Зборник радова Факултета техничких наука, Нови Сад

UDK: 4.3
DOI: https://doi.org/10.24867/32BE24Bjelica

ИНТЕРНИ ЈЕЗИК ЗА ОПТИМИЗАЦИЈУ ПРОЦЕСА РАЗВОЈА СИСТЕМА СА МИКРОСЕРВИСНОМ АРХИТЕКТУРОМ

INTERNAL LANGUAGE FOR OPTIMIZING DEVELOPMENT PROCESSES IN SYSTEMS WITH MICROSERVICE ARCHITECTURE

Лука Бјелица, *Факултет техничких наука, Нови Сад*

Област – ЕЛЕКТРОТЕХНИКА И РАЧУНАРСТВО

Кратак садржај – *Развој микросервиса трансформисао је начин на који се софтвер дизајнира, развија и распоређује, подстичући модуларност, скалабилност и независност компоненти крајњег система. Међутим, овај приступ доноси и изазове, посебно у контексту ефикасности, сложености и потребне количине ресурса за развој. У овом раду језици специфични за домен (ЈСД) [1] и генератори кода су представљени као решење за ове изазове, омогућавајући бржи развој прилагођен специфичним потребама пројекта.*

Кључне речи: *Микросервисна архитектура, Језици специфични за домен, Генератори кода*

Abstract – *The development of microservices has transformed the way software is designed, developed and deployed, encouraging modularity, scalability and independence of end-system components. However, this approach also brings challenges, especially in the context of efficiency, complexity and required amount of development resources. In this paper, domain-specific languages (DSLs) [1] and code generators are presented as a solution to these challenges, enabling faster development tailored to specific project needs.*

Keywords: *Microservice Architecture, Domain Specific Languages, Code Generators*

1. УВОД

Микросервисна архитектура омогућава развој система у облику независних сервиса који комуницирају преко дефинисаних интерфејса [2], решавајући недостатке монолитне архитектуре као што су неефикасно скалирање и повећан ризик отказивања [3]. Предности укључују независно скалирање, поновно коришћење компоненти и смањени ризик системских кварова. Међутим, ова архитектура уводи сложености које захтевају значајне ресурсе за имплементацију и одржавање.

Језици специфични за домен (ЈСД), попут SQL [4], нуде виши ниво апстракције и аутоматизацију, што у контексту микросервиса олакшава развој и одржавање стандардизацијом и смањењем ручног рада.

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

Генератори кода [5] додатно аутоматизују развој, користећи шаблоне за креирање комуникационих интерфејса и других компоненти. Овај процес смањује ризик од грешака и поједностављује одржавање система, иако изазови настају код регенерације кода након мануелних измена, због могућих конфликата или губитка података [5].

2. ПРЕГЛЕД СТАЊА У ОБЛАСТИ

Постојеће алатке за развој софтвера углавном решавају техничке изазове микросервисне архитектуре [3], али не и сам процес развоја, што отвара простор за нове приступе који убрзавају итерације, адаптацију и имплементацију функционалности. Алатке за брз развој [6] омогућавају краће време од концепта до реализације, побољшавају скалабилност и смањују трошкове одржавања. Модуларна архитектура и виши ниво апстракције чине системе прилагодљивим и ефикасним у реаговању на промене у захтевима или технологијама. Важни фактори при избору алатке укључују њену флексибилност, подршку, документацију и могућност интеграције са другим системима, како би успешно одговарала специфичним потребама пројекта.

3. ДИЗАЈН РЕШЕЊА

3.1. Циљеви

Циљ предложеног решења јесте да омогући флексибилан и брз начин за имплементацију софтверских решења вођених принципима микросервисне архитектуре, подстичући кориснике да развијају систем прилагођен тренутним потребама и ограничењима, уз праћење добрих пракси и софтверских образаца. Додатно, предложено решење олакшава дефинисање и одржавање комуникационих интерфејса између различитих модула, односно сервиса, осигуравајући конзистентност целокупног система.

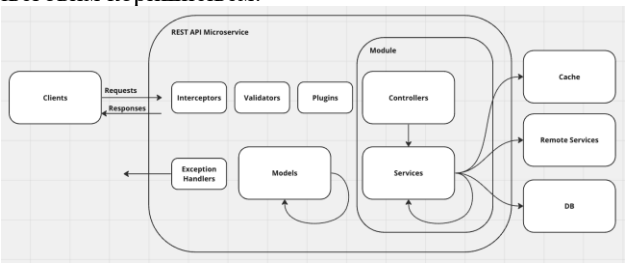
Предложено решење мора да испуни следеће захтеве:

1. Да буде разумљиво и ефикасно за коришћење програмеру
2. Да буде у што већој мери опште, односно да у што мањој мери буде везано за специфичну технологију или платформу

3. Да прикупи довољну количину података потребних за постизање жељеног нивоа аутоматизације
4. Да процес модуларизације, односно поделе решења на више мањих целина, буде брз и ефикасан
5. Да редукује ризик настанка грешака приликом повезивања различитих модула, односно делова система

3.2. Концепти

Илустрација 1 приказује концепте предложеног решења, постојеће концепте микросервисне архитектуре и ток извршавања система развијених његовим коришћењем.



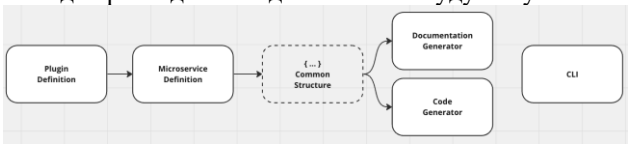
Слика 1. *Концепти*

Предложено решење уводи следеће концепте:

1. **Клијент**: Апликација која користи функционалности микросервиса путем API-а [7], обично преко HTTP [8] протокола.
2. **Пресретач захтева**: Компонента која обрађује захтеве пре или после обраде, користи се за аутентификацију, логовање, и примену безбедносних политика.
3. **Валидатор**: Проверава исправност и комплетност података у захтевима, смањује грешке током обраде и штити базу од невалидних података.
4. **Мапирање изузетака**: Управља грешкама током извршавања, логује детаље, обавештава системе за мониторинг и обезбеђује стабилност система.
5. **Додатак**: Компонента која се динамички додаје ради проширења или модификације функционалности, омогућава флексибилност и лакше ажурирање система.
6. **Модел**: Дефинише структуру и организацију података унутар система, укључујући базе података и екстерне системе.
7. **Модул**: Група компоненти (модел, сервис, контролер) која заједно обавља одређену функционалност у систему, омогућавајући бољу модуларност и скалабилност.
8. **Контролер**: Управља захтевима и одговорима, повезује клијента са пословном логиком и осигурава исправну обраду захтева.
9. **Сервис**: Независна функционална јединица која пружа одређене функционалности, комуницира с другим сервисима преко дефинисаних интерфејса, било локално или удаљено.

3.3. Архитектура

Архитектура предложеног система је дизајнирана тако да претходно наведени захтеви буду испуњени.



Слика 2. *Архитектура*

На Илустрација 2 приказана је архитектура предложеног система, са свим компонентама и везама између њих.

Системске компоненте се деле на компоненте за дефинисање и имплементацију решења, компоненте за аутоматизацију и компоненту за извршавање команди. Компонента за дефинисање и развој микросервиса представља улаз података у систем и омогућава прикупљање информација неопходних за функционисање крајњег решења и додатака. Компонента за развој додатака омогућава тимовима да имплементирају и дистрибуирају додатке, користећи јавне и приватне делове система, уз механизам за интеграцију током животног циклуса. Овај приступ подстиче заједнички развој и флексибилност решења.

Компонента за генерисање кода обавља аутоматизацију креирањем клијентског или серверског кода на основу заједничког језика, уз подршку за различите платформе и механизме за проширење. Проблем експоненцијалног раста броја генератора решава се увођењем заједничке структуре која омогућава превођење између платформи са минималним напором. Додатна аутоматизација обухвата генерисање документације на основу метаподатака. Компонента за извршавање команди омогућава интеракцију корисника са алатком, чинећи систем интуитивним и ефикасним за коришћење.

4. ИМПЛЕМЕНТАЦИЈА

4.1. Технологије

Предложено решење, у које спадају радни оквир, интерфејс командне линије и генератори кода, имплементирани су у JavaScript [9] програмском језику. Поред тога, коришћене су и следеће технологије: TypeScript [14], Webpack [10], Nunjucks [11], и PEG.js [12]. TypeScript је језик чија граматика представља надскуп JavaScript граматике, с циљем да омогући експлицитно дефинисање типова и интерфејса. Webpack је алатка којом се дефинисани пројекат у JavaScript језику пакује у један или више излазних датотека које зовемо пакет. Nunjucks је алатка која представља обрађивач шаблона, и коришћењем његовог уграђеног JCД-а могуће је генерисати динамичан садржај, у различитим окружењима. PEG.js је JavaScript библиотека за генерисање парсера, која омогућава дефинисање сложених граматика помоћу PEG нотације [1].

Коришћење Webpack-а омогућило је покретање оквира у било ком окружењу, док се CLI и генератор кода користе у терминалу током развоја. Nunjucks је

олакшао имплементацију и одржавање генератора, а PEG.js побољшао контролу над генерисањем кода са корисничким изменама.

4.2. Заједнички језик

Заједнички језик је имплементиран као интерни JCД унутар JSON [13] нотације. Знајући да се предложено решење користи углавном у контексту веб апликација, парсирање и серијализација су аутоматски подржани. Садржи два коренска поља, односно да складишти дефиниције сервиса и дефиниције модела. Сервиси се складиште као листа, док су типови ускладиштени као мапа која садржи парове кључ (назив модела) и вредност (дефиниција модела).

Дефиниција сервиса је сачињена од назива, коренске путање, и листе сервисних акција, док је дефиниција модела такође мапа која садржи дефиниције поља са паровима кључ (назив поља) и вредност (тип поља).

Дефиниција сервисне акције садржи HTTP методу, назив акције, путању, листу параметара које очекује, као и дефиницију или референцу на тип, односно модел резултата који враћа.

Дефиниција параметра садржи његов назив, дефиницију или референцу на тип односно модел, и извор одакле параметар долази (путања, секција за упит у путањи или тело захтева).

4.3. Дефинисање микросервиса

За исправан рад предложеног решења, корисник мора приложити довољно података за формирање структуре микросервиса у формату заједничког језика. Понашање контролера дефинише се као JavaScript класа, користећи механизме тог језика и подсистем за руковање зависностима. Структура интерфејса, која може бити екстерна (према клијентима) или интерна (унутар система), дефинише се механизмима предложеног решења заснованим на TypeScript декораторима, уз преклапања попут типова повратних вредности и параметара акција, што обезбеђује усклађеност између интерних и екстерних интерфејса.

4.4. Разрешавање графа зависности

Подсистем за руковање зависностима у предложеном решењу обухвата механизме за дефинисање и разрешавање графа зависности, укључујући и подршку за цикличне зависности. Проблем цикличних зависности, где компоненте међусобно зависе једна од друге, решава се аутоматски, користећи механизам референци. Овај процес се одвија у две фазе: у првој фази креирају се празне инстанце са референцама уместо стварних зависности, а у другој фази референце се замењују конкретним инстанцама, чиме се разрешавају све зависности у графу. Овим приступом се уклања одговорност са корисника, за разлику од других алатки које пријављују грешку при детекцији цикличних зависности и очекују од корисника да прилагоди систем [15].

Пошто JavaScript не подржава механизам референци, предложено решење ослања се на Proxy API за симулацију референци. Овај приступ обезбеђује флексибилност и ефикасност у раду са сложеним графовима зависности, укључујући и оне са цикличним везама. У случајевима сложенијих графова зависности, примењује се стратегија "подели и завладај" како би се проблем рашчланио на мање, лакше решиве делове. Ово чини предложено решење скалабилним и прилагођеним за комплексне системе [15].

4.5. Руковање додацима

Руковање додацима у предложеном решењу обухвата дефинисање, дистрибуцију и интеграцију, са фокусом на минимизирање напора за одржавање подсистема. За дистрибуцију се користи NPM [16], који је добро познат развојним тимовима и олакшава процес изградње и постављања пакета. Додатно, користи се семантичко верзионисање за праћење верзија, док JSON формат дефинише метаподатке пакета, укључујући назив, верзију и зависности, што обезбеђује транспарентност и лако управљање додацима.

Имплементација додатака ослања се на циљану технологију и дефинисани интерфејс који омогућава интеракцију са свим компонентама система током различитих фаза животног циклуса. Овај приступ пружа флексибилност и омогућава додацима да прошире функционалност система, док NPM инфраструктура и флексибилан интерфејс доприносе скалабилности и једноставнијој интеграцији нових функционалности у систем [16].

4.6. Генератор кода

Генератор кода у предложеном решењу омогућава аутоматизацију кроз генерисање кода на основу метаподатака који делују као заједнички језик, користећи механизам шаблонирања заснован на JavaScript библиотеци Nunjucks. Наведени приступ подржава поновну искористивост функционалности, јер се специфичности циљане технологије или платформе дефинишу у шаблону, користећи исте податке које генератор пружа. Шаблони се ослањају на Nunjucks синтаксу и организовани су у секције попут увоза зависности и дефиниције контролера, а подаци за шаблонирање се обрађују од стране генератора. Тренутно решење подржава генерисање клијентског кода за JavaScript, TypeScript и C# [17], док за серверски код подржава JavaScript и TypeScript.

Пројектна структура генератора организована је модуларно, где свака циљана технологија представља засебан модул који дефинише и имплементира шаблоне за генерисање кода. Оваква организација омогућава независност модула и поновну искористивост, док увођење подршке за нову технологију захтева креирање новог модула и имплементацију одговарајућих шаблона. Уколико је потребно, могу се имплементирати потпуно нови механизми, уз услов да се модул уклопи у очекивани интерфејс и региструје у решењу. Овај приступ

подржава развој прилагођених генератора и проширење решења са новим технологијама [17].

5. ГЕНЕРИСАЊЕ КОДА НАД ПОСТОЈЕЋИМ

Проблем одржавања баланса између аутоматски генерисаног кода и кориснички прилагођеног дела решен је имплементацијом PEG парсера, који омогућава очување корисничког кода приликом поновног генерисања. Овај приступ није везан за механизме специфичне технологије и не пребацује одговорност на корисника, већ користи PEG.js библиотеку за препознавање и екстракцију секција у којима се налази кориснички код, као што су тела метода у класи контролера или сервиса. Парсер је имплементиран за TypeScript и JavaScript језике, са фокусом на очување текста секција у оригиналном облику, што минимизује сложеност и ресурсе потребне за имплементацију и одржавање.

Додатне технологије могу се подржати креирањем нових парсера, што додатно умањује неопходан труд. На овај начин, предложено решење задовољава захтеве ефикасности и независности од специфичних технологија, док истовремено омогућава корисницима интуитиван начин за интеграцију прилагођеног кода у процес аутоматизације, без нарушавања функционалности или очекивања [17].

6. ЗАКЉУЧАК

Предложено решење је алат за брзи развој система базираних на микросервисној архитектури, са циљем оптимизације процеса развоја и прилагођавања развојним тимовима. Његова срж је интерни ЈСД који омогућава лакшу дефиницију интерфејса и структуре микросервиса, чиме се одређени степен аутоматизације, као што су генерисање кода и документације. Решење такође укључује механизам за додавање и управљање додацима, чиме се омогућава проширење функционалности на интуитиван начин.

Иако је флексибилно, тренутно подржава само три технологије, што ограничава његову примену као општег решења за оптимизацију процеса развоја. Ово представља изазов за тимове који користе више различитих технологија. Значајна предност је интуитиван начин очувања корисничког кода чак и након поновног генерисања, што смањује одговорност корисника и минимизује ресурсе потребне за одржавање.

Даљи развој усмерен је на унапређење општости и поузданости решења кроз подршку за више технологија и развој додатака у оквиру отвореног кода. Увођење сложенијих тестова и преписивање генератора документације као додатка помогло би у смањењу грешака и побољшању система, чиме би се приближило циљу да постане опште и поуздано решење за развој микросервисних система.

7. ЛИТЕРАТУРА

[1] I. Dejanović, Jezici specifični za domen, Novi Sad: Fakultet tehničkih nauka, 2021.

[2] J. Thönes, „Microservices,“ IEEE Software, т. 32, pp. 116-116, 2015.

[3] O. a. M. P. Al-Debagy, „A comparative review of microservices and monolithic architectures,“ IEEE, 2018.

[4] C. Fehily, SQL, CET, 2005.

[5] S. K. a. J.-P. Tolvanen, Domain-Specific Modeling: Enabling Full Code Generation, Wiley-IEEE Computer Society Pr, 2008.

[6] V. Silverthorne, „Rapid Application Development (RAD),“ TechTarget, April 2019. Доступно: https://www.techtarget.com/searchsoftwarequality/definition/rapid-application-development. [Последњи приступ Јун 2024].

[7] K. Lane, „Intro to APIs: History of APIs,“ Postman, 10 October 2019. Доступно: https://blog.postman.com/intro-to-apis-history-of-apis/. [Последњи приступ Јун 2024].

[8] J. G. J. M. H. F. L. M. P. L. T. B.-L. R. Fielding, „Hypertext Transfer Protocol -- HTTP/1.1,“ W3C/MIT, June 1999. Доступно: https://www.w3.org/Protocols/rfc2616/rfc2616.html. [Последњи приступ Јун 2024].

[9] M. F. K. G. Shu-yu Guo, „ECMAScript Language Specification,“ June 2024. Доступно: https://tc39.es/ecma262/. [Последњи приступ Јун 2024].

[10] P. C. Eugene Hlushko, „Webpack,“ 2024. Доступно: https://webpack.js.org/. [Последњи приступ Јун 2024].

[11] Mozilla, „Nunjucks,“ 2024. Доступно: https://mozilla.github.io/nunjucks/. [Последњи приступ Јун 2024].

[12] D. Majda, „PEG.js,“ 2019. Доступно: https://github.com/pegjs/pegjs. [Последњи приступ Јун 2024].

[13] Ecma International, „ECMA-404 The JSON Data Interchange Syntax,“ December 2017. Доступно: https://ecma-international.org/wp-content/uploads/ECMA-404_2nd_edition_december_2017.pdf. [Последњи приступ Јун 2024].

[14] Microsoft, „TypeScript,“ 2024. Доступно: https://www.typescriptlang.org/. [Последњи приступ Јун 2024].

[15] Mozilla, „Proxy API JavaScript,“ MDN Web Docs, 2024. Доступно: https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Proxy. [Последњи приступ Јун 2024].

[16] I. Z. Schlueter, „Node Package Manager,“ January 2010. Доступно: https://www.npmjs.com/. [Последњи приступ Јун 2024].

[17] M. J. Price, C# 7.1 i .NET Core 2.0, Packt, 2018.

Кратка биографија:

Лука Бјелица рођен је у Зрењанину 1999. год. Мастер рад на Факултету техничких наука из области Електротехнике и рачунарства – Софтверско инжењерство и информационе технологије одбранио је 2024. год.

контакт: bjelicaluka99@gmail.com