Zbornik radova Fakulteta tehničkih nauka, Novi Sad

UDK: 004.9
DOI: https://doi.org/10.24867/23BE03Jenjic

УТИЦАЈ *REACT LAZY* И *WEBPACK* КОНФИГУРАЦИЈА НА ПЕРФОРМАНСЕ АПЛИКАЦИЈЕ

IMPACT OF REACT LAZY AND WEBPACK CONFIGURATION ON APPLICATION PERFORMANCE

Јован Јењић, *Факултет техничких наука, Нови Сад*

Област – РАЧУНАРСТВО И АУТОМАТИКА

Кратак садржај – *У овом раду је представљен утицај одређених Webpack конфигурација и React lazy-a на перформансе апликација. Поред теоријских концепата који служе да би тема рада била јасна, овај рад прати развој веб апликације. Кроз развој апликације у неколико корака је приказан напредак у перформансама апликације што је документовано кроз одређене Google-ove метрике.*

Кључне речи: *Веб апликација, React, Webpack, lazy*

Abstract – *This paper presents the impact of certain Webpack configurations and React lazy on application performance. In addition to theoretical concepts that serve to make the topic of the work clear, this paper follows the development of a web application. Through the development of the application in several steps, the progress in the performance of the application is shown, which is documented through certain Google metrics.*

Keywords: *Web application, React, Webpack, lazy*

1. УВОД

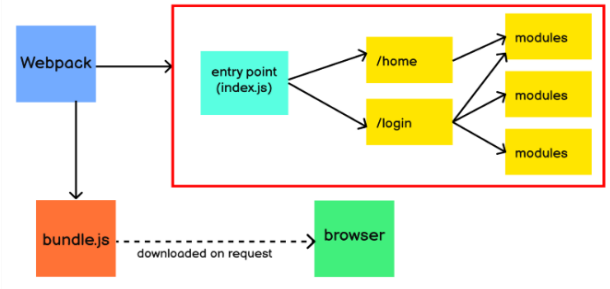
Овај рад се бави предностима које доносе *React* лењо учитавање (енг. lazy loading) и *Webpack* конфигурације у перформансама при имплементацији одређених веб апликација (енг. web application). Осврће се на модерне приступе у изради веб апликација, на предности појединих приступа у конфигурисању архитектуре веб апликација. Описани су појмови и дефиниције чије је разумевање неопходно. Обухваћене су основне теме који су уско повезане са темом овог рада. Обухваћени су концепти веб апликације, паковања (енг. bundling) и преузимање и извршавање апликације на интернет претраживачу (енг. web browser). Такође овај рад се бави и прегледом метрика и мерења перформанси веб апликација. Описано је чему је *Webpack* намењен, на чему се темељи и које су све могуће опције у конфигурисању *Webpack*-а у циљу израде што оптимизованијих апликација.

2. *WEBPACK*

Webpack је статички *bundler* модула за модерне *JavaScript* апликације. Ово је тренутно најзаступљенији алат за *bundling*. Преузима сав код из апликације и чини га употребљивим у веб претраживачу. Модули су комади *JavaScript* кода направљени за вишеструку употребу. Састављени су од кода, слика, екстерних билиотека које су увезене у апликацији или *CSS* стилова који су обједињени и упаковани како би се лако користили у веб претраживачу. *Webpack* одваја код на основу начина на који се користи у апликацији, а са особином модуларног расчлањивања одговорности постаје много лакши за управљати, отклањати грешке, верификовати и тестирати код.

У току *Webpack* обраде апликације он гради граф зависности од најчешће једне или више улазних тачака и затим комбинује сваки модул који је потребан апликацији у један или више пакета који су подељени у *chunk* датотеке. *Chunk* је посебна група повезаног кода који је компајлиран и трансформисан тако да га претраживач може покренути. Ако једна датотека директно зависи од друге, *Webpack* то третира као зависност. Слике, фонтови, стилови и друге ствари које се не кодирају директно у апликацији *Webpack* такође гледа као зависности [1]. Како би се процес *bundling*-а подешавао, *Webpack* користи плагине (енг. plugin) и учитаваче (енг. loaders).

Webpack пружа могућност за раздвајање кода (енг. code splitting) и лењо учитавање (енг. lazy loading) кода са уграђеним високо прилагодљивим плагинима. Такође подржава и дељење заједничких модула између различитих апликација, што посебно има значаја у *micro frontend* апликацијама [2]. На слици 1 је приказана шема креирања *bundle.js* датотеке на основу структуре кода подељених у фајлове где је улазна датотека приказана плавом бојом на слици.



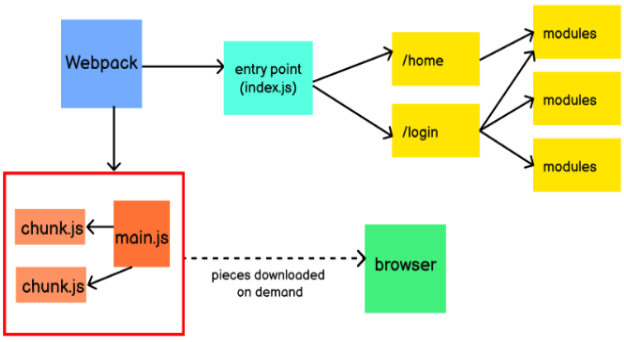
Слика 1. *Креирање више chunk датотека* [3]

НАПОМЕНА:
Овај рад проистекао је из мастер рада чији ментор је био др Александар Купусинац, ред. проф.

Са развојем апликације укупна величина изворног кода ће се повећати. Као резултат процеса *bundling*-а настаје датотека *bundle.js*. Претраживач мора у потпуности да преузме ову датотеку пре него што изврши код у њој, самим тим перформансе и брзина апликације опада са порастом величине ове датотеке, посебно ако су укључене *third-part* библиотеке.

Како би се решило дуго време учитавања које је изазвано великом *bundle.js* датотеком, *Webpack* тим је увео функцију раздвајања кода. Подела кода (енг. code splitting) омогућава да уместо једног великог *bundle.js* фајла, апликације буде подељена у неколико мањих датотека које претраживач може посебно и паралелно да преузима. Генерално, постојаће једна *bundle.js* датотека која ће се увек преузети при иницијалном учитавању сваке странице и постојаће неколико пратећих *bundle.js* фајлова, који су познати под називом *chunk.js* фајлови и они се учитавају на кориснички захтев за тим делом кода. Иако је укупна величина изворног кода остала иста, мања величина главне *bundle.js* датотеке ће убрзати почетно време учитавања и омогућиће бржу корисничку интеракцију са апликацијом а то директно води ка бољим перформансама.

Ако је *React* апликација креирана преко *Create React App (CRA)*, онда ће *Webpack* конфигурација која је генерисана од стране *CRA* подразумевано омогућити дељење кода, без додатног конфигурисања са стране програмера. Како у *React*-у постоје подразумеване конфигурације *Webpack*-а, тако да ће сваки динамични увоз који постоји у коду бити препознат са стране *Webpack*-а и за тај део увезеног кода ће бити креирана посебна *chunk* датотека [4]. На слици 2 је показана шема креирања више *chunk* датотека.



Слика 2. *Креирање више chunk датотека* [3]

Постоји више начина којом се постиже дељење кода. Један од начина је динамични увоз који користи синтаксу *import()*. Динамички увоз има другачију синтаксу од уобичајеног наичина за увоз неке датотеке или библиотеке. Када *Webpack* препозна синтаксу за динамични увоз он аутоматски покреће процес раздвајања кода.

Динамички увоз враћа *Promise* који се може користити само асинхроно. Уколико се не појави ниједна грешка, *Promise* ће да врати компоненту или функцију, у супротном ће вратити грешку коју можемо да ухватимо [5]. На слици 3 показан је пример динамичног увоза неке *text.js* датотеке. Коришћена је *async await* синтакса за динамички увоз.

```
```
// dynamic.js - import with async await
const loadDynamicImport = async () => {
  const blogContent = await import("./text.js");
  const str = blogContent();
  console.log(str);

  /* Output:
  Lorem ipsum dolor sit amet, consectetur adipiscing
  */
}
loadDynamicImport();
```
```

Слика 3. *Пример дунамичког увоза text.js компоненте* [3]

React лењо учитавање је техника оптимизације веб странице или веб апликације. Такође је позната као учитавање на захтев јер учитава само садржај видљив на екранима корисника. Када корисник посети неку веб страницу, уместо да учита целу страницу, учитава се само део странице. Затим лењо улитавање одлаже преузимање преосталог садржаја веб странице све док корисник не затражи тај део. На пример, ако веб страница садржи слику који није у фокусу након иницијалног учитавања странице, већ корисник мора да скролује доле да би видео слику, слика се учитава тек када корисник стигне до места где се налази слика. У *React* апликацијама поред слика могуће је лењо учитати и друге ресурсе, као на пример код. Заправо је *React* направио да лењо учитавање делова кода буде олакшано, тако што је омогућио да свака компонента буде стављена у посебан *chunk* фајл [5].

React lazy је релативно нова функција у *React*-у која омогућава лењо учитавање и дељење кода без коришћења неке *third-part* библиотеке као што је то био раније случај. Сада постоји *React lazy* која је интегрисана функција у саму основу *React* библиотеке. Ово уграђена функција омогућава да се лако изврши лењо учитавање компоненти.

Основна разлика између динамичног лењог учитавања и обичног увоза је у томе што у случају лењог учитавања се увози компонента или неки ресурс тек када се тај део захтева од стране корисника.

На слици 4 приказано је лењо учитавање компоненте које зависи од неког услова. Другим речима, *Webpack bundler* је препознао динамични увоз и креирао посебан *chunk.js* фајл за њега. Тај фајл ће шретраживач почети да преузима и обрађује так некон што услов буде испуњен. Услов најчешће буде испуњен као реакција на неку корисничку акцију клика и слично [5].

```
let OtherComponent = undefined;
// Never imported.
if (false) {
  OtherComponent = React.lazy(() => import('./OtherComponent.jsx'));
}
```

Слика 4. *Креирање више chunk датотека* [3]

Главна предност лењог учитавања су побољшавање перформанси. Учитавањем мањег *JavaScript* фајла претраживач ће смањити време учитавања *DOM*-а и побољшаће перформансе апликације. Корисници могу да приступе веб страници чак иако није све учитано. Неке од предности лењог учитавања [5]:

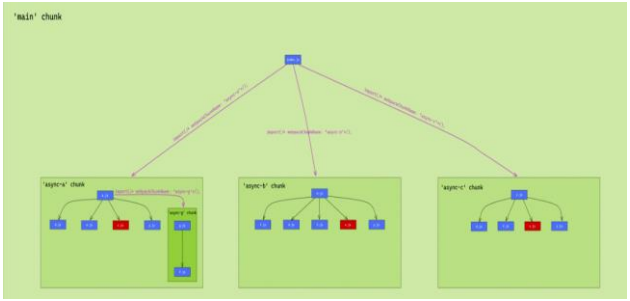
- **Брже почетно учитавање** - Лењо учитавање помаже да се смањи тежина странице се погледа величине, омогућавајући брже почетно учитавање.
- **Боље корисничко искуство** - Лењо учитавање побољшава корисничко искуство у апликацији. Добро корисничко искуство директно води ка повећању пословања и присутности корисника на апликацији, задржавајући посетиоце.
- **Мања потрошња пропусног опсега** - Лењо учитавање слика помаже у уштеди података и пропусног опсега. Ово посебно утиче на кориснике који немају брз интернет или имају слаб *CPU*.
- **Очување системских ресурса** - Лењо учитавање *React* компоненти помаже у очувању серверских и клијентских ресурса тако што захтева само делове читаве компоненте.
- **Смањени рад претраживача** - Претраживач не мора да обради слике или велики део кода све док слике или делови кода не буду захтевани од стране корисника као акција клика, скроловања или других акција.

SplitChunkPlugin аутоматски инфентификује модуле који би требали бити подељени у *chunk*-ove помоћу хеуристике користећи број понављања модула и категорије којима модули припарају.

Проблем који *SplitChunkPlugin* решава је дуплирање истог кода, што може довести до сувишног садржаја који се учитава преко мреже. *SplitChunkPlugin* је у стању да открије, на основу неких правила, модуле који су прескупи да би били дуплирани, затим их издваја у посебне *chunk* фајлове тако да се велики напор учитавања захтевних модула дешава само једном у оквиру једног *chunk* фајла. На пример, ако анализирањем *bundle*-а закључимо да се нека велика компонента понавља у више *chunk* фајлова, процес учитавања за сваки од тих *chunk* фајлова може да потраје дуго као и парсирање кода које следи након учитавања. Понављање овог скупог процеса није добро решење са становишта перформанси и неопходно је извршити одређене конфигурације *Webpack*-а како би се ово избегло. Управо такве ситуације решава овај плагин.

Дијаграм на слици 5 приказује 3 различита *chunk* фајла који су креирани на основу динамичког увоза. Такође, шематски је приказан садржај сваког *chunk* фајла, тј. који сви модули се налазе у њему. Занимљиво је приметити да постоји *x.js* датотека која се понавља у сва три *chunk* фајла. Као што је раније било споменуто, ово је случај који негативно утиче на перформансе веб апликације посебно ако је *x.js* нека велика и скупа датотека. Под претпоставком да *x.js* има 1000 линија кода, претраживач ће морати да учита *x.js* три пута јер је *x.js* део три различита *chunk* фајла. Узимајући у обзир ове информације, начин за решавање споменутог проблема би био да *x.js* буде смештен у посебан *chunk* фајл при чему би се овај садржај захтевао и обрађивао једном, уместо три пута.

Webpack-license-plugin издваја информације о лиценцама отвореног кода (енг. open source code) свих *npm* пакета у излазном *webpack* фајлу и помаже при идентификовању и решавању проблема са политиком лиценцирања отвореног кода. Овај *plugin* је подржан и тестиран у верзијама 2, 3, 4 и 5 *webpack*-а [6].



Слика 5. *Креирање више chunk датотека* [3]

3. ИМПЛЕМЕНТАЦИЈА

Приказане су и упоређене перформансе апликације пре и након укључивања лењог учитавања и након додатних *Webpack* конфигурација. За мерење перформанси ће бити коришћене основне *Google*-ове меторике из *Lighthouse* резултата.
За развијање *front-end* дела система коришћен је *Visual Studio Code* интегрисано развојно окружење и *React JavaScript* библиотека верзије 17.0.2 заједно са *Webpack*-ом који групише *JavaScript* фајлове и омогућава модуларну имплементацију система заједно са својим плугин решењем.

Сама апликација садржи неколико страница. *React* рутер (енг. Router) служи за навигирање између страница, свака страница се налази на посебној путањи и захтева посебан део кода који претраживач мора да учита како би страница била приказана, као што је приказано на слици 4.1. Свака страница садржи одређени део имплементиране логике у виду кода и увезене *third-part* библиотеке. Неке од библиотека се понављају неколико пута, тј. увезене су исте библиотеке у различитим страницама апликације. Циљ овог дела је да кроз неколико корака покаже напредак у перформансама апликације без превише детаљног залажења у само кодирање апликације.

Први корак садржи информације о апликацији без додавања лењог учитавања и без додатних *Webpack* конфигурација. Коришћен је стандардан начин увоза свих датотека. *JavaScript* код читаве апликације ће бити преузет од стране претраживача на иницијалном учитавању било које странице. Претраживач је у овој фази преузео пет *JavaScript* датотека које у укупном збиру чине приближно 2300kb.

Потребно је запазити да приликом навигирања на било који другу странице претражач неће преузети ниједан нови *JavaScript* фајл, јер је сав потребан *JavaScript* код преузет на првом иницијалном учитавању апликације. Жељени исход да претраживач преузме само довољан код који је потребно да изврши да би тренутна страница била видљива и функционална. *Lighthouse* резултата који је постигнут у овој фази развоја апликације је 76 бодова од максималних 100.

У другом кораку изкоришћена је предност подразумеваних *Webpack* конфигурација и уместо

уобичајеног начина увоза свих страница апликације у рутеру, коришћено је лењо учитавање који *Webpack* конфигурације препознају и аутоматски деле код на мање *chunk*-ове у зависности од сваке странице. Након поновног покретања апликације приликом иницијалног учитавања исте странице за разлику од претходне ситуације где је постојао само један *chunk* фајл, сада постоји осам мањих *chunk* фајлова. Након иницијалног учитавања странице, приликом навигирања на сваку следећу страницу апликације, претраживач преузима нове *chunk* фајлове и кешира их у своју меморију. Ово показује да свака страница која је лењо учитава преко рутера садржи свој део кода који се учитава тек приликом навигирања на ту страницу, а не иницијално. Укупан збир свих преузетих *JavaScript* датотека сада је приближно 1200 kb. *Lighthouse* резултат мерења је побољшан са 76 на 91 бод у овој фази.

Приликом анализирања садржаја сваког *chunk* фајла закључено је да постоји више *chunk* фајлова који садрже исту *third-part* библиотеку која долази из *node-module*-а. Другим речима различити *chunk* фајлови који ће бити преузети и извршени од стране претраживача на различитим страницама апликације садрже исти садржај. У следећем кораку имплементације оптимизованог решења ове апликације коришћене су додатне *Webpack* конфигурације. Како би било избегнута ситуација где се исти *JavaScript* код непотребно понавља на више места, циљ је креирати посебан *chunk* фајл који ће садржати баш такве делове ове апликације. На слици 6 је показан *cacheGroups* опција која је конфигурисана у оквиру *splitChunks* плагина. Са овом конфигурацијом омогућено је да сви делови већ креираних *chunk*-ова који се понављају у више од 3 *chunk* фајла, буду убачени у посебан *common chunk* фајл који ће се асинхроно учитати само једном, при иницијалном улитавању било које странице. На овај начин је решен проблем понављања истог садржаја у више различитих *chunk* фајлова.

```
splitChunks: {
  chunks: 'all',
  name: false,
  cacheGroups: {
    common: {
      name: 'common',
      minChunks: 3,
      chunks: 'async',
      priority: 10,
      reuseExistingChunk: true,
      enforce: true
    }
  },
},
```

Слика 6. *CacheGrpups konfiguracija*

Приликом даљег испитивања садржаја сваког *chunk* фајла закључено је да више нема истог садржаја који се понавља. Такође, овим је постигнуто додатно смањање укупне величине свих *JavaScript* датотека које је потребно да буду преузете и извршене од стране претраживача на истој страници. Након поновног тестирања перформанси веб апликације, очековано је постигнут нови напредак у перформансама. *Lighthouse* резултат мерења је побољшан са 91 на 94 бод у овој фази.

5. ЗАКЉУЧАК

У првом кораку имплементације приказано је неоптимизовано решење који је потпуно функционално, али кроз мерење перформанси и величину *JavaScript* кода који претраживач мора да преузме и изврши, било је приказано да постоји простор за побољшавање перформанси. У сваком кораку имплементације акценат је био на метрикама које су мериле различите приступе у мерењу перформанси апликације, као и осврtање на конкретан узрок који доводи до бољих или лошијих перформанси. Сваки следећи корак у имплементацији је уводио нове ствари које поспешују рад апликације и то је мерењем *Lighthouse* метрикама и документовано.

У корацима имплементације редом је укључено лењо учитавање а затим и додатне *Webpack* конфигурације где су перформансе апликације итеративно напредовале. Резултат је био функционална апликације високих перформанси. Иако у делу имплементације показано да је и на малим апликацијама могуће значајно побољшати перформансе апликације, важно је запазити да постоје и друге *Webpack* опције које могу да утичу на перформансе као и да споменуте опције могу дати различите резултате у зависности од постављених параметара.

6. ЛИТЕРАТУРА

[1] James Thomas, *What is Wabpack?*, 2019. Доступно: https://levelup.gitconnected.com/what-is-webpack-4fdb624597ae

[2] Morgan Persson, *JavaScript DOM Manipulation Performanse, Comparing Vanilla JavaScript and Leading JavaScript Front-end Frameworks*, 2020. Доступно: https://www.theseus.fi/bitstream/handle/10024/345959/Laurila_Sonja.pdf?sequence=2&isAllowed=y

[3] https://parceljs.org/features/scope-hoisting/

[4] Nathan Sebhastian, *Understanding Webpack's Code Splitting Feature*, 2021. Доступно: https://blog.bitsrc.io/understanding-webpacks-code-splitting-feature-3077768e4066

[5] https://www.copycat.dev/blog/react-lazy/

[6] https://www.npmjs.com/package/webpack-license-plugin

Кратка биографија:

Јован Јењић је рођен у Бањој Луци 1997. године. Основне академске студије завршио је 2020. године на Факултету техничких наука у Новом Саду. Мастер рад на Факултету техничких наука из области Рачунарство и аутоматика – Електронско пословање одбранио је 2022. године