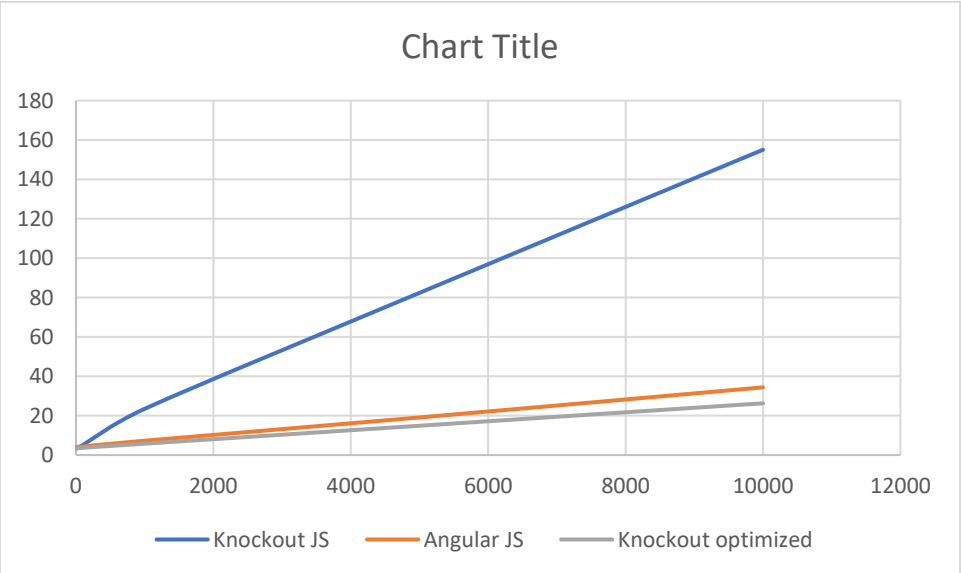
8.7. att. **Tīri JavaScript dati**

Taču ziņošanas mehānisma uzstādīšana aizņem atmiņu, kā arī modelis sastāv no novērojamajiem objektiem, nevis tīriem JavaScript datiem kā Angular JS un React JS ietvaros [18].

```
▼(10) [{…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}, {…}]
  ▼0:
    ▼age: ƒ c()
      ▶F: {change: Array(0)}
      ▶Hc: ƒ ()
      ▶Ja: ƒ (b,c)
      ▶Ob: ƒ (a)
      ▶Pb: ƒ (a)
        Qb: 1
        gb: true
      ▶notifySubscribers: ƒ U(a, c)
        Symbol(_latestValue): 31
        arguments: null
        caller: null
        length: 0
        name: "c"
      ▶prototype: {constructor: ƒ}
      ▶__proto__: Function
        [[FunctionLocation]]: knockout-min.js:40
      ▶[[Scopes]]: Scopes[4]
    ▶favorites: ƒ c()
    ▶firstName: ƒ c()
    ▶id: ƒ c()
    ▶image: ƒ c()
    ▶lastName: ƒ c()
    ▶reposts: ƒ c()
    ▶text: ƒ c()
    ▶__proto__: Object
  ▶1: {id: ƒ, firstName: ƒ, lastName: ƒ, age: ƒ, favorites: ƒ, …}
  ▶2: {id: ƒ, firstName: ƒ, lastName: ƒ, age: ƒ, favorites: ƒ, …}
  ▶3: {id: ƒ, firstName: ƒ, lastName: ƒ, age: ƒ, favorites: ƒ, …}
  ▶4: {id: ƒ, firstName: ƒ, lastName: ƒ, age: ƒ, favorites: ƒ, …}
  ▶5: {id: ƒ, firstName: ƒ, lastName: ƒ, age: ƒ, favorites: ƒ, …}
  ▶6: {id: ƒ, firstName: ƒ, lastName: ƒ, age: ƒ, favorites: ƒ, …}
  ▶7: {id: ƒ, firstName: ƒ, lastName: ƒ, age: ƒ, favorites: ƒ, …}
  ▶8: {id: ƒ, firstName: ƒ, lastName: ƒ, age: ƒ, favorites: ƒ, …}
  ▶9: {id: ƒ, firstName: ƒ, lastName: ƒ, age: ƒ, favorites: ƒ, …}
    length: 10
```


8.8. att. **Tie paši dati kā novērojamie objekti**

Attēlos redzams masīvs pirms (skat. 8.7. att.) un pēc (skat. 8.8. att.) pārveidošanas par novērojamo objektu modeli, un ir redzams, ka pēc pārveidošanas datu struktūra ir apjomīgāka – katra objekta katrs atribūts ir novērojamais objekts, kas spējīgs paziņot par izmaiņām.



8.9.att. **Aizņemtais atmiņas daudzums atkarībā no elementu skaita**

Attēlā (skat. 8.9.att.) redzams aizņemtās atmiņas pieaugums megabaitos atkarībā no saraksta, kas ģenerēts no datu modeļa, izmantojot datu piesaisti, elementu skaita pieauguma (skat. 17. pielikums). Saraksta elementiem ir vienkārša struktūra, tāpēc tie aizņem maz atmiņas un uzsvars testā ir uz modeļa aizņemto atmiņas daudzumu. Ja vienas lapas tīmekļa lietotnei ir paredzēts darboties viedtālruņos ar mazu apjomu operatīvās atmiņas, tad, atmiņas daudzumam pieaugot, var rasties nereaģējošas lietotāja saskarnes efekts, kā arī interneta pārlūks var kļūt pilnīgi nereaģējošs un tikt aizvērts. Šādā situācijā var veikt vienas reizes datu piesaisti DOM elementiem un sarakstiem, kurus paredzēts tikai ģenerēt un kuriem nav jāreaģē uz izmaiņām modelī [18]. Diagrammā redzams, ka, veicot vienas reizes datu piesaisti, atmiņas pieaugums atkarībā no elementu skaita ir tuvs Angular JS. Vēl jāņem vērā, ka ir iespējams par novērojamajiem objektiem pārveidot tikai noteiktus masīva objektu atribūtus, piemēram, tikai *firstName* atribūtu (skat. 8.7. att.), ja ir zināms, ka tikai tas mainīsies un pārējie nemainīsies, tādējādi novērojamo objektu būs mazāk, un atmiņa tiks ietaupīta [22].

REZULTĀTI

Bakalaura darba ietvaros tika izpētītas dažādas veiktspējas problēmas, ar ko iespējams saskarties, izstrādājot vienas lapas tīmekļa lietotnes, un to iespējamie risinājumi. Tika izskatītas iepriekšējos pētījumos neizskatītas veiktspējas problēmas, un daļa iepriekšējos pētījumos izskatīto veiktspējas problēmu izpēte tika papildināta.

Tika izpētīts tīmekļa pārlūkprogrammu automātiskais atkritumu savākšanas mehānisms, un ar testa lietotnēm tika veikti praktiski eksperimenti, lai pārbaudītu scenārijus, kuros iespējams izraisīt atmiņas noplūdes.

Tika izpētītas iespējas veikt aprēķinus vairākos pavedienos programmēšanas valodā JavaScript un ar testa lietotnēm tika pārbaudīta un novērtēta vairāku pavedienu efektivitāte konkrētos scenārijos.

Tika papildināta iepriekš veikta pētījuma testa lietotne dažādu DOM manipulāciju veikšanas scenāriju salīdzinājumam ar papildus scenārijiem un apjomīgākas DOM struktūras elementiem, un tika veikti un analizēti veiktspējas mērījumi un papildināti iepriekš veiktā pētījuma atzinumi.

Tika izpētīti animāciju un lapas satura ritināšanas veiktspējas problēmgadījumi un izveidotas testa lietotnes veiktspējas mērījumiem, salīdzinot ar optimizētām versijām, lai pārbaudītu problēmgadījumu aktualitāti un optimizāciju ietekmi uz veiktspēju.

Ietvaru izpētē tika sasniegti nepilnīgi rezultāti sakarā ar ierobežoto laiku un nelielo pieredzi, konkrētos ietvarus izmantojot, tāpēc tika tikai teorētiski apskatītas datu piesaistes metodes un izstrādāti testpiemēri pamatfaktoru veiktspējas pārbaudei, un katrā ietvarā eksistē vēl daudz ar datu piesaisti saistītu un nesaistītu faktoru, kas ietekmē veiktspēju.

Izveidotās testa lietotnes ļauj lasītājam pārliecināties par konkrētu veiktspējas problēmu aktualitāti, testēt lietotnes dažādās ierīcēs, kā arī papildināt to kodu, lai pārveidotu kodu sev interesējošiem scenārijiem.

Darba rezultātā no praktiski pārbaudītajām veiktspējas problēmām tika izstrādāts apkopojums, kas satur vadlīnijas darbā izskatīto veiktspējas problēmu novēršanai (skat. 1. pielikums).

SECINĀJUMI

Tīmekļa pārlūkprogrammas regulāri tiek uzlabotas, taču vēl arvien eksistē problēmas, kam jāpievērš uzmanība, veidojot vienas lapas tīmekļa lietotnes ar optimālu veiktspēju.

Neskatoties uz to, ka JavaScript valodā eksistē automātisks nevajadzīgās atmiņas atbrīvošanas mehānisms, eksistē veidi, kā neatbrīvot neizmantotu atmiņu, kas tika pierādīts, izveidojot un izpētot praktiskus piemērus. Var secināt, ka vienas lapas tīmekļa lietotnēs atmiņas pārvaldībai ir jāpievērš uzmanība, jo katras lapas ielādes rezultātā nenotiek lapas pārlāde, kas atbrīvo atmiņu.

Vienas lapas tīmekļa lietotnēs salīdzinājumā ar vairāku lapu lietotnēm potenciāli vairāk darba tiek darīts klienta pusē un mazāk darba tiek darīts servera pusē, kas nozīmē, ka var rasties vajadzība pēc apjomīgiem aprēķiniem. Pēc praktisko piemēru izpētes tiek secināts, ka eksistē gan situācijas, kad vairākus pavedienus var izmantot tikai lietotāja saskarnes bloķēšanas novēršanai, gan arī situācijas, kad tos var izmantot aprēķinu procesa paātrināšanai.

Eksistē dažādi veidi, kā valodā JavaScript veikt DOM manipulācijas. Neskatoties uz to, ka tīmekļa pārlūkprogrammas ir optimizētas visas DOM manipulācijas veikt pēc JavaScript koda izpildes beigām, eksistē elementu īpašības un JavaScript funkcijas, kas var izraisīt piespiedu izkārtošanas procesu un pasliktināt veiktspēju. Dažādām DOM manipulācijām var atšķirties veiktspēja gan ar, gan bez piespiedu izkārtošanas procesa, un, nepareizi veicot DOM manipulācijas, ir iespējams sasniegt atbildes laiku, kas ievērojami pārsniedz cilvēka uzmanības intervālu, pat tikai 200 *Twitter* ierakstu ģenerēšanā. No piespiedu izkārtošanas procesa ir iespējams izvairīties, taču ne visos gadījumos. Šīs veiktspējas problēmas attiecas uz gadījumiem, kad piespiedu izkārtošanas process tiek veikts vairākas reizes, un vairumā gadījumu, kad kādas lietotāja darbības rezultātā tas tiks izsaukts, piemēram, vienu reizi, veiktspējas problēmas neradīsies.

Eksistē dažādas CSS īpašības – tādas, kas izraisa izkārtojuma pārrēķināšanu, un tādas, kas to neizraisa. Pēc praktiska piemēra apskates tiek secināts, ka vienkāršām animācijām ir mazsvarīgi, kādas CSS īpašības tiek lietotas, taču, veicot animācijas ar sarežģītākiem un vairākiem elementiem, ir nozīmīgi izmantot īpašības, kas izkārtojuma pārrēķināšanu neizsauc. Fiksētu elementu praktiska piemēra mērījumi liecināja par to, ka šādu elementu izvietošana atsevišķā slānī var uzlabot veiktspēju, jo netiek pārrēķināta to atrašanās vieta attiecībā pret citiem elementiem. Tika secināts, ka elementiem piešķirtās CSS īpašības, piemēram, ēnojums var salīdzinoši būtiski ietekmēt veiktspēju.

Izstrādes ietvaru datu piesaistes teorētisko pamatu apskate un praktisko piemēru mērījumi vērsa uzmanību uz to, ka, izstrādājot tīmekļa lietotnes konkrētā izstrādes ietvarā, ir svarīgi zināt to, kā ietvarā strādā datu piesaiste.

Pētījumu var attīstīt tālāk – nav izskatīti visi iespējamie scenāriji un eksistē dažādas ierīces, kurās var izmantot vienas lapas tīmekļa lietotnes, ar dažādas jaudas procesoriem, videokartēm un dažādu operatīvās atmiņas apjomu. Testu veikšana ierīcēs ar sliktākiem parametriem var dot lielāku izpratni par to, cik svarīgi ir nepieļaut kļūdas, kas samazina veiktspēju, jo dators, kurā tika veikti testi šajā pētījumā, var rādīt ievērojami labākus rezultātus konkrētā situācijā, nekā, piemēram, mazjaudīgs viedtālrunis. Tāpat programmētāji ar pieredzi konkrētos vienas lapas tīmekļa lietotņu izstrādes ietvaros var veikt dziļāku to izpēti, salīdzināšanu, veiktspēju ietekmējošo faktoru analīzi un testa lietotņu izveidi, lai novērtētu, cik lielā mērā konkrēti faktori ietekmē veiktspēju – tas sniegtu konkrētu izstrādes ietvaru programmētajiem ieskatu tajā, cik svarīgi ir atsevišķi veiktspēju ietekmējošie faktori. Pētījumā veiktspējas mērījumi tika veikti *Google Chrome* pārlūkprogrammā, jo to izmanto vairāk nekā puse interneta lietotāju un tai eksistē veiktspējas mērīšanas rīki ar plašu funkcionalitāti, taču, veicot veiktspējas mērījumus dažādās pārlūkprogrammās, var noteikt, vai konkrētas veiktspējas problēmas ir aktuālas visās pārlūkprogrammās vai tikai daļā no tām.

IZMANTOTĀ LITERATŪRA UN AVOTI

1. Single-Page Applications [tiešsaiste]. – [atsauce: 19.02.2018.] Pieejams: https://msdn.microsoft.com/en-us/magazine/dn463786.aspx
2. How Browsers Work [tiešsaiste]. – [atsauce: 19.02.2018.] Pieejams: http://taligarsiel.com/Projects/howbrowserswork1.htm
3. React – A JavaScript Library for Building User Interfaces [tiešsaiste]. – [atsauce: 19.02.2018.] Pieejams: https://reactjs.org/
4. Angular JS [tiešsaiste]. – [atsauce: 19.02.2018] Pieejams: https://angularjs.org/
5. Knockout JS [tiešsaiste]. – [atsauce: 19.02.2018.] Pieejams: http://knockoutjs.com/
6. Using Web Workers [tiešsaiste]. – [atsauce: 19.02.2018.] Pieejams: https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers
7. Element.innerHTML [tiešsaiste]. – [atsauce: 08.05.2018.] Pieejams: https://developer.mozilla.org/en-US/docs/Web/API/Element/innerHTML
8. HTMLElement.offsetWidth [tiešsaiste]. – [atsauce: 08.05.2018.] Pieejams: https://developer.mozilla.org/en-US/docs/Web/API/HTMLElement/offsetWidth
9. On Layout & Web Performance [tiešsaiste]. – [atsauce: 10.05.2018.] Pieejams: http://kellegous.com/j/2013/01/26/layout-performance/
10. HTML Syntax [tiešsaiste]. – [atsauce: 09.05.2018.] Pieejams: https://www.w3.org/html/wg/wiki/AuthorSyntax
11. Constructing The Object Model [tiešsaiste]. – [atsauce: 09.05.2018.] Pieejams: https://developers.google.com/web/fundamentals/performance/critical-rendering-path/constructing-the-object-model
12. Render Tree Construction [tiešsaiste]. – [atsauce: 10.05.2018] Pieejams: https://developers.google.com/web/fundamentals/performance/critical-rendering-path/render-tree-construction
13. Memory Management [tiešsaiste]. – [atsauce: 19.02.2018.] Pieejams: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management
14. Performance: Rendering [tiešsaiste]. – [atsauce: 10.05.2018.] Pieejams: https://developers.google.com/web/fundamentals/performance/rendering/
15. JavaScript [tiešsaiste]. – [atsauce: 06.03.2018.] Pieejams: https://developer.mozilla.org/bm/docs/Web/JavaScript

16. Document Object Model [tiešsaiste]. – [atsauce: 06.03.2018.] Pieejams: https://developer.mozilla.org/en-US/docs/Web/API/Document_Object_Model/Introduction
17. One-Way Data Binding vs. Two-Way Data Binding [tiešsaiste]. – [atsauce: 06.03.2018.] Pieejams: https://study.com/academy/lesson/one-way-data-binding-vs-two-way-data-binding.html
18. Observables [tiešsaiste]. – [atsauce: 06.03.2018.] Pieejams: http://knockoutjs.com/documentation/observables.html
19. Developer Guide: Scopes - AngularJS Docs [tiešsaiste]. – [atsauce: 06.03.2018.] Pieejams: https://docs.angularjs.org/guide/scope
20. The Difference Between Virtual DOM and DOM [tiešsaiste]. – [atsauce: 06.03.2018.] Pieejams: http://reactkungfu.com/2015/10/the-difference-between-virtual-dom-and-dom/
21. Event Delegation [tiešsaiste]. – [atsauce: 06.03.2018.] Pieejams: https://javascript.info/event-delegation
22. Knockout JS mappings [tiešsaiste]. – [atsauce: 26.05.2018.] Pieejams: https://www.codeproject.com/articles/507941/knockout-js-mappings
23. Addy Osmani, Learning JavaScript Design Patterns, O'Reilly Media, 2012.
24. How to Get The Average Fps in Chrome Dev Tools [tiešsaiste]. – [atsauce: 16.05.2018.] Pieejams: https://stackoverflow.com/questions/48079661/how-to-get-the-average-fps-in-chrome-devtools
25. Nicholas C. Zakas, High Performance JavaScript, O'Reilly Media, 2010.
26. David Flannagan, JavaScript: The Definitive Guide, O'Reilly Media, 2001.
27. Wesley Hales. HTML5 and JavaScript Web Apps, O'Reilly Media, 2012.
28. J.M. Gustafson. HTML Web Application Development By Example, Packt Publishing, 2013.
29. Event Loop [tiešsaiste]. – [atsauce: 25.04.2018.] Pieejams: https://developer.mozilla.org/en-US/docs/Web/JavaScript/EventLoop
30. Philip Roberts: Event Loop [tiešsaiste]. – [atsauce: 25.04.2018.] Pieejams: https://www.youtube.com/watch?v=8aGhZQkoFbQ&t=95s
31. High Performance Web Worker Messages [tiešsaiste]. – [atsauce: 25.04.2018.] Pieejams: https://nolanlawson.com/2016/02/29/high-performance-web-worker-messages/

32. Beginners guide to web development. Single Page Applications [tiešsaiste]. – [atsauce: 25.04.2018.] Pieejams: https://www.codeschool.com/beginners-guide-to-web-development/single-page-applications
33. Single page apps in depth [tiešsaiste]. – [atsauce: 25.04.2018.] Pieejams: http://singlepageappbook.com/goal.html
34. HTML Select Form [tiešsaiste]. – [atsauce: 25.04.2018.] Pieejams: https://www.w3schools.com/html/tryit.asp?filename=tryhtml_elem_select_pre
35. What is the Document Objet Model [tiešsaiste]. – [atsauce: 25.04.2018.] Pieejams: https://www.w3.org/TR/WD-DOM/introduction.html
36. JavaScript HTML DOM. [tiešsaiste] – [atsauce: 25.04.2018.] Pieejams: https://www.w3schools.com/js/js_htmldom.asp
37. Browser & Platform Market Share [tiešsaiste]. – [atsauce: 23.03.2018.] Pieejams: https://www.w3counter.com/globalstats.php?year=2018&month=3
38. Paul Irish. Why Moving Elements With Translate() Is Better Than Pos:abs Top/left [tiešsaiste]. – [atsauce: 21.05.2018.] Pieejams: https://www.paulirish.com/2012/why-moving-elements-with-translate-is-better-than-posabs-topleft/
39. Introducting JSX [tiešsaiste]. – [atsauce: 25.04.2018.] Pieejams: https://reactjs.org/docs/introducing-jsx.html
40. Memory Leaks in JavaScript [tiešsaiste]. – [atsauce: 25.04.2018.] Pieejams: https://www.youtube.com/watch?v=CRoR_0K586I&list=PLSyMwbwM_tntJjtmW1I3-M4_r2_UNfwCI
41. Four Types Of Leaks In Your JavaScript code And How To Get Rid Of Them [tiešsaiste]. – [atsauce: 25.04.2018.] Pieejams: https://auth0.com/blog/four-types-of-leaks-in-your-javascript-code-and-how-to-get-rid-of-them/
42. Masoomeh Rudafshani, “Detection and Diagnosis of Memory Leaks in Web Applications”, Electrical and Computer Engineering, University of Waterloo, Waterloo, Canada, 2015.
43. Riku Nieminen, “Client-side Web Application Memory Management”, Faculty of Computing and Electrical Engineering, Tampere University of Technology, Tampere, Finland, 2015.
44. Memory Management [tiešsaiste]. – [atsauce: 25.04.2018.] Pieejams: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Memory_Management
45. Tero Parviainen. Change and its detection in JavaScript frameworks [tiešsaiste]. – [atsauce: 25.04.2018.] Pieejams: http://teropa.info/blog/2015/03/02/change-and-its-detection-in-javascript-frameworks.html

46. Christopher Chedeau. React's diff algorithm [tiešsaiste]. – [atsauce: 24.05.2018.] Pieejams: https://calendar.perfplanet.com/2013/diff/
47. React JS: Optimizing performance [tiešsaiste]. – [atsauce: 25.04.2018.] Pieejams: https://facebook.github.io/react/docs/optimizing-performance.html
48. Pete Hunt. The Secrets of React's Virtual DOM [tiešsaiste]. – [atsauce: 25.04.2018.] Pieejams: https://www.youtube.com/watch?v=-DX3vJiqxm4
49. Angular JS: $digest() [tiešsaiste]. – [atsauce: 25.04.2018.] Pieejams: https://docs.angularjs.org/api/ng/type/$rootScope.Scope#$digest
50. YouTube [tiešsaiste]. – [atsauce: 25.04.2018.] Pieejams: https://www.youtube.com/
51. Angular JS: One-time binding [tiešsaiste]. – [atsauce: 25.04.2018.] Pieejams: https://docs.angularjs.org/guide/expression#one-time-binding
52. Tutorial: Intro to React [tiešsaiste]. – [atsauce: 26.05.2018.] Pieejams: https://reactjs.org/tutorial/tutorial.html
53. Romāns Kolduns, "Tīmekļa lietotņu klienta puses optimizācija", Bakalaura darbs, Datorikas fakultāte, Latvijas Universitāte, Rīga, Latvija, 2017.
54. Ēriks Šarapovs, "Tīmekļa lietotņu ātrdarbības optimizācija", Maģistra darbs, Datorikas fakultāte, Latvijas Universitāte, Rīga, Latvija, 2018.
55. Compositor-only properties [tiešsaiste]. – [atsauce: 14.05.2018.] Pieejams: https://developers.google.com/web/fundamentals/performance/rendering/stick-to-compositor-only-properties-and-manage-layer-count
56. Why Moving Elements With Translate() Is Better Than Pos:abs Top/left [tiešsaiste]. – [atsauce: 14.05.2018.] Pieejams: https://www.paulirish.com/2012/why-moving-elements-with-translate-is-better-than-posabs-topleft/
57. Avoid Large, Complex Layouts and Layout Thrashing [tiešsaiste]. – [atsauce: 14.05.2018.] Pieejams: https://developers.google.com/web/fundamentals/performance/rendering/avoid-large-complex-layouts-and-layout-thrashing
58. Fixed background image performance issue [tiešsaiste]. – [atsauce: 15.05.2018.] Pieejams: https://medium.com/vehikl-news/fixed-background-image-performance-issue-6b7d9e2dbc55
59. Simplify Paint Complexity and Reduce Paint Areas [tiešsaiste]. – [atsauce: 15.05.2018.] Pieejams: https://developers.google.com/web/fundamentals/performance/rendering/simplify-paint-complexity-and-reduce-paint-areas

60. Using Chrome DevTools to profile the jsconf.eu site [tiešsaiste]. – [atsauce: 15.05.2018.] Pieejams: https://www.youtube.com/watch?v=QU1JAW5LRKU
61. What forces layout/reflow [tiešsaiste]. – [atsauce: 17.05.2018.] Pieejams: https://gist.github.com/paulirish/5d52fb081b3570c81e3a
62. An Introduction to Hardware Acceleration with CSS Animations [tiešsaiste]. – [atsauce: 18.05.2018.] Pieejams: https://www.sitepoint.com/introduction-to-hardware-acceleration-css-animations/
63. Angular's $digest Is Reborn In The Newer Version Of Angular [tiešsaiste]. – [atsauce: 19.05.2018.] Pieejams: https://blog.angularindepth.com/angulars-digest-is-reborn-in-the-newer-version-of-angular-718a961ebd3e
64. Jakob Nielsen, Usability Engineering, Morgan Kaufmann, 1993.
65. Steven C. Seow, Designing and Engineering Time, Addison-Wesley Professional, 2008.
66. Chrome Devtools: Evaluate Performance [tiešsaiste]. – [atsauce: 26.05.2018.] Pieejams: https://developers.google.com/web/tools/chrome-devtools/evaluate-performance/
67. Chrome Devtools: Memory Problems [tiešsaiste]. – [atsauce: 26.05.2018.] Pieejams: https://developers.google.com/web/tools/chrome-devtools/memory-problems/

PIELIKUMI

1. pielikums. **Vienas lapas tīmekļa lietotņu izstrādes vadlīnijas**

Kopsavilkums bakalaura darbā pētīto veiktspējas problēmu novēršanai

Problēmgadījums:
Lietotnes darbības laikā pārlūka aizņemtā atmiņa turpina pieaugt un daļa no atmiņas netiek atbrīvota.
Risinājums:
Nepieciešams pārbaudīt, vai visas notikumu apstrādes funkcijas, kam eksistē reference uz kādu no lietotnes izdzēstu elementu, tiek noņemtas. Piemēram, šajā piemērā mainīgais list tiek paturēts atmiņā arī brīdī, kad tajā glabātais DOM elements tas tiek izdzēsts:
`var list = document.getElementById("posts");//Ar šo variantu tiks izraisīta atmiņas noplūde`
`document.getElementById('leakbutton').addEventListener("click", function () {`
`  list.remove();`
`});`
Lai mainīgais *list* tiktu izdzēsts no atmiņas, nepieciešams noņemt nospiešanas notikuma apstrādes funkciju elementam ar *id* “leakbutton” vai arī nepieciešams definēt mainīgo *list* iekšā notikuma apstrādes funkcijā.
Brīdinājumi:
Nav

Problēmgadījums: Lietotnes darbības laikā pārlūka aizņemtā atmiņa turpina pieaugt, un daļa no atmiņas netiek atbrīvota. **Risinājums:** Nepieciešams pārliecināties, vai, definējot maza apjoma mainīgos, nav palikusi reference uz kādu mainīgo, kas aizņem daudz atmiņas. Piemēram, šajā piemērā tiek izmantota tikai neliela daļa no objekta, taču, tā kā notikuma apstrādes funkcijā ir reference uz mainīgo *oObject*, tas tiek paturēts atmiņā.

```
var attachTheListener = function(){
    var a1=[];
    var a2=[];
    for(var i=0; i<300000; i++){
        a1.push({name:"test"});
    }
    for(var i=0; i<300000; i++){
        a2.push({name:"test"});
    }
var oObject={
    p1:a1,
    p2:a2
}
document.getElementById('leakbutton').addEventListener("click", function () {
    alert(oObject.p1[0].name);
});
}
attachTheListener();
```

Lai šādu gadījumu novērstu, šī nelielā daļa no mainīgā *oObject* būtu jādefinē ārpus notikuma apstrādes funkcijas, un tad jāizmanto notikuma apstrādes funkcijā.

Brīdinājumi:

Nav

Problēmgadījums:

Lietotnes darbības laikā pārlūka aizņemtā atmiņa turpina pieaugt un daļa no atmiņas netiek atbrīvota.

Risinājums:

Nepieciešams pārliecināties, par to, vai neeksistē setInterval funkcijas, kas turpina darboties, lai gan elementi, ar ko tajos tiek veiktas darbības, vairs neeksistē. Piemēram, šajā gadījumā mainīgais element paliks atmiņā arī tad, kad elements ar id leakbutton būs izdzēsts.

```
var leakExample=function(){
    var posts = getData();
    setInterval(function() {
        var element = document.getElementById('leakbutton');
        //setInterval atstās atmiņā posts masīvu arī tad, kad izmantotais DOM elements ar id 'leakbutton' būs noņemt
        if(element) {
            element.innerHTML = posts[Math.floor(Math.random() * 10)].id;
        }
    }, 1000);
}
leakExample();
```

Šajā situācijā, lai izvairītos no negatīvajām sekām, elementa ar id leakbutton dzēšanas laikā būtu nepieciešams arī noņemt šo intervālu, izmantojot *clearInterval()* metodi.

Brīdinājumi:

Nav

Problēmgadījums:

Lietotnes darbības laikā pārlūka aizņemtā atmiņa turpina pieaugt un daļa no atmiņas netiek atbrīvota.

Risinājums:

Nepieciešams pārliecināties par to, vai netiek izveidoti nejauši globāli mainīgie – tie tiek paturēti atmiņā arī pēc tam, kad funkcijas darbība ir beigusies. Šajā piemērā tiek nejauši izveidots globāls mainīgais *window.posts*:

```
var leakExample=function(){
        posts = getData();//Šis kods izveido nevis lokālu mainīgo, bet window.posts
        var element = document.getElementById('leakbutton');
        if(element) {
            element.innerHTML = posts[Math.floor(Math.random() * 10)].id;
        }
}
leakExample();
```

Lai izvairītos no šīs situācijas, nepieciešams lietot atslēgas vārdus *var* vai *let* pirms mainīgā deklarēšanas.

Brīdinājumi:

Nav

Problēmgadījums:

Aizņemts vairāk atmiņas, nekā nepieciešams.

Risinājums:

Nelielu optimizāciju atmiņas izmantošanā iespējams veikt, izmantojot notikumu apstrādes funkciju pārnešanu augstākā līmenī.

A

```
for(var i=0; i<30000; i++){
    var listElement=document.createElement("li");
    listElement.innerHTML = Math.floor(Math.random()*10)+1;
    books.appendChild(listElement);
    listElement.addEventListener('click', function(e) {
    if (e.target.innerHTML==="10") {
        alert("10 clicked.");
    }
});
}
```

B

```
books.addEventListener('click', function(e) {
    if (e.target.innerHTML==="10") {
        alert("10 clicked.");
    }
}, false);


for(var i=0; i<30000; i++){
    var listElement=document.createElement("li");
    listElement.innerHTML = Math.floor(Math.random()*10)+1;
    books.appendChild(listElement);
}
```

Augstāk redzamajos scenārijos ar notikumu piesaistes funkcijām tiek panāks viens un tas pats rezultāts, taču atšķirība ir tāda, ka scenārijā B tiek izveidota tikai viena notikuma apstrādes funkcija, taču scenārijā A katram saraksta elementam tiek izveidota notikuma apstrādes funkcija.

Brīdinājumi:

Nav

Problēmgadījums:

No servera pieprasīto datu apstrāde klienta pusē aizņem laiku un bloķē tīmekļa lietotnes lietotāja saskarni.

Risinājums:

Lai novērstu to, ka lietotāja saskarne ir bloķēta un nereaģē laikā, kamēr tiek veikta datu apstrāde, ir iespējams pieprasīt servera datus no tīmekļa strādņa, kas saņems un apstrādās datus citā pavedienā, nebloķējot lietotāja saskarni, un atgriezīs jau apstrādātos datus galvenajam pavedienam.

Brīdinājumi:

Nav

Problēmgadījums:

Tīmekļa lietotnē sarežģīta datu apstrāde aizņem laiku un bloķē tīmekļa lietotnes lietotāja saskarni.

Risinājums:

Datus var nosūtīt tīmekļa strādnim, lai veiktu aprēķinus citā pavedienā un, kad aprēķini tiek pabeigti, tie tiek nosūtīti atpakaļ galvenajam pavedienam – šādā veidā netiek bloķēta tīmekļa lietotnes lietotāja saskarne. Ir vērts pārbaudīt to, cik ilgā laikā šie aprēķini tiek izpildīti, sadalot šos aprēķinus mazākās daļās un nododot tos vairākiem tīmekļa strādņiem. Pastāv iespēja, ka eksistē konkrēts tīmekļa strādņu daudzums, ar ko tiek sasniegta optimāla veiktspēja, un, izmantojot vairāk vai mazāk tīmekļa strādņu, veiktspēja samazinās.

Brīdinājumi:

Datu nosūtīšanai tīmekļa strādnim nepieciešams laiks, ja dati ir apjomīgi. Tīmekļa strādņu izmantošana pasliktinās veiktspēju, ja datu nosūtīšanai būs nepieciešams ilgāks laiks, nekā datu apstrādei galvenajā pavedienā. Tas nozīmē, ka lietotāja saskarne tiks bloķēta ilgāk, nekā datu apstrādei galvenajā pavedienā, kā arī apstrādātie dati vēl nebūs pieejami, kamēr tos vēl apstrādās tīmekļa strādnis.

Problēmgadījums:

Nepietiekama veiktspēja, vairākkārt papildinot kādu DOM elementu nepārtraukta koda izpildes laikā, izmantojot *innerHTML* funkciju.

```
for (var i = 0; i < iterations; i++) {
div.innerHTML += '<li>List item ' + i + '</li>';
}
console.timeEnd("Div - append " + iterations + "x");
}
```

Risinājums:

Ja katrā reizē, kad jāpapildina DOM elements, nav nepieciešams to fiziski pievienot, ir iespējams izveidot simbolu virkni, kur glabāt kodu, kas jāpievieno DOM elementam, un tikai beigās izpildīt *innerHTML* funkciju, tikai vienu reizi aizstājot DOM elementa saturu.

```
var html = '';
console.time("Div - append join " + iterations + "x");
for (var i = 0; i < iterations; i++) {
html += '<li>List item ' + i + '</li>';
}
div.innerHTML = html;
console.timeEnd("Div - append join " + iterations + "x");
```

Brīdinājumi:

Nav

Problēmgadījums:

Nepietiekama veiktspēja, vairākkārt papildinot kādu DOM elementu nepārtraukta koda izpildes laikā, izmantojot *innerHTML* funkciju.

```
for (var i = 0; i < iterations; i++) {
div.innerHTML += '<li>List item ' + i + '</li>';
}
console.timeEnd("Div - append " + iterations + "x");
}
```

Risinājums:

innerHTML funkciju var aizstāt ar *insertAdjacentHTML* vai *appendChild* funkciju, kas neaizvieto visu DOM elementa saturu, bet gan tikai pievieno jauno elementu.

```
for (var i = 0; i < iterations; i++) {
var el = document.createElement('li');
el.innerText = 'List item ' + i;
div.appendChild(el);
}
console.timeEnd("Div - appendChild " + iterations + "x");
}
```

appendChild funkcijai nav iespējams padot simbolu virkni kā *innerHTML* funkcijai, taču *insertAdjacentHTML* funkcijai tas ir iespējams bez negatīvajām sekām.

Brīdinājumi:

Nav

Problēmgadījums:

Nepietiekama veiktspēja, vairākkārt papildinot kādu DOM elementu nepārtraukta koda izpildes laikā, izmantojot innerHTML funkciju.

```
for (var i = 0; i < iterations; i++) {
div.innerHTML += '<li>List item ' + i + '</li>';
w = div.offsetWidth;
}
console.timeEnd("Div - append with reflow " + iterations + "x");
}
```

Risinājums:

Šādā gadījumā, kad katru reizi tiek papildināts DOM elementa saturs, izmantojot innerHTML, viss DOM elementa saturs tiek nomainīts pret kodu, kas tiek uzstādīts innerHTML metodē. Tā kā tiek izsaukta offsetWidth metode, visam DOM elementa saturam katru reizi tiek izsaukts izkārtošanas process. Lai šādā situācijā mazinātu negatīvo ietekmi uz veiktspēju, var aizvietot innerHTML metodi ar appendChild() metodi, kas neaizstāj visu DOM elementa saturu katrā papildināšanas reizē, līdz ar to izkārtošanas process netiks izsaukts katru reizi visam DOM elementa saturam, bet tikai saturam, ar ko DOM elements ir papildināts.

```
for (var i = 0; i < iterations; i++) {
var el = document.createElement('li');
el.innerText = 'List item ' + i;
div.appendChild(el);
w=div.offsetWidth;
}
```

Brīdinājumi:

Šajā gadījumā nestrādās uzlabošana, sastādot papildināmā satura simbola virkni un tikai pēc tam to uzstādot, izmantojot innerHTML, jo, sastādot šo simbolu virkni, papildināmais saturs netiek fiziski ievietots papildināmajā DOM elementā – tas notiek tikai beigās, kad tiek izsaukta innerHTML metode, līdz ar to offsetWidth funkcija neatgriezīs datus ar simbolu virknē sastādīto ievietojamo saturu, ja vēl nav izsaukta innerHTML metode.

Problēmgadījums:

Nepietiekama veiktspēja, vairākkārt papildinot kādu DOM elementu nepārtraukta koda izpildes laikā, regulāri veicot kāda neatkarīga elementa ģeometrisko īpašību nolasīšanu.

```
for (var i = 0; i < iterations; i++) {
var divInputWrap = document.createElement('div');
.......
aPostPropertyStar.innerText='Star ('+i+')';
div.appendChild(divInputWrap);
h=document.getElementById("listElementCount").offsetHeight;
}
```

Risinājums:

Šajā gadījumā tas, ka lietotāja saskarnes DOM katrā cikla iterācijā tiek pievienots saraksta elements, nozīmē to, ka eksistē izkārtošanas procesi, kas atlikti uz koda izpildes beigām. Ja nepieciešams nolasīt kādu lietotāja saskarnes ģeometrisko īpašību, kas nav saistīta ar katrā iterācijā pievienoto elementu, tik un tā tiks veikts piespiedu izkārtošanas process pievienojamajam elementam. Kā risinājums ir iespējams kāds no šiem variantiem:

- Uzstādīt sarakstam CSS īpašību *display: none,* kamēr tam tiek pievienoti elementi
- Klonēt saraksta elementu, pievienot visus vajadzīgos elementus un tad aizvietot oriģinālu
- Sastādīt sarakstu, izmantojot dokumenta fragmentu un tad ievietot to lietotāja saskarnes DOM

Šādā veidā katra saraksta elementa pievienošana neizraisīs piespiedu izkārtošanas procesu, kad tiek nolasīta kāda lietotāja saskarnes ģeometriskā īpašība.

```
for (var i = 0; i < iterations; i++) {
var el = document.createElement('li');
el.innerText = 'List item ' + i;
div.appendChild(el);
w=div.offsetWidth;
}
```

Brīdinājumi:

Aprakstītās metodes, kas ļauj izvairīties no piespiedu izkārtošanas procesa, atgriezīs lietotāja saskarnes ģeometriskās īpašības tādas, kādas tās ir bez pievienotajiem elementiem, tāpēc tas strādās tikai gadījumos, kad ģeometriskās īpašības, kas jānolasa, ir neatkarīgas no elementa, kas, piemēram, atzīmēts kā *display: none*.

Problēmgadījums:

Nepietiekams kadru skaits sekundē, veicot sarežģītu animāciju, piemēram, objekta ar sarežģītu ēnojumu vai vairāku objektu reizē pārvietošanu pa ekrānu.

Risinājums:

Ja animācija veikta ar CSS īpašībām, kas izraisa izkārtošanas, zīmēšanas un kompozitēšanas procesus, piemēram, *top, left, bottom, right*, tad kadru skaitu sekundē iespējams uzlabot, izmantojot CSS īpašības, kas izsauc tikai kompozitēšanas posmu, piemēram, *transform: translate.*

Brīdinājumi:

Nav

Problēmgadījums:

Nepietiekams kadru skaits sekundē, veicot lietotāja saskarnē esošā satura ritināšanu.

Risinājums:

Ja lietotāja saskarnē eksistē kāds fiksēts elements, piemēram, fiksēta galvene, kājene vai fons, tad, iespējams, tas ir cēlonis samazinātam kadru skaitam sekundē. Lai izvairītos no tā, ka fiksētais elements ritināšanas laikā katru reizi tiek pārzīmēts attiecībā pret citiem elementiem, nepieciešams to izvietot citā slānī. Izvietot citā slānī iespējams, izmantojot CSS *will-change*: transform īpašību. Izvietošana citā slānī nodrošina to, ka fiksētais elements nav katru reizi jāpārzīmē attiecībā pret citiem elementiem.

Brīdinājumi:

Katrs jaunizveidotais slānis tiek ielādēts video kartes atmiņā kā tekstūra, līdz ar to ir jāņem vērā, ka, izveidojot par daudz slāņu, šī atmiņa var tikt pārslogota. Tas nozīmē, ka atsevišķu slāni vajadzētu izveidot tikai tad, ja ir redzams, ka kadru skaits sekundē kļūst lielāks.

Problēmgadījums:

Nepietiekama veiktspēja lietotāja saskarnes atjaunināšanai, izmainot piesaistītos datus React JS ietvarā veidotai lietotnei.

Risinājums:

Ja šī veiktspējas problēma rodas, piemēram, sarakstam piesaistīta masīva atjaunināšanas rezultātā, nepieciešams izmantot shouldComponentUpdate metodi, kas pirms katra saraksta elementa JavaScript DOM reprezentācijas izveides pārbauda, vai šī saraksta elementa dati ir mainījušies, salīdzinot ar iepriekšējo reizi. Šādā veidā tā vietā, lai katram elementam salīdzinātu iepriekšējo DOM reprezentāciju ar jaunizveidoto, DOM reprezentācijas tiek salīdzinātas tikai tiem saraksta elementiem kuru dati ir izmainījušies.

Brīdinājumi:

Nav

Problēmgadījums:

Nepietiekama veiktspēja lietotāja saskarnes atjaunināšanai, izmainot piesaistītos datus Angular JS ietvarā veidotai lietotnei.

Risinājums:

Nepieciešams pārbaudīt, vai lietotāja saskarnes elementiem, kam tas iespējams, dati piesaistīti ar vienas reizes datu piesaisti. Šajā gadījumā veiktspējas problēmas, iespējams, rodas sakarā ar to, ka ir izveidots pārāk daudz vērotāju, kas tiek pārbaudīti katrā datu izmaiņas reizē. Ja dati ir piesaistīti ar vienas reizes datu piesaisti, tad vērotāji netiek

izveidoti. Tādējādi, ja tiek visās iespējamajās vietās izmantota vienas reizes datu piesaiste, vērotāju skaits ir mazāks. **Brīdinājumi:** Dati, kas piesaistīti ar vienas reizes datu piesaisti, neizraisa izmaiņas komponentēs, kam tie piesaistīti, kā arī izmaiņas komponentēs neizraisa izmaiņas datos.
Problēmgadījums: Augsts atmiņas patēriņš ar Knockout JS veidotai vienas lapas tīmekļa lietotnei. **Risinājums:** Nepieciešams pārbaudīt, vai visiem elementiem, kam nav nepieciešama automātiska lietotāja saskarnes atjaunināšana, ir veikta vienas reizes datu piesaiste. Dati, kam ir veikta vienas reizes datu piesaiste netiek balstīti uz novērojamajiem objektiem, un dati, kas nav novērojamie objekti, bet gan tīri JavaScript dati, aizņem ievērojami mazāk atmiņas. **Brīdinājumi:** Dati, kas piesaistīti ar vienas reizes datu piesaisti, neizraisa izmaiņas komponentēs, kam tie piesaistīti, kā arī izmaiņas komponentēs neizraisa izmaiņas datos, jo tam ir nepieciešami novērojamie objekti.

2. pielikums. **DOM Elementa noplūdes JavaScript kods**

Pilns kods pieejams: https://github.com/SinglePageApplicationPerformance/Test-Applications/tree/master/Memory%20leaks/DOM%20Node%20Leak

```
(function () {
    'use strict';

        var posts = [];
        for (var i = 0; i < 10000; i++) {
            posts.push({
                text: 'Lorem Ipsum has been the ' + i + ' industry\'s standard dummy text
ever since the 1500s, when an unknown printer took a galley of type and scrambled it to
make a type specimen book.'
            });
        }
            for (var i = 0; i < posts.length; i++) {
                var newListItem = document.createElement("li");
                var ListValue = document.createTextNode(posts[i].text);
                newListItem.appendChild(ListValue);
                document.getElementById("posts").appendChild(newListItem);
            }

    var list = document.getElementById("posts");
    document.getElementById('leakbutton').addEventListener("click", function () {
        list.remove();
    })
})();
```

3. pielikums. **Pilna elementa noplūdes JavaScript kods**

Pilns kods pieejams: https://github.com/SinglePageApplicationPerformance/Test-Applications/tree/master/Memory%20leaks/Whole%20Object%20Leak

```
(function () {
    'use strict';
    var attachTheListener = function(){
        var a1=[];
        var a2=[];
        for(var i=0; i<300000; i++){
            a1.push({name:"test"});
        }
        for(var i=0; i<300000; i++){
            a2.push({name:"test"});
        }
    var oObject={
        p1:a1,
        p2:a2
    }
    document.getElementById('leakbutton').addEventListener("click", function () {
        alert(oObject.p1[0].name);
    });
    }
    attachTheListener();

})();
```

4. pielikums. **setInterval atmiņas noplūdes JavaScript kods**

Pilns kods pieejams: https://github.com/SinglePageApplicationPerformance/Test-Applications/tree/master/Memory%20leaks/setInterval%20Leak

```
function removeOtherButton(){
        document.getElementById("leakbutton").remove();
        document.getElementById("removeinterval").disabled = false;
    }
(function () {
    var getData = function(){
        var posts = [];
        for (var i = 0; i < 100000; i++) {
            posts.push({
                firstName: 'Jake' + i,
                lastName: 'Smith' + i,
                text: 'Lorem Ipsum has been the ' + i + ' industry\'s standard dummy text
ever since the 1500s, when an unknown printer took a galley of type and scrambled it to
make a type specimen book.',
                favorites: i + 100,
                reposts: i + 50,
                image: "test" + i,
                age: i,
                id: i
                });
        }
    return posts;
    }
    document.getElementById('removeinterval').addEventListener("click", function () {
        clearInterval(window.testinterval);
        list.remove();
    });
```

```
    var leakExample=function(){
        var posts = getData();
        testinterval = setInterval(function() {
            var element = document.getElementById('leakbutton');
            //setInterval atstās atmiņā posts masīvu arī tad, kad izmantotais DOM elements
ar id 'leakbutton' būs noņemts
            if(element) {
                element.innerHTML = posts[Math.floor(Math.random() * 10)].id;
            }
        }, 1000);
    }
    leakExample();
})();
```

5. pielikums **Nejaušs globāls mainīgais**

Pilns kods pieejams: https://github.com/SinglePageApplicationPerformance/Test-Applications/tree/master/Memory%20leaks/Global%20Variable%20Leak

```
(function () {
    var getData = function(){
        var posts=[];
        for (var i = 0; i < 100000; i++) {
            posts.push({
                firstName: 'Jake' + i,
                lastName: 'Smith' + i,
                text: 'Lorem Ipsum has been the ' + i + ' industry\'s standard dummy text
ever since the 1500s, when an unknown printer took a galley of type and scrambled it to
make a type specimen book.',
                favorites: i + 100,
                reposts: i + 50,
                image: "test" + i,
                age: i,
                id: i
                });
        }
    return posts;
    }
    var leakExample=function(){
            posts = getData();//Šis kods izveido nevis lokālu mainīgo, bet window.posts
    }
    leakExample();
})();
```

6. pielikums. **Notikumu delegācija**

Pilns kods pieejams: https://github.com/SinglePageApplicationPerformance/Test-Applications/tree/master/Memory%20leaks/Event%20Delegation

```
books.addEventListener('click', function(e) {
        if (e.target.innerHTML==="10") {
            alert("10 clicked.");
        }
    }, false);
    for(var i=0; i<30000; i++){
        var listElement=document.createElement("li");
        listElement.innerHTML = Math.floor(Math.random()*10)+1;
        books.appendChild(listElement);
    }
```

7. pielikums. **Apļveida reference**

Pilns kods pieejams: https://github.com/SinglePageApplicationPerformance/Test-Applications/tree/master/Memory%20leaks/Circular%20Reference%20Leak

```
function removeOtherButton(){
        document.getElementById("leakbutton").remove();
    }
(function () {

    function leakExample() {
        var a1={
            name:"",
            list:[]
        };
        var a2={
            name:"",
            list:[]
        };
        for(var i=0; i<300000; i++){
            a1.list.push({name:"test"});
        }
        for(var i=0; i<300000; i++){
            a2.list.push({name:"test"});
        }
        a1.name = a2.list[0].name;
        a2.name = a1.list[0].name;
    }
    leakExample();

})();
```

8. pielikums. **Fragments no vairāku tīmekļa strādņu testa lietotnes**

Pilns kods pieejams: https://github.com/SinglePageApplicationPerformance/Test-Applications/tree/master/Web%20workers/Multiple%20web%20worker%20test

```
//10000 elementu garš masīvs
var arrToProcess = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1.....
function start() {
    var start = window.performance.now();
    var numberOfWorkers = parseInt(document.getElementById("numberOfWorkers").value);
    var intervals = arrToProcess.length / parseInt(numberOfWorkers);
    if (numberOfWorkers == 0 || !numberOfWorkers) {
        //10000 elementu garš masīvs
        var arrToProcess = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1.....
        console.time("zeroWorkers");
        var iSum = 0;
        for (var i = 0; i < arrToProcess.length; i++) {
            var elem = arrToProcess[i];
            for (var j = 0; j < arr.length; j++) {
                elem = elem * arr[j];
                elem = elem * 2;
                elem = elem / 2;
                elem = elem * 2;
                elem = elem / 2;
                elem = elem * 2;
                elem = elem / 2;
                elem = elem * 2;
                elem = elem / 2;
                elem = elem * 2;
                elem = elem / 2;
                elem = elem * 2;
                elem = elem / 2;
                elem = elem * 2;
                elem = elem / 2;
                elem = elem * 2;
                elem = elem / 2;
                elem = elem * 2;
                elem = elem / 2;
                elem = elem * 2;
                elem = elem / 2;
                elem = elem / 1;
            }
            iSum = iSum + elem;
        }
        console.timeEnd("zeroWorkers");
        console.log(iSum);
        var end = window.performance.now();
        alert("Using " + (numberOfWorkers ? numberOfWorkers : 0) + " web workers the task
was completed in " + (end - start) + " miliseconds.");
    } else {
        var self = this;
        var aDeferreds = [];
        var iResolved = 0;
        var iSum = 0;
        iEndOfLastInterval = 0;
        console.time("workerTest");
        for (var i = 0; i < numberOfWorkers; i++) {
            aDeferreds.push($.Deferred());
            var iStart = Math.floor(iEndOfLastInterval);
            var iEnd = arrToProcess.length % numberOfWorkers !== 0 && (i + 1) ===
numberOfWorkers ? (Math.floor(iEndOfLastInterval + intervals) + 1) :
Math.floor(iEndOfLastInterval + intervals);
            iEndOfLastInterval = iEnd;
```

```
            worker = new Worker("worker.js");
            worker.postMessage({
                arrToProcess: arrToProcess,
                start: iStart,
                end: iEnd
            });
            worker.onmessage = function(e) {
                iSum = iSum + e.data;
                aDeferreds[iResolved].resolve();
                iResolved++;
            };
        }
        $.when.apply($, aDeferreds).then(function() {
            var end = window.performance.now();
            console.timeEnd("workerTest");
            console.log(iSum);
            alert("Using " + numberOfWorkers + " web workers the task was completed in " +
(end - start) + " miliseconds.");
        });
    }

}
```

9. pielikums. **Fragments no liela apjoma datu apstrādes ar tīmekļa strādni**

Pilns kods pieejams: https://github.com/SinglePageApplicationPerformance/Test-Applications/tree/master/Web%20workers/Web%20worker%20huge%20data%20transfer

```
var arrToProcess = [];
$.getJSON("smallvalues.json", function(data) {
    arrToProcess = data;
});

function start() {
    var sScenario = "";
    if (document.getElementById('s1').checked) {
        sScenario = document.getElementById('s1').value;
    } else if (document.getElementById('s2').checked) {
        sScenario = document.getElementById('s2').value;
    }
    var useWorker = document.getElementById("useWorker").checked;
    if (!useWorker) {
        var arr = [1, 1, 1, 1, 1, 1, 1, 1, 1, 1];
        console.time("zeroWorkers");
        var iSum = 0;
        for (var i = 0; i < arrToProcess.length; i++) {
            var elem = arrToProcess[i].value;
            for (var j = 0; j < arr.length; j++) {
                elem = elem * arr[j];
                elem = elem * 2;
                elem = elem / 2;
                elem = elem * 2;
                elem = elem / 2;
                elem = elem * 2;
                elem = elem / 2;
                elem = elem * 2;
                elem = elem / 2;
                elem = elem * 2;
                elem = elem / 2;
                elem = elem * 2;
                elem = elem / 2;
                elem = elem * 2;
                elem = elem / 2;
                elem = elem * 2;
```

```
                elem = elem / 2;
                elem = elem * 2;
                elem = elem / 2;
                elem = elem * 2;
                elem = elem / 2;
                elem = elem / 1;
            }
            iSum = iSum + elem;
        }
        console.timeEnd("zeroWorkers");
        console.log(iSum);
    } else {
        console.time("workerTest");
        worker = new Worker("worker.js");
        if (sScenario === "stringify") {
            worker.postMessage({
                scenario: sScenario,
                arrToProcess: JSON.stringify(arrToProcess)
            });
        } else {
            worker.postMessage({
                scenario: sScenario,
                arrToProcess: arrToProcess
            });
        }
        worker.onmessage = function(e) {
            console.timeEnd("workerTest");
            console.log(e.data);
        };

    }

}
```

10. pielikums. **DOM manipulāciju testa lietotnes A3 scenārijs**

Pilns visu A scenāriju kods pieejams:

https://github.com/SinglePageApplicationPerformance/Test-Applications/tree/master/Rendering/Simple%20list%20items%20rendering

```
function runAppendChild() {
var iterations = parseInt(document.getElementById('listElementCount').value);
var div = document.createElement('div');
document.body.appendChild(div);
console.time("Div - appendChild " + iterations + "x");
for (var i = 0; i < iterations; i++) {
var el = document.createElement('li');
el.innerText = 'List item ' + i;
div.appendChild(el);
}
console.timeEnd("Div - appendChild " + iterations + "x");
}
```

11. pielikums. **DOM manipulāciju testa lietotnes B3 scenārijs**

Pilns visu B scenāriju kods pieejams:

https://github.com/SinglePageApplicationPerformance/Test-Applications/tree/master/Rendering/Twitter%20items%20rendering

```
function runAppendChild() {
    var iterations = parseInt(document.getElementById('listElementCount').value);
    var div = document.createElement('div');
    document.body.appendChild(div);
    console.time("Div - appendChild " + iterations + "x");
    for (var i = 0; i < iterations; i++) {
        var divInputWrap = document.createElement('div');
        var divMedia = document.createElement('div');
        var divBodyContainer = document.createElement('div');
        var divRow = document.createElement('div');
        var divMediaBody = document.createElement('div');
        var h4PostHeading = document.createElement('h4');
        var spanName = document.createElement('span');
        var aNickName = document.createElement('a');
        var spanTimeAgo = document.createElement('span');
        var pPostBody = document.createElement('p');
        var pPostRating = document.createElement('p');
        var divRow2 = document.createElement('div');
        var divBottomLinks = document.createElement('div');
        var aPostPropertyExpand = document.createElement('a');
        var aPostPropertyReply = document.createElement('a');
        var aPostPropertyRepost = document.createElement('a');
        var aPostPropertyStar = document.createElement('a');
        var aPostPropertyMore = document.createElement('a');
        divInputWrap.appendChild(divMedia);
        divMedia.appendChild(divBodyContainer);
        divBodyContainer.appendChild(divRow);
        divBodyContainer.appendChild(divRow2);
        divRow.appendChild(divMediaBody);
        divMediaBody.appendChild(h4PostHeading);
        divMediaBody.appendChild(pPostBody);
        divMediaBody.appendChild(pPostRating);
        h4PostHeading.appendChild(spanName);
        h4PostHeading.appendChild(aNickName);
        h4PostHeading.appendChild(spanTimeAgo);
        divRow2.appendChild(divBottomLinks);
        divBottomLinks.appendChild(aPostPropertyExpand);
        divBottomLinks.appendChild(aPostPropertyReply);
        divBottomLinks.appendChild(aPostPropertyRepost);
        divBottomLinks.appendChild(aPostPropertyStar);
        divBottomLinks.appendChild(aPostPropertyMore);
        divInputWrap.className = 'row input-wrap';
        divMedia.className = 'media';
        divBodyContainer.className = 'body-container';
        divRow.className = 'row';
        divMediaBody.className = 'media-body';
        h4PostHeading.className = 'post-heading';
        pPostBody.className = 'post-body';
        pPostRating.className = 'post-rating';
        divRow2.className = 'row';
        divBottomLinks.className = 'bottom-links';
        aPostPropertyExpand.className = 'post-property';
        aPostPropertyMore.className = 'post-property';
        aPostPropertyReply.className = 'post-property';
        aPostPropertyRepost.className = 'post-property';
        aPostPropertyStar.className = 'post-property';
```

```
        spanName.innerText = 'Jake Smith ' + i;
        aNickName.innerText = '@jsmith' + i;
        spanTimeAgo.innerText = i + ' minutes ago';
        pPostBody.innerText = 'Lorem Ipsum has been the ' + i + ' industry\'s standard
dummy text ever since the 1500s, when an unknown printer took a galley of type and
scrambled it to make a type specimen book.'
        pPostRating.innerText = 'Rating: ' + i;
        aPostPropertyExpand.innerText = 'Expand';
        aPostPropertyMore.innerText = 'More';
        aPostPropertyReply.innerText = 'Reply';
        aPostPropertyRepost.innerText = 'Repost (' + i + ')';
        aPostPropertyStar.innerText = 'Star (' + i + ')';
        div.appendChild(divInputWrap);
    }
    console.timeEnd("Div - appendChild " + iterations + "x");
}
```

12. pielikums **Animācija, izmantojot CSS īpašības top un left**

Pilns kods pieejams: https://github.com/SinglePageApplicationPerformance/Test-Applications/tree/master/Animations/Animating%20with%20top%20and%20left

```
.animate {
    position: absolute;
    top: 100px;
    left: 100px;
    animation: move 3s ease infinite;
}

@keyframes move {
    50% {
        top: 500px;
        left: 500px;
    }
}

.animatableDiv {
    box-shadow: inset 0 1px 1px 1px #000, inset 0 0 0 9px #111, 0 0 0 1px #ccc, 0 0 1px 2px
hsla(0, 0%, 0%, .4), 0 0 50px hsla(0, 0%, 0%, .3), 0 0 100px hsla(0, 0%, 0%, .5), 0 0 200px
hsla(0, 0%, 0%, .5);
    height: 100px;
    width: 100px;
    background-color: grey;
}
```

13. pielikums. **Animācija, izmantojot CSS īpašību tranform: translate**

Pilns kods pieejams: https://github.com/SinglePageApplicationPerformance/Test-Applications/tree/master/Animations/Animating%20with%20translate

```
.animate {
    position: absolute;
    top: 100px;
    left: 100px;
    animation: move 3s ease infinite;
}

@keyframes move {
  50% {
     transform: translate(400px, 400px);
  }
}

.animatableDiv {
    box-shadow: inset 0 1px 1px 1px #000, inset 0 0 0 9px #111, 0 0 0 1px #ccc, 0 0 1px 2px
hsla(0, 0%, 0%, .4), 0 0 50px hsla(0, 0%, 0%, .3), 0 0 100px hsla(0, 0%, 0%, .5), 0 0 200px
hsla(0, 0%, 0%, .5);
    height: 100px;
    width: 100px;
    background-color: grey;
}
```

14. pielikums. **Fiksēta fona CSS koda fragments, neizmantojot will-change**

Pilns kods pieejams: https://github.com/SinglePageApplicationPerformance/Test-Applications/tree/master/Scrolling/Fixed%20background%20element%20separate%20layer/Fixed%20background%20one%20layer

```
.fixedBackground {
  align-items: center;
  display: flex;
  justify-content: center;
  min-height: 100%;
  position: relative;
  background-size: cover;
  background: url('../images/fixedbackground.jpg') no-repeat center top fixed;
  -moz-background-size: cover;
  -o-background-size: cover;
  -webkit-background-size: cover;
  -moz-box-shadow:    inset 0 0 10px #000000;
  -webkit-box-shadow: inset 0 0 10px #000000;
  box-shadow:         inset 0 0 10px #000000;
}
```

15. pielikums. **Fiksēta fona CSS koda fragments, izmantojot will-change**

Pilns kods pieejams: https://github.com/SinglePageApplicationPerformance/Test-Applications/tree/master/Scrolling/Fixed%20background%20element%20separate%20layer/Fixed%20background%20separate%20layer

```
.fixedBackground {
  align-items: center;
  display: flex;
  justify-content: center;
  min-height: 100%;
  position: relative;
}

.fixedBackground::before{
    background-image: url('../images/fixedbackground.jpg');
    background-repeat: no-repeat;
    background-size: cover;
    -moz-box-shadow:    inset 0 0 10px #000000;
    -webkit-box-shadow: inset 0 0 10px #000000;
    box-shadow:         inset 0 0 10px #000000;
    content: '';
    height: 100%;
    left: 0;
    position: fixed;
    top: 0;
    width: 100%;
    will-change: transform;
    z-index: -1;
}
```

16. pielikums. **React JS saraksta komponente**

Pilns kods un salīdzinājuma lietotnes pieejamas: https://github.com/SinglePageApplicationPerformance/Test-Applications/tree/master/JavaScript%20Frameworks/React%20JS

```
class PostList extends React.Component {
    constructor(props) {
        super(props);
        this.onChangeOne = this.onChangeOne.bind(this);
        this.renderItems = this.renderItems.bind(this);
    }
    componentWillMount() {
        this.setState({ posts: this.props.posts });
    }
    onChangeOne(event) {
        var posts = this.state.posts;
        var iRandomIndex = Math.floor((Math.random() * posts.length));
        posts[iRandomIndex].favorites = 9999;
        this.setState({ posts: posts , numberOfItems: posts.length});
    }
    updateNumberOfItems(event){
        this.state.numberOfItems = event.target.value;
```

```
    }
    renderItems(){
      var arr = [];
        for (var i = 0; i < this.numberInput.value; i++) {
            arr.push({
                firstName: 'Jake' + i,
                lastName: 'Smith' + i,
                text: 'Lorem Ipsum has been the ' + i + ' industry\'s standard dummy text
ever since the 1500s, when an unknown printer took a galley of type and scrambled it to
make a type specimen book.',
                favorites: i + 100,
                reposts: i + 50,
                image: "test" + i,
                age: i,
                id: i
            });
        }
    this.setState({ posts:arr, numberOfItems: arr.length});
  }
    render() {
        console.log(this.state);
        return React.createElement(
            "div",
            null,

            React.createElement(
                "input",
                { ref: function(input) {
                this.numberInput = input;
           }.bind(this)}
            ),
            React.createElement(
                "button",
                { onClick: this.renderItems },
                "Display this number of items"
            ),
            React.createElement(
                "br",
                {}
            ),
            React.createElement(
                "button",
                { onClick: this.onChangeOne },
                "Change one item"
            ),
            this.state.posts.map(post => {
                    var sUserName = (post.firstName.slice(0, 1) +
post.lastName).toLowerCase();
                    var iRating = (post.favorites + post.reposts) / 50;
                    return React.createElement(
                        "div",
                        null,
                        React.createElement("hr", null),
                        React.createElement(ListItem, _extends({
                            sUserName: sUserName,
                            iRating: iRating
                        }, post))
                    );
                })
        );
    }
}

class ListItem extends React.Component {
  constructor(props) {
    super(props);
  }
  //shouldComponentUpdate improves performance
```

```
  shouldComponentUpdate(nextProps, nextState) {
    return nextProps.lastName !== this.props.lastName || nextProps.firstName !==
this.props.firstName || nextProps.image !== this.props.image || nextProps.favorites !==
this.props.favorites || nextProps.reposts !== this.props.reposts || nextProps.text !==
this.props.text;
  }
  render() {
    const { firstName, lastName, sUserName, iRating, reposts, favorites, text, image } =
this.props;
    return React.createElement(
      "div",
      { className: "row input-wrap" },
      React.createElement(
        "div",
        { className: "media" },

        React.createElement(
          "div",
          { className: "body-container" },
          React.createElement(
            "div",
            { className: "row" },
            React.createElement(
              "div",
              { className: "media-body" },
              React.createElement(
                "h4",
                { className: "post-heading" },
                firstName,
                " ",
                lastName,
                " ",
                React.createElement(
                  "a",
                  null,
                  "@",
                  sUserName
                ),
                React.createElement(
                  "span",
                  { className: "timeAgo" },
                  "  4 minutes ago"
                )
              ),
              React.createElement(
                "p",
                { className: "post-body" },
                text
              ),
              React.createElement(
                "p",
                { className: "post-rating" },
                "Rating: ",
                iRating
              )
            )
          ),
          React.createElement(
            "div",
            { className: "row" },
            React.createElement(
              "div",
              { className: "bottom-links" },
              React.createElement(
                "a",
                { className: "post-property", href: "#" },
                "Expand"
              ),
```

```
              React.createElement(
                "a",
                { className: "post-property", href: "#" },
                "Reply"
              ),
              React.createElement(
                "a",
                { className: "post-property", href: "#" },
                "Repost (",
                reposts,
                ")"
              ),
              React.createElement(
                "a",
                { className: "post-property", href: "#" },
                "Star (",
                favorites,
                ")"
              ),
              React.createElement(
                "a",
                { className: "post-property", href: "#" },
                "More"
              )
            )
          )
        )
      )
    );
  }
}
```

17. pielikums. **Knockout JS divu virzienu datu piesaistes koda fragments**

Pilns kods un salīdzinājuma lietotnes pieejamas:

https://github.com/SinglePageApplicationPerformance/Test-Applications/tree/master/JavaScript%20Frameworks/Knockout%20JS%20memory

```
ko.options.deferUpdates = true;
var ViewModel = function() {
    this.allItems = [];
    this.numberOfItems = 0;
    var posts = ko.mapping.fromJS([]);
    var self = this;
    self.allItems = posts;
    this.changeNumerOfDisplayedItems = function() {
        var arr = [];
        for (var i = 0; i < self.numberOfItems; i++) {
            arr.push({
                firstName: 'Jake' + i,
                lastName: 'Smith' + i,
                text: 'Lorem Ipsum has been the ' + i + ' industry\'s standard dummy text
ever since the 1500s, when an unknown printer took a galley of type and scrambled it to
make a type specimen book.',
                favorites: i + 100,
                reposts: i + 50,
                image: "test" + i,
                age: i,
                id: i
            });
        }
        self.allItems(ko.mapping.fromJS(arr).slice());

    }
    this.changeOne = function() {
        self.allItems()[Math.floor(Math.random() * self.numberOfItems)].favorites(9999);
    }
};

ko.applyBindings(new ViewModel());
```

18. pielikums**. Angular JS divu virzienu datu piesaistes koda fragments**

Pilns kods un salīdzinājuma lietotnes pieejamas:

https://github.com/SinglePageApplicationPerformance/Test-Applications/tree/master/JavaScript%20Frameworks/Angular%20JS%20performance

```
var performanceTestApp = angular.module('performanceTest', []);
var posts = [];
performanceTestApp.controller('postsController', function($scope) {
    $scope.numberOfItems = 0;
    $scope.posts = posts;
    $scope.changeOnePost = function() {
        var randomIndex = Math.floor(Math.random() * ($scope.posts.length - 1));
        $scope.posts[randomIndex].favorites = 9999;
    };
```

```
    $scope.changeNumerOfDisplayedItems = function() {
        var arr = [];
        for (var i = 0; i < $scope.numberOfItems; i++) {
            arr.push({
                firstName: 'Jake' + i,
                lastName: 'Smith' + i,
                text: 'Lorem Ipsum has been the ' + i + ' industry\'s standard dummy text
ever since the 1500s, when an unknown printer took a galley of type and scrambled it to
make a type specimen book.',
                favorites: i + 100,
                reposts: i + 50,
                image: "test" + i,
                age: i,
                id: i
            });
        }
        $scope.posts = arr;
    };
    $scope.posts = posts;


});
```

Bakalaura darbs „Vienas lapas tīmekļa lietotņu veiktspēja” izstrādāts LU Datorikas fakultātē.

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

Autors: _____ Gustavs Helmuts Felsbergs

Rekomendēju/nerekomendēju darbu aizstāvēšanai

Vadītājs: profesors, Dr. dat. Uldis Straujums _____ ___.05.2018.

Recenzents: asociētā profesore, Dr. dat. Darja Solodovņikova

Darbs iesniegts Datorikas fakultātē ___.05.2018.

Dekāna pilnvarotā persona:

vecākā metodiķe Ārija Sproģe _____

Darbs aizstāvēts bakalaura gala pārbaudījuma komisijas sēdē

_____

Komisijas sekretārs/e: _____