LATVIJAS UNIVERSITĀTE
EKSAKTO ZINĀTŅU UN TEHNOLOĢIJU FAKULTĀTE

JAVA VALODĀ BALSTĪTU TĪMEKĻA IETVARU SALĪDZINĀJUMS

BAKALAURA DARBS

Autors: **Daniels Janovskis**
Studenta apliecības Nr.: dj21030
Darba vadītājs: pasniedzējs Dr. Sergejs Rikačovs

RĪGA 2025

ANOTĀCIJA

Darbā tiek salīdzināti populārie Java tīmekļa ietvari. Salīdzinājums fokusējas uz to arhitektūras risinājumiem, veiktspēju un pielietošanas ērtumu tīmekļa vietņu izstrādē. Tas iekļauj gan teorētisko pētījumu, gan ietvaru praktisko pielietojumu, izstrādājot līdzīgas funkcionalitātes REST API vietnes ar dažādiem ietvariem. Salīdzinājums tiks veikts, izvērtējot katra ietvara priekšrocības un trūkumus, prasības pret resursu izmantošanu, piemērotību specifiskiem uzdevumiem un veiktspējas rādītājiem.

Rezultātā, balstoties uz darba izstrādes laikā veikto pētījumu un vietņu testu rezultātiem, tiks secināts, kādās situācijās un kādiem nolūkiem ir labāk piemērots katrs no apskatītiem ietvariem.

Atslēgvārdi: ietvars, REST API, salīdzinājums, priekšrocības un trūkumi

ABSTRACT

COMPARISON OF JAVA WEB FRAMEWORKS

This work compares popular Java web frameworks. The comparison focuses on their architectural solutions, performance, and ease of use in web development. It includes both theoretical research and practical application of frameworks by developing REST API sites with similar functionality using different frameworks. The comparison will be conducted by evaluating each framework's advantages and disadvantages, resource requirements, suitability for specific tasks, and performance indicators.

As a result, based on the research conducted during the work and the test results of the sites, conclusions will be drawn about which situations and purposes each of the examined frameworks is better suited for.

Keywords: framework, REST API, comparison, advantages and disadvantages

SATURA RĀDĪTĀJS

APZĪMĒJUMU SARAKSTS 6
IEVADS 7
1. IESKATS JAVA TĪMEKĻA IETVAROS 8
1.1. Java tīmekļa ietvaru evolūcija 8
2. APSKATĀMIE IETVARI 9
2.1 Kopsavilkums par ietvariem 9
2.1.1 Spring Boot 9
2.1.2. Play Framework 9
2.1.3. Micronaut 10
2.1.4. Quarkus 10
2.1.5. Vert.x 10
2.1.6. Dropwizard 11
3. JAVA TĪMEKĻA IETVARU PADZIĻINĀTA IZPĒTE 12
3.1 Pētnieciskās metodikas pamatojums 12
3.2 Ietvaru atlases kritēriji un izvēlētie kandidāti 13
3.2.1 Spring Boot izvēles pamatojums 13
3.2.2 Micronaut tehnoloģiskās inovācijas 13
3.2.3 Quarkus natīvā kompilācija 14
3.4. Spring Boot ietvara analīze 14
3.4.1 Arhitektūras koncepcija un galvenie principi 14
3.4.2 Dependency Injection mehānisms 15
3.4.3 REST API realizācijas principi 16
3.4.4 Datu piekļuves slānis 17
3.4.5 Cloud-native iespējas 17
3.4.6 Drošības risinājumi 18
3.4.7 Testēšanas infrastruktūra 19
3.4.8 Monitoringa un aktuatora iespējas 19
3.5 Micronaut ietvara detalizēta analīze 19
3.5.1 Arhitektūras koncepcija un inovācijas 19
3.5.2 Compile-time Dependency Injection 20
3.5.3 REST API realizācijas principi 21
3.5.4 Reaktīvā programmēšana 22
3.5.5 Datu piekļuves slānis 23
3.5.6 Cloud-native iespējas 23

3.5.7 Testēšanas infrastruktūra 24
3.5.8 Monitoringa un aktuatora iespējas 24
3.6 Quarkus ietvara detalizēta analīze 25
3.6.1 Arhitektūras koncepcija un unikālās īpašības 25
3.6.2 GraalVM natīvā kompilācija 26
3.6.3 Resursu-orientēta REST API pieeja 26
3.6.4 Panache datu piekļuves modelis 27
3.6.5 Izstrādātāju produktivitāte 29
3.6.6 Cloud-native iespējas 29
3.6.7 Testēšanas infrastruktūra 30
3.6.8 Monitoringa un diagnostikas iespējas 31
4. REST API VIETNE IETVARU PRAKTISKAI SALĪDZINĀŠANAI. 32
4.1. Kāpēc REST API? 33
4.2. Vietnes datubāzes diagramma 34
4.3. Vietnes struktūra un vaicājumi. 35
4.4. Lietotāju autentifikācija un drošības risinājumi 40
4.5. Ietvaru izstrādes risinājumi 40
4.5.1. Projekta struktūra 40
4.5.2. Konfigurācija 41
4.5.3. Izmantotas tehnoloģijas 41
4.6. Mērījumu metrikas 42
5. PRAKTISKA SALĪDZINĀJUMĀ REZULTĀTI 43
5.1. Spring Boot ietvara rezultāti 43
5.1.1. Uzstadīšana un konfigurēšana 43
5.1.2. Darbietilpība 44
5.1.3. Veiktspēja 44
5.1.4. GraalVM natīvā kompilācija 45
5.1.5. Dokumentācija 45
5.1.6. Drošība 45
5.1.7. Priekšrocības un trūkumi 46
5.2. Micronaut ietvara rezultāti 46
5.2.1. Uzstādīšana un konfigurēšana 46
5.2.2. Darbietilpība 46
5.2.3. Veiktspēja 47
5.2.4. GraalVM natīvā kompilācija 47
5.2.5. Dokumentācija 48
5.2.6. Drošība 48

5.2.7. Priekšrocības un trūkumi ... 48
5.3. Quarkus ietvara rezultāti ... 49
5.3.1. Uzstādīšana un konfigurēšana ... 49
5.3.2. Darbietilpība ... 50
5.3.3. Veiktspēja ... 50
5.3.4. GraalVM natīvā kompilācija ... 51
5.3.5. Dokumentācija ... 51
5.3.6. Drošība ... 51
5.3.7. Priekšrocības un trūkumi ... 51
5.4. Koda struktūras salīdzinājums ... 52
6. REZULTĀTU ANALĪZE ... 56
6.1. Izstrāde ... 56
6.2. Veiktspēja ... 57
6.3. Galējie rezultāti ... 58
6.3.1. Izstrādes ērtums un darbietilpība ... 59
6.3.2. Veiktspēja un resursu patēriņš ... 59
6.3.3. Drošība un autentifikācija ... 60
6.3.4. Dokumentācija un kopienas atbalsts ... 60
SECINĀJUMI ... 61
IZMANTOTĀ LITERATŪRA UN AVOTI ... 63
PIELIKUMI ... 65
1. pielikums. Spring Boot ietvara produktu kontrolieris ... 65
2. pielikums. Micronaut ietvara produktu kontrolieris ... 66
3. pielikums. Quarkus ietvara produktu kontrolieris ... 69

APZĪMĒJUMU SARAKSTS

DI jeb atkarību injekcija (Dependency Injection) - Objektorientētas programmēšanas tehnika, kas samazina iekodētas (hardcoded) atkarības starp objektiem, uzlabojot koda kvalitāti un testējamību

JVM - Java virtuālā mašīna

MVC - Model, view, controller

Docker - konteinerizācijas platforma, kas ļauj programmatūru un tās atkarības iepakot standartizētos konteineros, kuri satur visu nepieciešamo koda izpildei

Kontroles inversija (Inversion of Control) - programmatūras arhitektūras princips, kurā programmas komponenti saņem savu darbības plūsmu un atkarības no ārēja ietvara vai konteinera (piemēram Spring container), nevis paši tieši kontrolē to izveidi un vadību.

Bean (pupa) - Java objekts, ko pārvalda Spring IoC (Inversion of Control) konteiners. Tas ir standarta Java klases instance ar īpašām konfigurācijas īpašībām, kas ļauj konteineriem veikt automātisku objektu inicializāciju, konfigurāciju un atkarību injekciju.

Reaktīva programmēšana - programmēšanas paradigma, kas balstās uz datu plūsmas un izmaiņu automātisku izplatīšanu sistēmā, kur komponenti automātiski reaģē uz datu izmaiņām un notikumiem reāllaikā.

Bezservera (Serverless) arhitektūra – mākoņskaitļošanas modelis, kurš ļauj palaist lietotnes kodu bez nepieciešamības pārvaldīt serverus. Serveru uzturēšanu veic mākoņpakalpojumu sniedzējs.

Kubernetes - atvērtā pirmkoda konteineru orķestrācijas platforma, kas automatizē programmatūras konteineru izvietošanu, mērogošanu un pārvaldību datoru klasteros.

LTS (Long Time Support) - ir programmatūras izlaides veids, kas nodrošina ilgāku uzturēšanas un atbalsta periodu nekā standarta vai regulārie izlaidumi. Šīs versijas ir paredzētas stabilitātei un drošībai, nevis jaunākajām funkcijām.

ORM (Object-Relational Mapping) – programmēšanas tehnika, kas savieno datubāzes ar objektorientētas programmēšanas koncepcijām, automātiski pārveidojot datus starp objektorientētajām programmēšanas valodām un relāciju datubāzēm.

JWT (JSON Web Token) – atklātais standarts piekļuves tokenu izveidei klients-serveris lietotnēs.

Boilerplate kods – kods, kas atkārtojas vairākās projekta vietās bez ievērojamām izmaiņām.

Endpoint – URL adrese, kas pārstāv kādu programmas funkcionalitāti.

Separation of Concerns - datorprogrammas dizaina princips, kas sadala programmas klases un funkcijas dažādās sekcijās.

IEVADS

Mūsdienās Java programmēšanas valoda tiek plaši pielietota dažādas sarežģītības tīmekļa vietņu izstrādē, pateicoties tās efektīviem pārnesamības, mērogojamības, drošības un integrācijas risinājumiem. Objektorientēta pieeja ļauj organizēt kodu vairākos objektos un moduļos, kuru atkārtota lietošana atvieglo kopējo izstrādes procesu. Java valodai pastāv vairāki ietvari ar plašu infrastruktūru, kā arī vairākiem moduļiem un bibliotēkām. Spring Boot jau vairākus gadus ir vispopulārākais no eksistējošiem Java tīmekļa ietvariem un ir plaši lietots vairāku dažāda mēroga vietņu izstrādē. Tomēr pēdējā laikā tika izstrādāti daudzi ietvari, kas ir domāti līdzīgiem nolūkiem un darbojas pēc līdzīga principa, taču izmanto citādas, dažreiz vairāk progresīvas pieejas. Daudzos gadījumos tie var būt vairāk piemēroti un efektīvāk risina apskatāmas problēmas un izaicinājumus tīmekļa vietņu izstrādē.

Šī darba teorētiskas daļas mērķis ir izpētīt vairākus populārus Java ietvarus, veidojot priekšstatu par katra ietvara galvenajām īpašībām, kā arī priekšrocībām un trūkumiem, salīdzinot ar citiem; pēc ieskata izvēlēties 3 ietvarus – vispopulārāko no tiem un divus citus, kurus varētu uzskatīt par tā vislabāko alternatīvu, lai plašāk aprakstītu to arhitektūru, mikroservisu risinājumus, drošību un citas funkcionalitātes realizāciju. Savukārt praktiskas daļas mērķis ir izveidot REST API vietni ar 3 izvēlētu ietvaru palīdzību, lai uz praktiska piemēra salīdzinātu to funkcionalitāti ar vairākām metrikām. Balstoties uz teorētiska un praktiska salīdzinājuma tiks izdarīti secinājumi par ietvaru kopējo funkcionalitāti, veiktspēju, drošību un pārējām īpašībām, un tiks atzīmētas katra ietvara stipras un vājās puses.

Darba autors jau bija sācis šo pētījumu kursa darba ietvaros, tāpēc šajā darbā tas tiks paplašināts ar vairāku ietvaru apskatu, plašāku iedziļināšanos to arhitektūrā un praktisko realizāciju.

1. IESKATS JAVA TĪMEKĻA IETVAROS

1.1. Java tīmekļa ietvaru evolūcija

Java tīmekļa ietvari ir būtiski mainījušies līdz ar tīmekļa tehnoloģiju attīstību. Sākotnēji Java tīmekļa izstrāde balstījās uz Java Servlet API un JSP (JavaServer Pages). Vēlāk parādījās Enterprise Java Beans (EJB) un Java Enterprise Edition (Java EE) standarts, kas piedāvāja visaptverošu risinājumu uzņēmumu lietojumprogrammu izstrādei.

2002. gadā Rod Johnson izlaida Spring Framework, kas piedāvāja vieglāku alternatīvu Java EE, ieviešot kontroles inversiju (IoC) un atkarību ievietošanas (DI) principus. Spring Framework popularitāte strauji auga, jo tas piedāvāja:

- Vienkāršāku izstrādes procesu
- Labāku testējamību
- Mazāku atkarību no specifiska aplikāciju servera [1]

Kopš tā laika tika izstrādāti vairāki Java tīmekļa ietvari, un mūsdienās to izvēle ir plaša, un katrs ietvars cenšas risināt specifiskas problēmas: izstrādes un izvietošanas paātrināšana, resursu patēriņa samazināšana un labāka integrācija ar modernām tehnoloģijām.

2. APSKATĀMIE IETVARI

Java valoda ir ieguvusi popularitāti tīmekļa vietņu izstrādē, tāpēc šodien pastāv ļoti daudz Java valodā balstītu tīmekļa ietvaru, kas piedāvā atšķirīgas pieejas un paradigmas tīmekļa vietņu izstrādē. Tās atvieglo dažādus izstrādes aspektus un piedāvā izstrādātājam labāku pieredzi vietnes izveides laikā. Katrs no ietvariem izšķiras ar saviem risinājumiem un unikālām priekšrocībām. Tomēr vispopulārākais joprojām paliek Spring Boot ietvars, kas pēdējos 20 gados ir kļuvis par faktisko standartu Java tīmekļa vietņu izstrādei. [2] Vairākos ietvaros, kas tika izstrādāti vēlāk tika implementēta Spring Boot ietvara funkcionalitāte, un to arhitektūra dažreiz ir ļoti līdzīga. Bet daudzi modernāki ietvari tika izstrādāti, novērtējot tā trūkumus un nepilnības, un mēģina minimizēt tos, implementējot jaunus, dažreiz progresīvākus risinājumus.

Šajā nodaļā tiks apskatīti vairāki Java balstīti tīmekļa ietvari, izpētot to popularitāti, pieejamību un pielietošanu.

2.1 Kopsavilkums par ietvariem

2.1.1 Spring Boot

Spring Boot ir vispopulārākais Java tīmekļa ietvars, kas tika izlaists 2014. gadā kā Spring Framework paplašinājums. Tā galvenais mērķis bija vienkāršot Spring Framework lietošanu, samazinot nepieciešamo konfigurāciju un paātrinot izstrādes procesu. Spring Boot ir ieguvis plašu popularitāti, pateicoties tā plašajai ekosistēmai un nobriedušajai kopienai.

Tā jaunākā versija ir 3.5, kura atbalsta Java valodas pēdējo 24. versiju, un pēdējo LTS versiju (Java 21). [14]

Spring Boot arhitektūra vienkāršo vietņu izstrādi, vienlaikus saglabājot Spring Framework ietvara elastību un paplašināmību. Tā balstās uz vairākām galvenajām komponentēm un izmanto MVC pieeju.

2.1.2. Play Framework

Play Framework ir moderns, viegls un elastīgs tīmekļa ietvars, kas tika izveidots 2007. gadā un paredzēts reaktīvu, augstas veiktspējas tīmekļa lietojumprogrammu izstrādei. Tas ir balstīts uz Scala valodu, bet piedāvā pilnvērtīgu atbalstu arī Java izstrādei. Play ietvars atšķiras no tradicionālajiem Java ietvariem ar savu bezstāvokļa (stateless) un resursu efektīvu arhitektūru, kas to padara īpaši piemērotu mūsdienu tīmekļa lietojumprogrammām. Tam arī

piemīt "Just Hit Refresh" izstrādes pieeja, kas ļauj redzēt koda izmaiņas tieši pārlūkprogrammā bez manuālas rekompilēšanas vai servera restartēšanas.

Tā jaunākā versija ir 3.0.x, kas pilnībā atbalsta Java 21 un Scala 3

Play Framework arhitektūra ir veidota ap MVC modeli, izmantojot neatkarīgu pieprasījumu apstrādi un nebloķējošas I/O operācijas, kas padara to īpaši piemērotu lietojumprogrammām ar lielu vienlaicīgu lietotāju skaitu un reaktīvām sistēmām. [15]

2.1.3. Micronaut

Micronaut ir salīdzinoši jauns ietvars, kas tika izstrādāts 2018. gadā ar mērķi vienkāršot un paātrināt mikroservisu izveidi. Tas ir veidots kā viegls un modulārs JVM ietvars, kas piedāvā alternatīvu tradicionālajiem risinājumiem. OCI kompānija, kas iepriekš radīja populāro Grails ietvaru, ir ieviesusi Micronaut kā modernu platformu, kas apvieno labākās prakses no esošajiem ietvariem ar inovatīvām pieejām mikroservisu arhitektūras realizēšanai. [16]

Micronaut jaunākā versija ir 4.8.2., atbalsta Java valodas jaunākas versijas.

2.1.4. Quarkus

Quarkus ir modernais Java ietvars, kas izstrādāts 2019. gadā ar mērķi padarīt Java valodu par vadošo platformu Kubernetes un serverless vidē. Tā galvenie mērķi ir nodrošināt zemu resursu patēriņu, ātru startēšanas laiku un optimālu veiktspēju konteineru vidē, vienlaikus saglabājot izstrādātāju produktivitāti.

Jaunākā versija Quarkus ietvaram ir 3.22.x

Qurakus ietvars pašlaik ir viena no labākajām izvēlēm mākoņlietotņu izstrādei. Tas arī piedāvā ērtu izstrādi, pateicoties tādām īpašībām, kā Live Coding un Dev Services, kas atvieglo izstrādes procesu. [17]

2.1.5. Vert.x

Vert.x ir augstas veiktspējas, poliglots un reaktīvs ietvars, kas tika izlaists 2011. gadā ar mērķi nodrošināt vienkāršu un efektīvu veidu, kā izstrādāt modernas, reaktīvas JVM lietojumprogrammas. Atšķirībā no daudziem ietvariem, Vert.x nav stingri saistīts ar Java, bet ļauj izstrādātājiem rakstīt kodu vairākās valodās, tostarp Java, Kotlin, JavaScript, Ruby, Groovy un Ceylon.

Tā jaunākā versija ir 5.0., kas atbalsta Java 21. Vert.x realizē reaktīvo programmēšanas modeli un ir orientēts uz asinhrono notikumu apstrādi, izmantojot notikumu cilpu modeli, kas līdzīgs Node.js, bet ar vairāku pavedienu priekšrocībām. [18]

2.1.6. Dropwizard

Dropwizard ir kompakts Java ietvars, kas tika izveidots 2011. gadā ar mērķi ātri izveidot RESTful tīmekļa pakalpojumus ar minimālu konfigurāciju. Tas apvieno vairākas nobriedušas un pārbaudītas Java bibliotēkas vienotā, saskaņotā platformā, kas ļauj izstrādātājiem ātri uzsākt darbu, neuztraucoties par bibliotēku savietojamību vai konfigurāciju.

Tā jaunākā versija ir 5.0.0, kas tika modernizēta, lai pilnībā izmantotu Java modulāro sistēmu un atbalstītu jaunākās Java versijas, tostarp Java 17 un 21.

Dropwizard arhitektūra ir vērsta uz vienkāršību un veiktspēju, piedāvājot izolētu, autonomu izpildes modeli, kas padara to ideāli piemērotu mikroservisu izstrādei. Tas ietver iebūvētu atbalstu veselības pārbaudēm, metrikām un konfigurācijas pārvaldībai, kas ir būtiski elementi modernās mākoņrēķinu (Cloud Counting) lietojumprogrammās. [19]

3. JAVA TĪMEKĻA IETVARU PADZIĻINĀTA IZPĒTE

3.1 Pētnieciskās metodikas pamatojums

Pētījuma uzsākšanai tika izstrādāta metodoloģija, kas balstīta gan uz teorētisku analīzi, gan uz praktisku pieeju ietvaru funkcionalitātes novērtēšanai. Mūsdienu Java ekosistēmā tīmekļa aplikāciju izstrādes risinājumi piedāvā dažādus arhitektūras konceptus, produktivitātes rīkus un tehniskos risinājumus, kas ir pielāgoti specifiskajām vajadzībām.

Java tīmekļa izstrādes kontekstā "ietvars" (angliski “framework”) ir komplekss tehniskais risinājums, kas nodrošina programmatūras izstrādātājiem strukturētu un standartizētu pieeju aplikāciju izveidei. Tas ietver gan koda organizācijas konvencijas, gan gatavus komponentus biežāk sastopamo problēmu risināšanai, tādā veidā ļaujot izstrādātājiem koncentrēties uz biznesa loģikas implementāciju. [1]

Sākotnēji tika apsvērta teorētiska ietvaru salīdzināšana, analizējot to tehniskās specifikācijas, arhitektūras risinājumus, pieejamo funkcionalitāti un dokumentāciju. Tomēr šāda pieeja nesniegtu pietiekami dziļu izpratni par ietvaru praktisko pielietojumu un lietošanas pieredzi. Tāpēc šim pētījumam tika izstrādāta arī praktiska testēšanas metodika.

Šī metodika paredz vienota parauga lietojumprogrammas izstrādi visos analizējamos ietvaros, izmantojot identiskas prasības un datu struktūras. Šāda pieeja nodrošina objektīvu salīdzinājumu, jo:

- Visi ietvari tiek testēti vienādos apstākļos
- Tiek izmērīti konkrēti, identiski parametri
- Praktiskā implementācija atklāj nianses, kas teorētiskā analīzē var palikt nepamanītas

Parauga lietojumprogramma tika definēta kā REST API serviss ar datu saglabāšanas, nolasīšanas, atjaunināšanas un dzēšanas funkcionalitāti, kas implementē tipiskākās biznesa vajadzības.

3.2 Ietvaru atlases kritēriji un izvēlētie kandidāti

No plašā Java tīmekļa izstrādes ietvaru klāsta sākotnējai analīzei tika izvēlēti septiņi ietvari: Spring Boot, Quarkus, Micronaut, Play Framework, Vert.x Dropwizard. Plašākai izpētei un praktiskai realizācijai tika atlasīti trīs ietvari, kas pārstāv dažādas pieejas modernai tīmekļa risinājumu izstrādei.

3.2.1 Spring Boot izvēles pamatojums

Spring Boot tika iekļauts detalizētajā analīzē, jo tas ir de facto standarts Java tīmekļa izstrādē. [2] Šis ietvars piedāvā principu "viedoklis pēc noklusējuma" (opinionated defaults), kas ievērojami samazina konfigurācijas apjomu. To raksturo:

- Plaša ekosistēma ar vairāk nekā 50 starta pakotņu (starters) atbalstu
- Ievērojams kopienas atbalsts ar tūkstošiem gatavu risinājumu drošībai, datu piekļuvei un lietojumprogrammu integrācijai
- Ilggadēja stabilitāte un uzticība [20]

Spring Boot izvēle dod iespēju analizēt tradicionālu pieeju, ar kuru varētu salīdzināt jaunākas ietekmes.

3.2.2 Micronaut tehnoloģiskās inovācijas

Micronaut tika izvēlēts kā otrais analizējamais ietvars, jo tas pārstāv jaunu paaudzi Java tīmekļa izstrādē. Tā galvenā atšķirība ir kompilācijas laika metadatu apstrāde, atsakoties no tradicionālās Java refleksijas, kas nodrošina:

- Ievērojami mazāku atmiņas patēriņu
- Ātrāku palaišanas laiku (sekunžu vietā – milisekundes)
- Optimizāciju mikroservisu arhitektūrai un bezservera (serverless) videi [4]

Šis ietvars piedāvā alternatīvu tradicionālajai pieejai, saglabājot līdzīgu programmēšanas modeli kā Spring Boot (izmantojot anotācijas), bet fundamentāli mainot to apstrādes mehānismu.

3.2.3 Quarkus natīvā kompilācija

Quarkus tika izvēlēts kā trešais analizējamais ietvars tā revolucionārās pieejas dēļ Java aplikāciju izstrādē. Red Hat atbalstītais ietvars ir radīts ar mērķi optimizēt Java lietojumprogrammas Kubernetes videi, piedāvājot:

- GraalVM natīvās kompilācijas atbalstu, kas pārveido Java kodu mašīnkodā
- "Supersonic Subatomic Java" pieeju ar izteikti zemu resursu patēriņu
- Reaktīvo programmēšanas paradigmu integrāciju
- Imperatīvā un reaktīvā programmēšanas modeļa apvienošanu vienā risinājumā [6]

Quarkus ir interesants pētījuma objekts tieši tā specializācijas dēļ uz modernām lietojumprogrammu izvietošanas (deployment) stratēģijām konteineru vidē.

3.4. Spring Boot ietvara analīze

3.4.1 Arhitektūras koncepcija un galvenie principi

Spring Boot arhitektūra balstās uz Spring Framework pamatprincipiem, vienlaikus ievērojami vienkāršojot tās lietošanu un konfigurāciju. Tā arhitektūras koncepcija "opinionated defaults" (viedoklis pēc noklusējuma) paredz automātisku konfigurāciju un prātīgus noklusējuma iestatījumus, palīdzot izstrādātājiem saņemt vēlamo konfigurāciju un koncentrēties uz biznesa loģiku, nevis tērēt daudz laika, konfigurējot izstrādes vidi.

```
@SpringBootApplication public class
SpringBootTestApplication {


   public static void main(String[] args) {
       SpringApplication.run(SpringBootTestApplication.class, args);
   }
}
```

Attēls 3.1. "Spring lietotnes startēšanas funkcija"

Šis vienkāršais koda fragments demonstrē Spring Boot minimālistisko pieeju - ar vienu anotāciju un vienu metodi tiek palaista pilnvērtīga tīmekļa lietojumprogramma ar iekļautu serveri un visu nepieciešamo funkcionalitāti. Tas ir novērojami atšķirīgi no tradicionālās Java EE arhitektūras, kur būtu nepieciešama ievērojama apjoma konfigurācija.

Spring Boot, savukārt, konfigurē aplikāciju, balstoties uz:

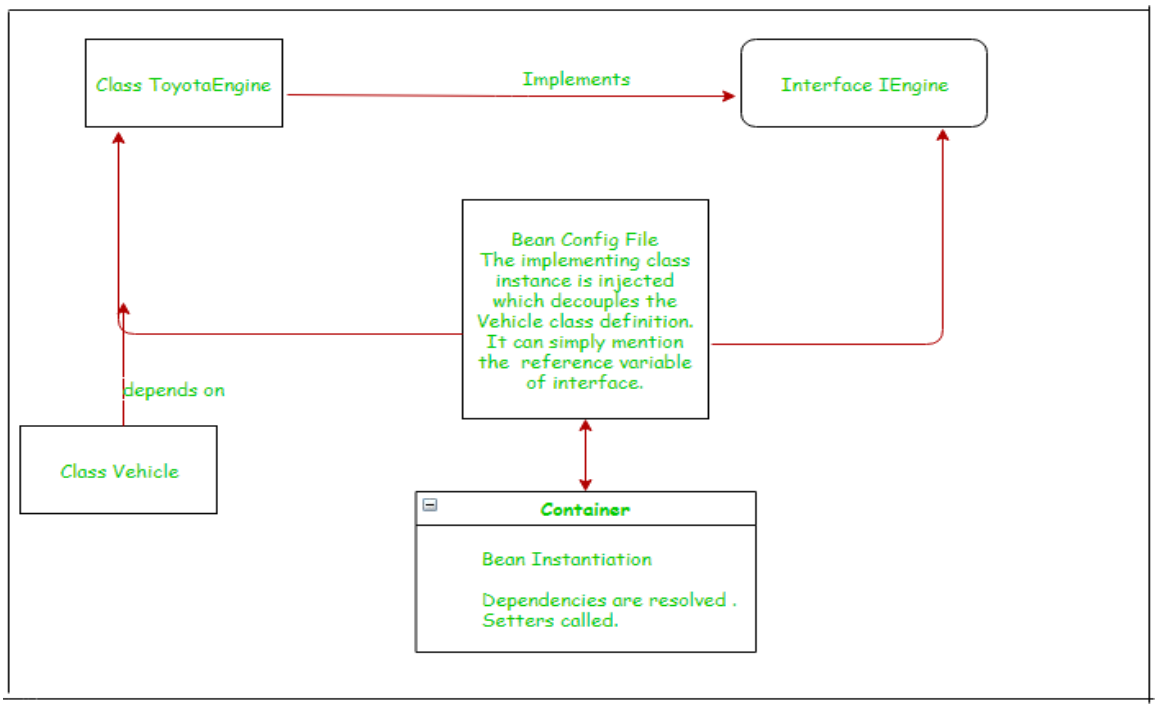
- Pievienotajām atkarībām (dependencies) un to pakotnēm - iepriekš definētiem kopumiem, kas vienkāršo izstrādi, piemēram:
    - spring-boot-starter-web (tīmekļa lietojumprogrammu izstrādei)
    - spring-boot-starter-data-jpa (JPA datubāzu integrācijai)
    - spring-boot-starter-security (drošības funkcionalitātei)
- Definētajām īpašībām (properties)
- Classpath esošajām klasēm

Iebūvēts servera konteiners ļauj veidot pašpietiekamas lietojumprogrammas bez ārēja aplikāciju servera. Spring Boot pēc noklusējuma izmanto Apache Tomcat serveru, bet atbalsta arī citus serverus, tādus kā Jetty jeb Netty. [21]
Spring Boot nodrošina pilnvērtīgu Model-View-Controller arhitektūras atbalstu.

3.4.2 Dependency Injection mehānisms

Atkarību injekcija (Dependency Injection, DI) ir centrālais Spring Boot konceptuālais elements, kas veido ietvara kodolu. Spring Boot izmanto Spring Framework DI konteineri, kas automātiski pārvalda komponenšu atkarības un to dzīves ciklu.

Spring Boot DI realizācija ir balstīta uz refleksijas mehānismu, kas izpildes laikā (runtime) skenē un analizē klases, lai identificētu un ievietotu nepieciešamās atkarības. Šī pieeja nodrošina augstu elastību, bet dažreiz ietekmē veiktspēju un atmiņas patēriņu. [1]



Attēls 3.2. "Spring DI procesu plūsma" [36]

Spring Boot DI mehānisma priekšrocība ir tā elastība un viegla konfigurējamība, bet salīdzinājumā ar modernākiem ietvariem tā izmanto vairāk resursu, jo balstās uz izpildes laika refleksiju.

3.4.3 REST API realizācijas principi

Spring Boot piedāvā visaptverošu REST API izstrādes atbalstu, izmantojot tā MVC arhitektūru. Tas nodrošina deklaratīvu pieeju ar anotācijām, kas veicina tīru un saprotamu kodu.

```
@RestController
@RequestMapping("/api/products")
public class ProductController
{
  private final ProductService productService;

  public ProductController(ProductService productService)
  {
    this.productService = productService;
  }
  @GetMapping("/{id}")
  public ResponseEntity<Product> getProduct(@PathVariable Long id)
  {
    return productService.findById(id)
      .map(ResponseEntity::ok)
      .orElse(ResponseEntity.notFound().build());
  }

  @PostMapping
  @ResponseStatus(HttpStatus.CREATED)
  public Product createProduct(@Valid @RequestBody Product product)
  {
    return productService.save(product);
  }
}
```

Attēls 3.3. "Spring kontroliera piemērs"

Spring Boot atbalsta arī uzlabotus REST API izstrādes risinājumus:

- **HATEOAS** (Hypermedia as the Engine of Application State) - hipersaišu pievienošana API atbildēm
- **OpenAPI/Swagger dokumentācija** - automātiska API dokumentācijas ģenerēšana

- **Paging un Sorting** - integrēta atbalsta funkcionalitāte lielu datu kopu apstrādei

3.4.4 Datu piekļuves slānis

Spring Boot datu piekļuves risinājumi balstās uz Spring Data moduli, kas nodrošina vienotu un konsekventu pieeju dažādiem datu avotiem. Spring Data JPA ir viens no populārākiem ORM risinājums, kas būtiski vienkāršo datubāzes operācijas.

```
public interface ProductRepository extends JpaRepository<Product, Long> {
    List<Product> findByCategory(String category);

    @Query("SELECT p FROM Product p WHERE p.price > :minPrice")
    List<Product> findExpensiveProducts(@Param("minPrice") BigDecimal minPrice);

    Optional<Product> findBySku(String sku);
}
```

Attēls 3. 4. "JPA repozitorija piemērs"

Kā ir redzams attēlā, @Query anotācija ļauj ievietot SQL vaicājumus Java kodā.

Spring Data nodrošina deklaratīvu pieeju datubāzes operācijām caur repozitorijiem. Tas atbalsta vairākus repozitoriju tipus (CrudRepository, PagingAndSortingRepository, JpaRepository, ReactiveCrudRepository utt.). Katrs repozitorija tips ir piemērots specifiskām gadījumam, piemēram, CrudRepository ir pamata tips, kas nodrošina bāzes CRUD operācijas un ir labi pielāgots neliela mēroga vietnēm, JpaRepository piedāvā JPA-specifisko iespēju, kas ir labāk integrējama ar Hibernate datubāzu ietvaru un, savukārt, ReactiveCrudRepository ir piemērots reaktīviem projektiem. [7]

Spring Data atbalsta gan SQL, gan NoSQL datubāzes, kā arī nodrošina arī automātisku datubāzes migrāciju ar Flyway vai Liquibase integrāciju un atbalsta reaktīvo datu piekļuvi (Reactive Repositories) ar Spring Data R2DBC. [22]

3.4.5 Cloud-native iespējas

Spring Boot piedāvā plašu cloud-native risinājumu komplektu caur Spring Cloud projektu. Šī ekosistēma nodrošina visus nepieciešamos komponentus mikroservisu arhitektūras izveidei:

1. **Service Discovery** - mikroservisu reģistrācija un atklāšana:
    - Spring Cloud Netflix Eureka vai
    - Spring Cloud Consul Discovery

2. **Circuit Breaker ar Spring Cloud Resilience4j** kā kļūmjnoturības risinājums
3. **Spring Cloud Gateway** - centralizēta mikroservisu maršrutēšana:
4. **Spring Cloud Config** - centralizēta konfigurācijas pārvaldība:
5. **Distributed Tracing** - distribuēta trasēšana:
    - Spring Cloud Sleuth
    - Integrācija ar Zipkin

Spring Boot arī sniedz natīvu Kubernetes atbalstu caur Spring Cloud Kubernetes, nodrošinot integrāciju ar Kubernetes Service Discovery, ConfigMaps un Secrets. [23]

3.4.6 Drošības risinājumi

Spring Boot drošības risinājumi balstās uz Spring Security ietvaru, kas piedāvā visaptverošu drošības modeli.

```java
@Configuration
@EnableWebSecurity
public class SecurityConfig {
    @Bean
    public SecurityFilterChain securityFilterChain(HttpSecurity http) throws Exception {
        return http
            .authorizeHttpRequests(authorize -> authorize
                .requestMatchers("/api/public/**").permitAll()
                .requestMatchers("/api/admin/**").hasRole("ADMIN")
                .anyRequest().authenticated()
            )
            .formLogin(form -> form
                .loginPage("/login")
                .permitAll()
            )
            .oauth2Login(oauth2 -> oauth2
                .loginPage("/login")
            )
            .build();
    }
}
```

Attēls 3.5. "Drošības konfigurācijas piemērs"

Spring Security piedāvā mehānismus:

1. **Autentifikācijas līmenī**:
    - Formu autentifikācija
    - OAuth 2.0 / OpenID Connect
    - JWT (JSON Web Tokens)
    - LDAP

2. **Autorizācijas līmenī**:
    - URL balstīta piekļuve
    - Metožu līmeņa drošība ar @PreAuthorize un @PostAuthorize
    - RBAC (Role-Based Access Control)
3. **CSRF aizsardzība** un citi drošības pasākumi [38]

3.4.7 Testēšanas infrastruktūra

Spring Boot piedāvā bagātīgu testēšanas ekosistēmu, kas integrējas ar populāriem testēšanas ietvariem:

Spring Boot piedāvā vairākus testēšanas rīkus. JUnit un Mockito ļauj veikt un pārvaldīt Unit testus, anotācija @SpringBootTest - integrācijas testēšanai un anotācijas: @WebMvcTest, @DataJpaTest, @RestClientTest - slice testēšanu. [24]

3.4.8 Monitoringa un aktuatora iespējas

Spring Boot Actuator monitoringa rīks nodrošina production-ready monitoringa un pārvaldības iespējas:

1. **Health endpoints** - sistēmas veselības statusa pārbaude
2. **Metrikas** - sistēmas metriku eksportēšana Prometheus, Grafana un citiem rīkiem
3. **Logging** - dinamisks log līmeņu konfigurēšana
4. **Thread dump un Heap dump** - diagnostikas informācija
5. **Environment** - konfigurācijas informācijas piekļuve

3.5 Micronaut ietvara detalizēta analīze

3.5.1 Arhitektūras koncepcija un inovācijas

Micronaut arhitektūra ir būvēta ar mērķi pārvarēt tradicionālo Java ietvaru ierobežojumus, it īpaši attiecībā uz resursu patēriņu un palaišanas laiku. Ietvara galvenā inovācija ir refleksijas aizvietošana ar kompilācijas laika metadatu apstrādi. [3]

```java
@Singleton
public class Application {
    public static void main(String[] args) {
        Micronaut.run(Application.class, args);
    }
}
```

Attēls 3.6. "Micronaut vietnes startēšanas funkcija"

Šis vienkāršais piemērs demonstrē Micronaut minimālistisko pieeju, kas līdzinās Spring Boot, taču aiz šīs līdzības slēpjas fundamentāli atšķirīgs darbības princips. Micronaut lietojumprogrammas struktūra ir ātrāka un resursus taupoša, jo visas metadatu un anotāciju apstrādes, kā arī koda ģenerēšanas darbības, notiek kompilācijas laikā, nevis programmas izpildes laikā. Iebūvēts Netty serveris nodrošina asinhronu un efektīvu tīkla komunikāciju.

3.5.2 Compile-time Dependency Injection

Micronaut ietvara pamatā ir revolucionāra pieeja atkarību injekcijai (DI), kas pilnībā atšķiras no Spring Boot un citiem vairāk tradicionālajiem Java ietvariem. Minimāla refleksijas izmantošana izraisa zemāku atmiņas patēriņu un ātrāku palaišanas laiku. Palaižot Micronaut programmu, tiek izmantoti tikai tie komponenti, kas nepieciešami, pateicoties ietvara modulārai arhitektūrai.

```java
@Singleton
public class ProductService {

    private final ProductRepository productRepository;
    private final CategoryRepository categoryRepository;

    public ProductService(ProductRepository productRepository, CategoryRepository categoryRepository) {
        this.productRepository = productRepository;
        this.categoryRepository = categoryRepository;
    }

    public ProductDto create
```

Attēls 3.7. "Micronaut atkarību injekcija"

Šajā piemērā @Singleton anotācija liek Micronaut kompilatoram ģenerēt proxy klasi un nepieciešamās injekcijas klases kompilācijas laikā. Tas novērš vajadzību pēc izpildes laika refleksijas un dinamiskās klašu ielādēšanas.

Katrai pupai (bean) tiek ģenerētas specializētas BeanDefinition klases, atkarību grafam tiek veikta validācija kompilācijas laikā, nevis izpildes laikā un pupu injekcijas notiek

caur tiešiem metožu izsaukumiem, nevis caur refleksijas mehānismiem. Atkarību grafa validācija arī notiek kompilācijas laikā. [25]

Šī pieeja sniedz vairākas priekšrocības:

- Samazināts atmiņas patēriņš, jo nav jāsaglabā refleksijas metadati
- Ātrāks palaišanas laiks, jo nav jāveic klašu skenēšana izpildes laikā
- Ātrākas pupu injekcijas, jo tās notiek kā tiešie metožu izsaukumi
- Kompilācijas laika kļūdu pārbaudes ļauj agrāk atklāt problēmas

3.5.3 REST API realizācijas principi

Micronaut nodrošina deklaratīvu, anotācijām balstītu pieeju REST API izveidei, kas ārēji līdzinās Spring Boot, bet ar atšķirīgu iekšējo darbības mehānismu:

```
@Controller
public class ProductController {
    private final ProductService productService;

    public ProductController(ProductService productService) {
        this.productService = productService;
    }
   }

    @Get("/{id}")
    public HttpResponse<Product> getById(Long id) {
        return productService.findById(id)
            .map(HttpResponse::ok)
            .orElse(HttpResponse.notFound());
    }

    @Post
    @Status(HttpStatus.CREATED)
    public Product create(@Body @Valid Product product) {
        return productService.save(product);
    }
}
```

Attēls 3.8. "Micronaut kontroliera piemērs"

Micronaut REST API īpašības:

1. **Kontrolieri** definēti ar @Controller anotāciju
2. **Maršruti** definēti ar @Get, @Post, @Put, @Delete anotācijām
3. **Parametru bindings** ar @PathVariable, @QueryValue, @Body anotācijām
4. **Validācija** ar Jakarta Validation anotācijām
5. **HTTP atbildes** ar HttpResponse klasi vai @Status anotāciju

Micronaut papildus piedāvā URI šablonu variantus ar augstu veiktspēju, validāciju kompilācijas laikā, kas ļauj identificēt problēmas ātrāk un reaktīvās atbildes ar RxJava, Reactor vai Coroutines integrāciju.

3.5.4 Reaktīvā programmēšana

Micronaut piedāvā visaptverošu reaktīvās programmēšanas atbalstu, kas ļauj efektīvi apstrādāt asinhronus datus un notikumus.

```
@Controller
public class ReactiveProductController {
    private final ReactiveProductRepository repository;

    public ReactiveProductController(ReactiveProductRepository repository) {
        this.repository = repository;
    }

    @Get
    public Flux<Product> listAll() {
        return repository.findAll();
    }

    @Get("/{id}")
    public Mono<HttpResponse<Product>> getById(Long id) {
        return repository.findById(id)
            .map(HttpResponse::ok)
            .defaultIfEmpty(HttpResponse.notFound());
    }
}
```

Attēls 3.9. "Micronaut reaktīva kontroliera piemērs"

Reaktīvās programmēšanas īpašības Micronaut ietvarā:

- **Natīvs atbalsts** populārām reaktīvām bibliotēkām:
    - Project Reactor (Mono, Flux)
    - RxJava (Observable, Flowable)

    - Kotlin Coroutines

- **Reaktīvs HTTP klients efektīvai ārējo API patērēšanai:**
    - Deklaratīvi definēti ar @Client anotāciju
    - Nebloķējošas I/O darbības

- **Reaktīvs datu piekļuves slānis ar R2DBC integrāciju:**
    - Nebloķējošas datubāzes operācijas
    - Augsta caurlaidība ar mazāku pavedienu skaitu

- **Backpressure atbalsts -** plūsmas kontrole starp datu plūsmas komponentiem [26]

3.5.5 Datu piekļuves slānis

Micronaut Data piedāvā kompilācijas laika repozitoriju implementācijas ģenerēšanu, kas novērš runtime refleksijas nepieciešamību datu piekļuvei. SQL vaicājumi Micronautā arī tiek ģenerēti kompilācijas laikā.

Micronaut Data atbalsta vairākus datu avotus:

- JPA (Hibernate)
- JDBC
- R2DBC (reaktīvās datubāzes)
- MongoDB

3.5.6 Cloud-native iespējas

Micronaut ir projektēts ar iespēju ērti realizēt mikroservisu arhitektūru un cloud-native lietojumprogrammas:

1. **Service Discovery** integrācija:
    - Consul
    - Eureka
    - Kubernetes
2. **Konfigurācijas pārvaldība**:
    - Distributed Configuration
    - Kubernetes ConfigMaps un Secrets
3. **Circuit Breaker** ar @CircuitBreaker anotāciju:

- Integrācija ar Resilience4j
- Deklaratīvu kļūmjnoturības definēšanu

4. **Vieglsvara un ātra HTTP klienta funkcionalitāte**:
   - Deklaratīvi HTTP klienti ar @Client anotāciju
   - Automātiska un efektīva slodzes balansēšana (load balancing)
5. **Message-driven mikroservisi**:
   - Kafka, RabbitMQ integrācija
   - Bezservera funkciju atbalsts [27]

3.5.7 Testēšanas infrastruktūra

```
@Client("product-service")
public interface ProductClient {
    @Get("/products")
    List<Product> getProducts();

    @Get("/products/{id}")
    Optional<Product> getProduct(Long id);
}
```

Attēls 3.10. "Micronaut HTTP klienta deklarācija"

Micronaut piedāvā efektīvu testēšanas infrastruktūru, kas fokusējas uz ātrumu un resursu efektivitāti:

1. **Ātra testu palaišana** - minimāls konteksta ielādes laiks
2. **Tīra DI ar katru testu**, vienlaikus saglabājot ātrumu
3. **Skaidrs HTTP testa klients** ar deklaratīvām operācijām
4. **Mockito integrācija** vienību testēšanai
5. **Testcontainers atbalsts** integrācijas testiem

3.5.8 Monitoringa un aktuatora iespējas

Micronaut Management modulis piedāvā monitoringa un pārvaldības funkcionalitāti, kas ir līdzīga Spring Boot Actuator, bet ar zemāku resursu patēriņu:

1. **Health endpoints** sistēmas veselības stāvokļa pārbaudēm
2. **Metrikas** ar Micrometer integrāciju:

    - Prometheus
    - StatsD
    - Atlas
3. **Tracing** ar OpenTracing vai OpenTelemetry:
    - Zipkin
    - Jaeger
4. **Logs** pārvaldība un konfigurācija [28]

3.6 Quarkus ietvara detalizēta analīze

3.6.1 Arhitektūras koncepcija un unikālās īpašības

Quarkus arhitektūra tiek pozicionēta kā "Supersonic Subatomic Java", kas nozīmē ātru startēšanu un aplikācijas mazumu. Šī ietvara mērķis ir radikāli samazināt Java aplikāciju palaišanas laiku un atmiņas patēriņu. Tā galvenā inovācija ir aplikāciju optimizācija natīvai kompilācijai ar GraalVM.

```java
@Path("/hello")
public class GreetingResource {

    @GET
    @Produces(MediaType.TEXT_PLAIN)
    public String hello() {
        return "Hello from Quarkus";
    }
}
```

Attēls 3.11. "Quarkus vietnes resursa piemērs"

Šis vienkāršais piemērs demonstrē Quarkus deklaratīvo, JAX-RS balstīto pieeju. Aiz šīs vienkāršības slēpjas kompleksa arhitektūra, kas nodrošina ārkārtīgi efektīvu koda izpildi gan JVM, gan natīvās kompilācijas režīmā.

Galvenās Quarkus arhitektūras īpašības:

- **Container-First pieeja** - optimizēta arhitektūra konteineru vidēm
- **Build-time optimizācija** - maksimāli daudz darba tiek veikts kompilācijas laikā
- **GraalVM native image atbalsts** - Java koda kompilēšana neatkarīgos bināros failos
- **Unificēts konfigurācijas modelis** - vienots JVM un natīvajam režīmam

- **Quarkus Extensions ekosistēma** - optimizētas komponentes ietvara paplašināšanai [5]

3.6.2 GraalVM natīvā kompilācija

Quarkus centrālā inovācija ir pilnvērtīgs atbalsts GraalVM natīvai kompilācijai, kas pārveido Java kodu mašīnkodā, iegūstot neticami ātru palaišanas laiku un zemu atmiņas patēriņu. Natīvās kompilācijas process veic vairākas būtiskas optimizācijas:

- **Ahead-of-Time (AOT) kompilācija** - kods tiek pārveidots mašīnkodā pirms izpildes
- **Dead code elimination** - neizmantotā koda noņemšana
- **Statiskā inicializācija kompilācijas laikā** - klašu inicializācija pirms izpildes
- **Refleksijas, JNI un Dynamic Proxies analīze** - statiska refleksijas gadījumu identifikācija [29]

Rezultātā palaišanas laiks ievērojami samazinās, tiek sasniegts minimāls atmiņas patēriņš, programma kļūst labāk piemērota bezservera pielietojumiem ar ātriem aukstajiem startiem un tiek būtiski samazināti konteineru izmēri.

Tomēr Quarkus pieejai pastāv arī trūkumi:

- Ierobežota refleksijas izmantošana
- Ierobežota dinamisko klašu ielādēšana
- Ilgāks kompilācijas process
- Augstākas prasības pret izstrādes vidi (RAM)

3.6.3 Resursu-orientēta REST API pieeja

Quarkus REST API implementācija balstās uz JAX-RS jeb Jakarta REST specifikāciju, atšķirībā no Spring Boot un Micronaut MVC arhitektūras. Šī pieeja uzsver resursu (nevis kontrolieru) centrālo lomu:

```java
@Path("/api/products")
@Produces(MediaType.APPLICATION_JSON)
@Consumes(MediaType.APPLICATION_JSON)
public class ProductResource {

    @Inject
    ProductService productService;

    @GET
    @Path("/{id}")
    public Response getById(@PathParam("id") Long id) {
        return productService.findById(id)
            .map(p -> Response.ok(p).build())
            .orElse(Response.status(Response.Status.NOT_FOUND).build());
    }

    @POST
    @Transactional
    public Response create(Product product) {
        productService.create(product);
        return Response.status(Response.Status.CREATED).entity(product).build();
    }
}
```

Attēls 3.12. "Quarkus resursa piemērs"

Quarkus REST (iepriekš pazīstams kā RESTEasy Reactive) ir labi piemērots gan tradicionālo, gan reaktīvo REST API vietņu izstrādei. [30]

3.6.4 Panache datu piekļuves modelis

Quarkus piedāvā Panache - Hibernate ORM paplašinājumu, kas ievieš Active Record paternu, tādejādi būtiski vienkāršojot datu piekļuves kodu. Tas ļauj definēt metodes tieši entītiju klasēs, novēršot nepieciešamību pēc atsevišķām repozitoriju klasēm. Attēlā 3.13. personas entītija manto PanacheEntity klasi, kurā ir jau iebūvētas CRUD operācijas kā statiskās metodes.

```java
@Entity
public class Person extends PanacheEntity {
    public String name;
    public LocalDate birth;
    public Status status;

    public static Person findByName(String name){
        return find("name", name).firstResult();
    }

    public static List<Person> findAlive(){
        return list("status", Status.Alive);
    }

    public static void deleteStefs(){
        delete("name", "Stef");
    }
}
```

Attēls 3.13. “Panache entitijas piemērs”

Panache arī ļauj izmantot tradicionālo repozitoriju paternu, ar Panache Repository klasēm. Šī pieeja nodrošina vairākas priekšrocības, salīdzinot ar standartiem CRUD vai JPA repozitoriju tipiem:

- **Tīra atdalīšana starp datu modeli un piekļuves loģiku** - entītijas paliek kā vienkāršas POJO klases
- **DI saderīgs modelis** - repozitoriji var tikt injicēti ar CDI
- **Samazināts boilerplate kods** - mantotās standartmetodes no PanacheRepository:
    - findAll() - visu ierakstu atgriešana
    - findById(id) - ieraksta meklēšana pēc ID
    - persist(entity) - ieraksta saglabāšana
    - delete(entity) - ieraksta dzēšana
    - count(), deleteAll(), listAll() u.c. [8]

3.6.5 Izstrādātāju produktivitāte

Quarkus izceļas ar inovatīviem rīkiem, kas būtiski uzlabo izstrādātāju produktivitāti un veicina ātru izstrādes ciklu:

Live Coding (hot reload) režīms

Šīs režīms būtiski atvieglo programmatūras izstrādi, jo: koda atjaunināšana notiek bez manuālas restartēšanas. Tas ļauj taupīt laiku, jo bieža programmas restartēšana ievērojami palēnina kopējo izstrādes procesu.

Dev Services

Dev Services ir Quarkus funkcionalitāte, kas automātiski nodrošina izstrādes infrastruktūru Docker konteineros. Tā nodrošina:

- **Automātisku konteineru palaišanu** datubāzēm, message brokeriem utt.
- **Automātisku konfigurāciju** savienojuma parametriem
- **Infrastruktūras pārvaldību** izstrādes procesa ietvaros

Dev UI

Quarkus Dev UI ir grafiska saskarne, kas pieejama izstrādes režīmā /q/dev ceļā. Tās īpašības ir:

- Reāllaika konfigurācijas vizualizācija un pārvaldība
- Endpointu pārlūks ar iespēju testēt API
- Veiktspējas un metriku monitorings
- Paplašinājumu pārvaldība un konfigurācija [31]

3.6.6 Cloud-native iespējas

Quarkus ir dziļi integrēts ar Kubernetes ekosistēmu, nodrošinot vienu no labākām cloud-native pieredzēm:

```
@Kubernetes
@KubernetesHealthCheck(readiness = true, liveness = true)
public class ApplicationConfig {
    // Konfigurācijas kods
}
```

Attēls 3.14. "Kubernetes veselības pārbaudes konfigurācijas piemērs"

Kubernetes funkcionalitāte:

- **Automātiska Kubernetes resursu ģenerēšana, balstoties uz anotācijām un aplikācijas konfigurāciju**:
  - Deployment
  - Service
  - Route
- **Health Check integrācija** ar MicroProfile Health
- **ConfigMap un Secret** automātisks mapping uz konfigurācijas parametriem
- **Service Discovery** ar Kubernetes DNS mehānismiem [6]

Service Resilience

```
@CircuitBreaker(requestVolumeThreshold = 4)
@Fallback(fallbackMethod = "fallbackRecommendations")
@Timeout(250)
public List<Product> getRecommendations() {
    ...
}
```

Attēls 3.15. "CircuitBreaker kļūmjnoturības funkcionalitāte"

Quarkus Service Resilience funkcionalitāte:

- **Circuit Breaker** - kļūmjnoturībai
- **Retry mehānisms** - atkārtotiem mēģinājumiem
- **Timeout pārvaldība** - ar deklaratīvām anotācijām
- **Fallback stratēģijas** - kļūmju gadījumiem [32]

3.6.7 Testēšanas infrastruktūra

Quarkus testēšanas ietvars fokusējas uz efektivitāti un ātru izpildi, piedāvājot vairākus testēšanas modeļus:

- **Rest-assured integrācija** - API testēšanai
- **Kontinuālās testēšanas režīms** - izstrādes procesā
- **Ātra konteksta ielādēšana starp testiem**
- **Native tests atbalsts** - natīvo bināro failu testēšanai
- **Test Resources integrēts atbalsts** - testēšanas atkarībām [33]

3.6.8 Monitoringa un diagnostikas iespējas

Quarkus nodrošina plašu monitoringa un diagnostikas iespēju klāstu:

- **MicroProfile Health** - standartizēta sistēmas veselības stāvokļa pārbaude:
  - /q/health/live - dzīvotspējas pārbaude
  - /q/health/ready - gatavības pārbaude
- **MicroProfile Metrics** - standartizēta metriku vākšana:
  - /q/metrics - Prometheus-formāta metrikas
  - Integrācija ar Micrometer (or SmallRye in older Quarkus versions)
- **OpenTelemetry integrācija** - distributed tracing:
  - Zipkin
  - Jaeger
- **Observability instrumentācija**:
  - JVM metrikas
  - Natīvu attēlu metrikas
  - Aplikācijas metrikas [34]

4. REST API VIETNE IETVARU PRAKTISKAI SALĪDZINĀŠANAI.

Pētījuma praktiskajā daļā tika izmantota empīriska pieeja, kuras mērķis ir maksimāli objektīvi novērtēt ietvaru stipras un vājas puses. Šī metodika kombinē divu komponentu analīzi:

1. **Vienotas testa lietojumprogrammas izstrāde** - identiska REST API servisa izveide ar katru no izvēlētajiem ietvariem, izmantojot vienādas vai maksimāli identiskas prasības, datu struktūras un funkcionalitāti. Šāda pieeja ļauj tiešā veidā salīdzināt ietvaru tehniskās iespējas, koda daudzumu un izstrādes pieredzi.
2. **Daudzfaktoru novērtēšana** - ietvaru vērtēšana pēc vienotas kritēriju kopas, aptverot gan tehniskos, gan praktiskos aspektus, kas ļauj identificēt katra risinājuma stiprās un vājās puses dažādos lietojuma scenārijos.

Praktiskā izpēte tika veikta, realizējot vienādu REST API funkcionalitāti trijos izvēlētajos ietvaros. Tika gan mērīti kvantitatīvie rādītāji (koda apjoms, startēšanas laiks, atmiņas patēriņš), gan vērtēti kvalitatīvie kritēriji (kā izstrādes ērtums un dokumentācijas kvalitāte).

Kā REST API vietni darba autors izvēlējas realizēt vienkāršu e-komercijas tīmekļa lietotni, kas sastāv no vietnes moduļa, kurš balstās uz viena no salīdzinājumiem ietvariem un datubāzes servera, hostēta uz PostgreSQL RDBMS (relāciju datubāzu pārvaldības sistēma).

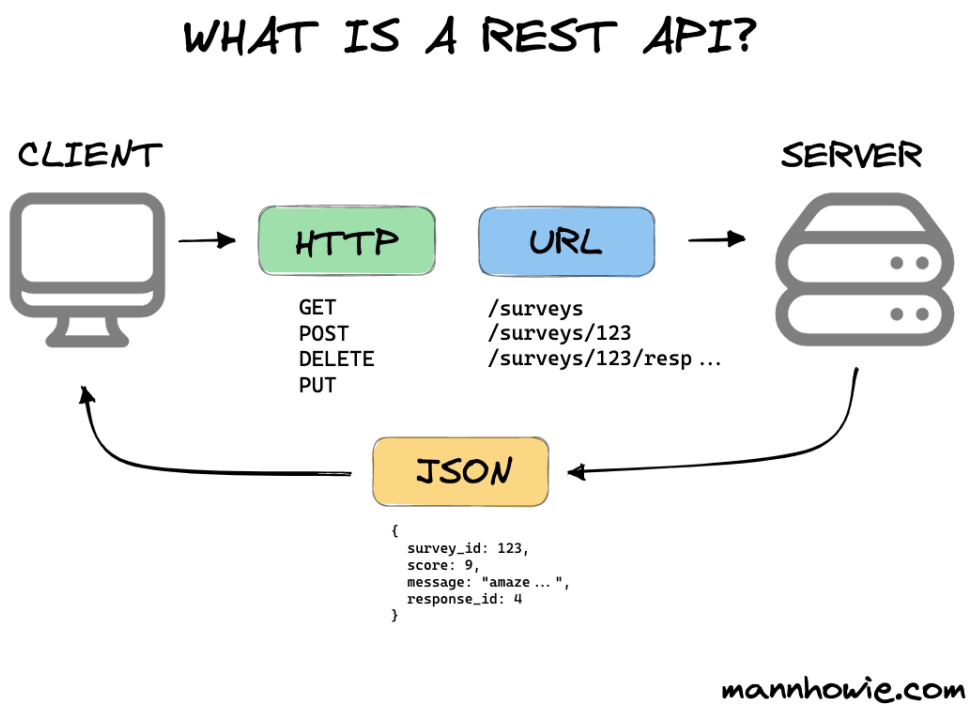
Pētījumā tika testēta ietvaru spēja nodrošināt dažādas aktuālas REST API vietņu izstrādes pieejas un paradigmas.

Vietnes izstrāde tika veikta Jetbrains IntellijId izstrādes vidē, kas šodien ir vispiemērotākā lietotne daudzveidīgu Java programmu izstrādei un piedāvā visplašāko izstrādes rīku komplektu. Piemēram Quarkus projektam tiek jau pēc noklusējuma izveidots Docker fails, un programma jau pēc noklusējuma tiek palaista konteinerā, nodrošinot visaugstāko programmas veiktspēju un zemāku resursu patēriņu. Šai videi ir arī ļoti ērts datubāzu integrāciju grafiskais interfeiss, kas ļauj minūtes laikā uztaisīt programmas savienojumu ar datubāzi un pārbaudīt, vai vide ir spējīga ar to komunicēt. Veidojot Java projektu šajā vidē, var jau uzreiz izvēlēties vajadzīgas atkarības.

4.1. Kāpēc REST API?

REST (Representational State Transfer) arhitektūras stils lietotnēm ir ideāla platforma ietvaru salīdzināšanai, jo tā ļauj atdalīt datu apstrādes un manipulācijas procesu no to reprezentācijas. Šāds koncepts ļauj:

1. Izstrādāt servera puses un klienta puses risinājumus neatkarīgi
2. Nodrošināt datus vairākām nesaistītām lietotnēm vienlaicīgi
3. Skaidri strukturēt pārbaudes scenārijus dažādiem ietvaru aspektiem



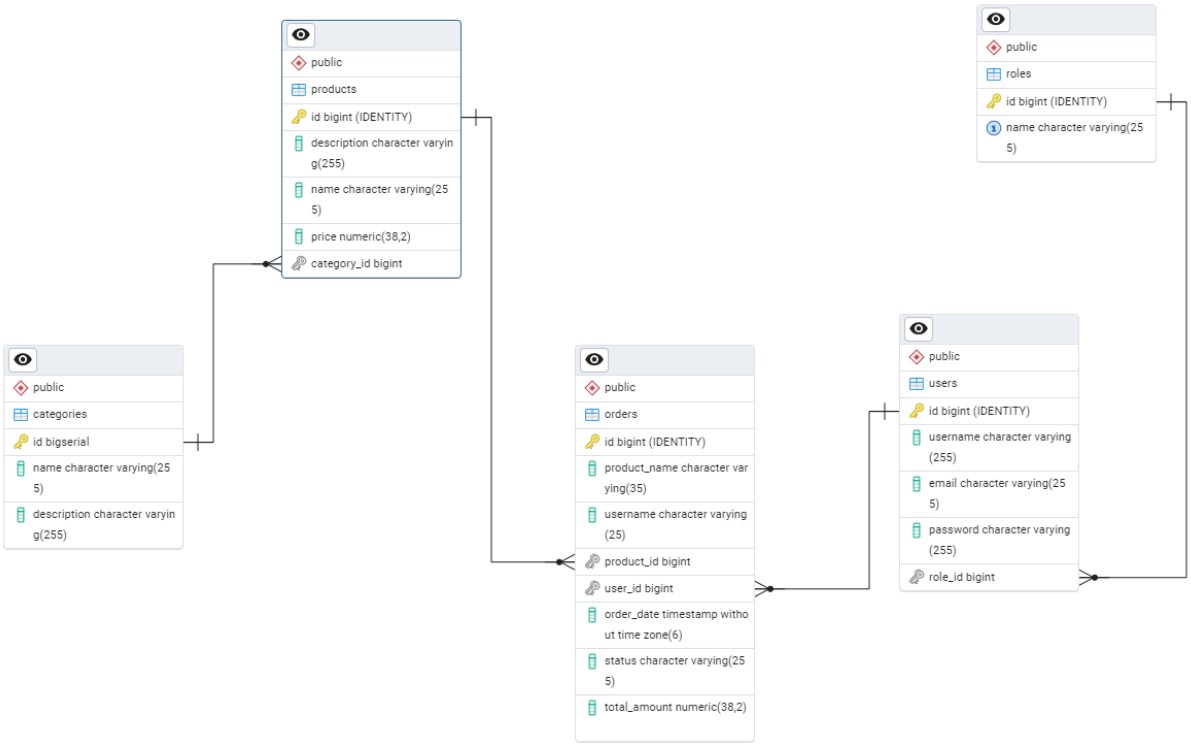
Attēls 4.1. "Standarta REST API darbības shēma" [9]

Pētījuma praktiskajā daļā izveidota sistēma ne pilnīgi atbilst klasiskajam REST API konceptam, jo papildus tīrajam datu apmaiņas slānim tajā ietverta arī HTML lapu ģenerēšanas funkcionalitāte.

Lai gan šāda pieeja nedaudz atkāpjas no REST arhitektūras pamatprincipiem, kas paredz pilnīgu datu reprezentācijas atdalīšanu no to apstrādes, autors uzskatīja par būtisku iekļaut arī šo aspektu salīdzinājumā. Šāds lēmums pamatojams ar to, ka reālās produkcijas sistēmās bieži nepieciešams apvienot gan tīru REST API funkcionalitāti, gan arī servera pusē ģenerētu HTML saturu. Tādējādi ietvaru veidņu dzinēju integrācijas salīdzinājums sniedz pilnīgāku priekšstatu par katra ietvara praktisko pielietojamību. Piemēram, Quarkus ietvars

tiek bieži lietots kopā ar tām piemēroto Qute dzinēju, tāpēc tā salīdzināšana ar citiem dzinējiem, kas tiek bieži pielietoti kopā ar citiem ietvariem būtu svarīga daļa šīm pētījumam. Tas ļaus novērtēt ne tikai ietvaru spēju apstrādāt JSON/XML datu plūsmas, bet arī to elastību situācijās, kur nepieciešama hibrīda pieeja ar servera pusē ģenerētu lietotāja saskarne. Šāds analīzes aspekts ir īpaši vērtīgs organizācijām, kas plāno migrēt no monolītiskajām aplikācijām uz modernāku arhitektūru, saglabājot atsevišķas servera pusē renderētas komponentes.

4.2. Vietnes datubāzes diagramma



Attēls 4.2. "Datubāzes diagramma"

Datubāze sastāv no produktiem, kategorijām, pasūtījumiem, kā arī no lietotājiem un to lomām ("USER un "ADMIN"). Šāda struktūra ļaus notestēt standartu REST API funkcionalitāti, veicot dažādu tabulu pieprasījumu un ietvaru drošības rīku realizāciju, piešķirot ierobežojumus dažām sistēmas funkcijām. Datubazē pirms testēšanas bija 25 kategorijas ar 300 produktiem.

4.3. Vietnes struktūra un vaicājumi.

Vietnes funkcionalitāte balstās uz divām galvenajām lapām - kategorijas un produkti. Mājaslapā tiek radītas visas eksistējošas kategorijas un uzspiežot uz tām, lietotājs var apskatīt šīs kategorijas produktus. Abās lapas pastāv rediģēšanas, dzēšanas un jaunu elementu pievienošanas funkcionalitāte, tātad tiek izmantoti GET, POST, PUT un DELETE tipu HTTP vaicājumi. Zemāk tiek aprakstīti vietnes pamatvaicājumi un to atbildes. Šie vaicājumi dod iespēju notestēt katra ietvara funkcionalitāti, veiktspēju un saņemt salīdzinājumam nepieciešamas metrikas.

4.1. tabula

Kategoriju saraksts		
URI	/	
Piekļuve	ALL	
HTTP	Atbilde	Atbildes nozīme
GET	200	OK

Caur šo URI var nokļūt pie kategoriju saraksta.

4.2. tabula

Produktu saraksts		
URI	/category/{categoryId}	
Piekļuve	ALL	
HTTP metode	Atbilde	Atbildes nozīme
GET	200	OK

Caur šo URI var nokļūt pie produktu saraksta.

4.3. tabula

Kategorijas izveide		
URI	/categories	
Piekļuve	ADMIN	
HTTP metode	Atbilde	Atbildes nozīme
POST	201	Created

Caur šo URI var izveidot jauno kategoriju

4.4. tabula

Produktu izveide		
URI	/products	
Piekļuve	ADMIN	
HTTP metode	Atbilde	Atbildes nozīme
POST	201	Created

Caur šo URI var izveidot jauno produktu

4.5. tabula

Kategoriju rediģēšanas forma		
URI	/categories/edit/{id}	
Piekļuve	ADMIN	
HTTP metode	Atbilde	Atbildes nozīme
GET	200	OK

Caur šo URI var apskatīt kategorijas rediģēšanas formu, kurā tiek ievadīti jauni dati.

4.6. tabula

Produktu rediģēšanas forma		
URI	/products/edit/{id}	
Piekļuve	ADMIN	
HTTP metode	Atbilde	Atbildes nozīme
GET	200	OK

Caur šo URI var apskatīt produkta rediģēšanas formu, kurā tiek ievadīti jauni dati.

4.7. tabula

Produktu rediģēšana		
URI	/products/{id}	
Piekļuve	ADMIN	
HTTP metode	Atbilde	Atbildes nozīme
PUT	200	OK
DELETE	204	No Content

Caur šo URI tiek veikta produktu izmaiņu saglabāšana, kā arī produktu dzēšana.

4.8. tabula

Kategoriju rediģēšana		
URI	/categories/{id}	
Piekļuve	ADMIN	
HTTP metode	Atbilde	Atbildes nozīme
PUT	200	OK
DELETE	204	No Content

Caur šo URI tiek veikta kategoriju izmaiņu saglabāšana, kā arī kategoriju dzēšana.

4.9. tabula

Produkta pasūtīšana		
URI	/order/{productId}	
Piekļuve	USER	
HTTP metode	Atbilde	Atbildes nozīme
POST	201	Created

Izsaucot šo vaicājumu, lietotājs pasūta produktu.

4.10. tabula

Pasūtījumu apskats		
URI	/orders	
Piekļuve	USER	
HTTP metode	Atbilde	Atbildes nozīme
GET	200	OK

Šīs vaicājums ļauj lietotājam apskatīt visus savus pasūtījumus un to statusu.

4.11. tabula

Visu pasūtījumu apskats		
URI	/admin/orders	
Piekļuve	ADMIN	
HTTP metode	Atbilde	Atbildes nozīme
GET	200	OK

Izsaucot šo vaicājumu, administrators var apskatīt visu lietotāju pasūtījumus.

4.12. tabula

Lietotāja autentifikācija		
URI	/login	
Piekļuve	ALL	
HTTP metode	Atbilde	Atbildes nozīme
GET	200	OK
POST	200	OK

Izsaucot šo vaicājumu, lietotājs piekļūst autentifikācijas formai, kur ir jāievada reģistrēta lietotāja datus (GET vaicājums) un nosūta savus datus serverim (POST vaicājums).

4.13. tabula

Lietotāja reģistrācija		
URI	/register	
Piekļuve	ALL	
HTTP metode	Atbilde	Atbildes nozīme
GET	200	OK
POST	201	Created

Izsaucot šo vaicājumu, lietotājs piekļūst reģistrācijas formai, kur ir jāievada jaunā lietotāja datus (GET vaicājums) un nosūta tos serverim, kas reģistrē jauno lietotāju datubāzē (POST vaicājums).

4.14. tabula

Lietotāja izrakstīšanās		
URI	/logout	
Piekļuve	USER	
HTTP metode	Atbilde	Atbildes nozīme
POST	200	OK

Caur šo vaicājumu lietotājs izrakstās no sava konta, kļūstot par anonīmo lietotāju.

4.15. tabula

Produktu meklēšana		
URI	/category/{categoryId}/search	
Piekļuve	ALL	
HTTP metode	Atbilde	Atbildes nozīme
GET	200	OK

Izsaucot šo vaicājumu, var meklēt produktus pēc atslēgas vārdiem.

4.4. Lietotāju autentifikācija un drošības risinājumi

Lietotāju autentifikācijai tika izmantota formu bāzēta pieeja. Lietotāja entītija sastāv no vārda, e-pasta, paroles un saitēm ar pasūtījumu un lomu entītijām. Lietotāju lomas ir "ADMIN" administratoriem un "USER" parastiem lietotājiem un tās tiek glabātas UserRole tabulā.

```java
@Entity
@Table(name = "users")
@Data
public class User {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    @Column(insertable=false, updatable=false)
    private Long id;

    @Column(nullable = false, unique = true)
    private String username;

    @Column(nullable = false, unique = true)
    private String email;

    @Column(nullable = false)
    private String password;

    @OneToMany(mappedBy = "user", cascade = CascadeType.ALL)
    private List<Order> orders = new ArrayList<>();

    @ManyToOne(fetch = FetchType.EAGER)
    @JoinColumn(name = "role_id")
    private UserRole role;

    public void addUserRole(UserRole adminUserRole) {
        this.role = adminUserRole;
    }
}
```

Attēls 4.3. "Lietotāja entītija Spring Boot ietvarā"

4.5. Ietvaru izstrādes risinājumi

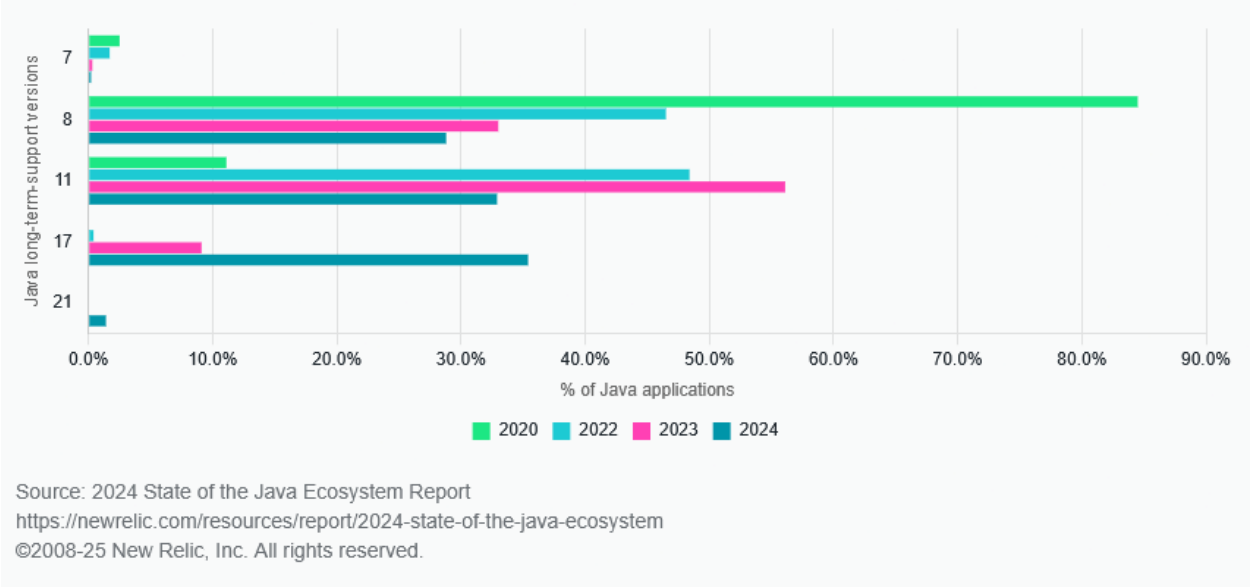
4.5.1. Projekta struktūra

Izstrādei tika izmantoti ietvariem raksturīgi REST API vietņu "Separation of Concerns" principa izstrādes paterni: Spring Boot un Micronaut vietnes balstās uz savu MVC modeli ar kategoriju, produktu, lietotāju un citu entītiju modeļiem, kontrolieriem, skatiem, servisiem, repozitorijiem un DTO (Data Transfer Object) klasēm. Quarkus vietnes struktūra ir ļoti līdzīga, taču izmanto nedaudz atšķirīgo JAX-RS pieeju, aizvietojot kontrolierus ar resursiem un izmantojot citas klašu anotācijas, kā arī Active Record pieeju, kas ļauj izvairīties

no repozitoriju lietošanas. Atbilstoši katra ietvara prasībām, tika pievienotas konfigurācijas klases, piemēram drošības konfigurācija.

4.5.2. Konfigurācija

Visu triju projektu konfigurācijai un būvēšanai tika izmantots Apache Maven atvērtā pirmkoda projekta pārvaldības un būvēšanas rīks, kas ir visbiežāk izmantots Java projektiem. [10] Visi projekti balstās uz Java 17 ar Oracle OpenJDK (Java Development Kit) 17 versiju. Šī ir LTS (Long Time Support) Java valodas versija, kas iznāca 2021. gada septembrī. Tā ir stabila un moderna Java valodas versija, kas tiek plaši izmantota mūsdienīgās Java programmās. Iepriekšējā gadā tā ir kļuvusi par vispopulārāko versiju Java pasaulē un tās popularitāte turpina pieaugt. Jaunāka Java 21 versija, kas arī ir LTS versija, pagaidām ir lietota daudz retāk un varētu būt vājāk optimizēta integrācijai ar vairākām servisiem un bibliotēkām, tāpēc tās izmantošana nebūtu vislabākais variants šim pētījumam. Oracle OpenJDK arī ir viens no populārākiem JDK vendoriem.



Attēls 4.4. "Java valodas versiju tirgus sadalījums pēdējos gados" [11]

4.5.3. Izmantotas tehnoloģijas

Skatu apstrādei Spring Boot un Micronaut ietvari izmanto Thymeleaf dzinēju, kas ir viens no populārākiem skatu dzinējiem Java MVC lietotņu izstrādē. Quarkus izmanto iebūvēto Qute dzinēju, kas ir vislabāk piemērots šīm ietvaram.

Datu piekļuvei Spring Boot un Micronaut ietvari izmanto JPA tipa repozitorijus, nodrošinot klasisko datu piekļuves veidu, bet Quarkus izmanto savu PanacheEntity modeļu tipu ar Active Record paternu.

Lietotāju autentifikācijai un autorizācijai un piekļuves kontrolei tika izmantoti katra ietvara iebūvētie drošības mehānismi, kas tika sakonfigurēti atbilstoši pētījuma mērķiem.

4.6. Mērījumu metrikas

Pētījuma rezultātu analīzei tika izvelētas sekojošās metrikas:

1. **Uzstādīšanas sarežģītība** – cik viegli vai grūti ir uzstādīt un konfigurēt ietvaru
2. **Koda apjoms** – funkcionalitātes un konfigurācijas koda apjoms
3. **Izstrādes laiks** – cik ilgs laiks nepieciešams līdzvērtīgas funkcionalitātes izveidei
4. **Palaišanas laiks** – cik ātri lietotne startējas (īpaši svarīgi konteineru vidē)
5. **Atmiņas patēriņš** – cik daudz resursu patērē aplikācija darbības laikā
6. **Caurlaidība jeb vaicājumu atbildes ātrums** – cik ātri ietvars apstrādā HTTP pieprasījumus (definētus 4.3. sadaļā)

Tās atspoguļo vairākus svarīgus rādītājus, kurus izstrādātāji parasti ņem vērā, izvēloties sev piemērotāku ietvaru.

5. PRAKTISKA SALĪDZINĀJUMĀ REZULTĀTI

Šī pētījuma praktiska daļa ietvēra sevī triju REST API vietņu izstrādi ar katru no salīdzināmiem ietvariem. Vietņu struktūra un izstrādes risinājumi ir maksimāli identiski un atbilst 4. sadaļa aprakstam un projektējumam. Veiktspējas radītāji tika saņemti, palaižot katra ietvara vietni un izpildot visus 4. sadaļā definētus HTTP pieprasījumus. Tika arī izmērīts atsevišķu 4 tipu pieprasījumu laiks sekundēs, lai parādītu, cik ātri ietvars apstrādāja katra no 4 izmantotiem HTTP vaicājumu tipiem atsevišķu HTTP vaicājumu. Katra vietne tika palaista gan ar standartu JVM virtuālo mašīnu ar IntellijId vides noklusētiem iestatījumiem, gan ar GraalVM virtuālo mašīnu, pielietojot natīvo kompilāciju un palaižot vietni caur Docker konteineri. Natīva kompilācija tika veikta ar Maven rīka *mvn package -Pnative* konsoles komandu. Testi tika veikti uz Windows operētājsistēmas, datorā ar 16 GB RAM.

5.1. Spring Boot ietvara rezultāti

5.1.1. Uzstadīšana un konfigurēšana

Spring projekts tiek palaists caur galveno klasi ar @SpringBootApplicaiton anotāciju. Lai tas darbotos, ir nepieciešamas vairākas atkarības, kas Maven rīka izmantošanas gadījumā atradās pom.xml failā. Darbā tika izmantota Spring Boot 3.4.3. versija. Zemāk ir redzamas šī projekta uzstādīšanai bija nepieciešamas atkarības ar to identifikatoriem un versijām:

- spring-boot-starter-web (3.4.3. versija) vajadzīga, lai iestartētu Spring Boot tīmekļa vietni
- spring-boot-starter-test (3.4.3. versija) vajadzīga, lai testētu Spring Boot vietni
- spring-boot-starter-data-jpa (3.4.3. versija) vajadzīga, lai izmantotu JPA tipa repozitorijus datubāzes integrācijai
- spring-boot-starter-actuator (3.4.3. versija) vajadzīga, lai pievienotu Spring Boot Actuator monitoringa rīku
- spring-boot-starter-validation (3.4.3. versija) vajadzīga pupu validācijai
- postgresql (42.7.5. versija) vajadzīga, lai integrētu PostgreSQL datubāzi
- spring-boot-starter-thymeleaf (3.4.3. versija) vajadzīga Thymeleaf skatu apstrādei
- spring-boot-starter-security (3.4.3. versija) vajadzīga drošas autentifikācijas/autorizācijas pārvaldei.

5.1.2. Darbietilpība

Tā kā autoram ir bija iepriekšēja izstrādes pieredze ar šo ietvaru, šīs vietnes izstrāde aizņēma mazāk laika, salīdzinot ar pārējiem trim. Par konfigurācijas kodu šajā darbā tiek skaitīti konfigurācijas faili vai klases (XML, YAML, application.properties, SecurityConfig.java, DataInitializer.java), anotācijas (@Configuration, @Entity).
Par funkcionalitātes kodu – visas metodes, anotācijas vai lauki, kas implementē kādu funkcionalitāti.

Tabulā 5.1. ir redzamas Spring Boot darbietilpības metrikas

5.1. tabula

Izstrādes laiks	Konfigurācijas koda apjoms	Funkcionalitātes koda apjoms
48 stundas	137 rindiņas	389 rindiņas

5.1.3. Veiktspēja

5.2. tabula

Startēšanas laiks	Atmiņas izmantošana	Caurlaidība (vidējais atbildes laiks HTTP pieprasījumam)
6,116 s	Heap 86,21 MB Non-heap: 123,56MB	68,97 req/s

Tabula 5.3. atsevišķi parāda atbilžu laiku uz dažādu tipu HTTP pieprasījumiem. Katrs no tiem tika veikts 10 reizes.

5.3. tabula

Endpoint	Tips	Laiks
/category/{categoryId}	GET	0,399 s
/order/{productId}	POST	0,202 s
/products/{id}	PUT	0,130 s
/products/{id}	DELETE	0,145 s

5.1.4. GraalVM natīvā kompilācija

Lai palaistu Spring Boot vietni uz GraalVM, bija jāpievieno natīvas kompilācijas spraudnis “native-maven-plugin”. [35] Zemāk ir redzamas metrikas Spring Boot vietnei, kas tika iedarbināta ar natīvo kompilāciju.

5.4. tabula

Startēšanas laiks	Atmiņas izmantošana	Caurlaidība (vidējais atbildes laiks HTTP pieprasījumam)
6,288 s	Heap 68,52 MB Non-heap 128,41 MB	62,03 req/s

5.1.5. Dokumentācija

No visiem trijiem ietvariem Spring Bootam ir visplašākā dokumentācija, apmācības, risinājumu piemēri utt., kas arī atvieglo apmācību un neskaidrību saprašanu programmas izstrādes laikā. Oficiālā dokumentācijas vietne https://docs.spring.io ir plaši un saprotami aprakstīti visi ietvara aspekti un ar kodu piemēriem tiek demonstrēta daudzu risinājumu implementācija. Pastāv arī vairāki apmācoši portāli, kā piemēram https://www.baeldung.com kur arī tiek skaidroti Spring Boot ietvara risinājumi un tehnoloģijas, un pastāv daudz instrukciju, kā tos pareizi izmantot savā projektā. Tā kā Spring Boot ir vispopulārākais Java tīmekļa ietvars, tīmekļa forumos, tādos kā https://stackoverflow.com/ ir atradāmas vairākas tēmas par problēmām un neskaidrībām, saistītiem ar Spring Boot programmu izstrādi un, saskaroties ar tām, ir liela varbūtībā, ka ar tādām problēmām ir jau saskaris kāds cits izstrādātājs un ir saņēmis skaidrojumu, kā tos atrisināt.

5.1.6. Drošība

Spring Security ietvars ir viegli uzstādāms un ērts lietošanai, nodrošinot drošu lietotāju autentifikāciju/autorizāciju un pārvaldot piekļuvi kontrolieru metodēm. Tajā ir iebūvēti rīki paroļu apstrādei un validācijai, ļaujot lietotājam ietaupīt laiku, nekonfigurējot tos manuāli. Drošības konfigurācija sastāv no pupām passwordEncoder (paroļu kodēšana), authenticationManager (autentifikācijas pārvaldei), filterChain (drošības noteikumu definēšanai).

Lai atdalītu metodes, kas ir pieejami visiem no metodēm, kurus var izsaukt tikai administrators vai ielogots lietotājs tika izmantotas @PreAuthorize anotācijas, kas pārbauda esoša lietotāja statusu, pirms izsaucot metodi.

5.1.7. Priekšrocības un trūkumi

Kā jau tika pieminēts, galvenā Spring Boot ietvara priekšrocība ir tā popularitāte un uzticamība. Par to ir pieejams visvairāk dokumentāciju un apmācību, tāpēc šis ietvars var būt vieglāks apgūšanai. Tas ir labi pielāgots populārajām tehnoloģijām un pieejām, taču vairākas modernākas tehnoloģijas, piemēram GraalVM nātīvā kompilācija varētu prasīt grūtāko konfigurāciju. Spring Boot arī prasa vairāk resursu un darbojas lēnāk, nekā Micronaut vai Quarkus,

5.2. Micronaut ietvara rezultāti

5.2.1. Uzstādīšana un konfigurēšana

Šajā darbā tika izmantota Micronaut ietvara 4.7.6. versija. Projekta veiksmīgai darbībai bija jāpievieno sekojošas atkarības:

- micronaut-http-client (4.8.12. versija) vajadzīga Micronaut ietvara kodolam, lai palaistu Micronaut tīmekļa vietni
- micronaut-http-server-netty (4.8.12. versija) vajadzīga Netty servera pievienošanai
- micronaut-security (4.8.12. versija) vajadzīga Micronaut Security autentifikācijas rīku izmantošanai
- micronaut-jdbc-hikari (6.1. versija) vajadzīga SQL datubāzes darbībām
- postrgresql (42.7.5. versija) vajadzīga, lai integrētu PostgreSQL datubāzi
- micronaut-views-thymeleaf (1.3.2 versija) vajadzīga Thymeleaf skatu apstrādei
- micronaut-data-hibernate-jpa (4.12.0. versija) vajadzīga, lai izmantotu JPA tipa repozitorijus datubāzes integrācijai
- micronaut-security-jwt (4.12.0. versija) vajadzīga JWT tokenu izveidei
- spring-security-core (6.4.6. versija) vajadzīga, lai implementētu paroļu kodēšanas mehānismu
- micronaut-micrometer-core (5.10.0 versija) vajadzīga, lai saņemtu metrikas
- micronaut-management (4.8.12 versija) vajadzīga programmas pārvaldei
- lombok (1.18.36. versija) vajadzīga Lombok anotāciju izmantošanai

5.2.2. Darbietilpība

Šī ietvara vietnes izstrāde autoram aizņēma visvairāk laikā, jo nebija pietiekamas izstrādes pieredzes, kā arī ir bijušas grūtības ar drošības funkcionalitātes implementāciju ietvara minimālistiskas pieejas dēļ.

Tabulā 5.4. ir redzamas Micronaut darbietilpības metrikas

5.5. tabula

Izstrādes laiks	Konfigurācijas koda apjoms	Funkcionalitātes koda apjoms
57 stundas	192 rindiņas	418 rindiņas

5.2.3. Veiktspēja

5.6. tabula

Startēšanas laiks	Atmiņas izmantošana	Caurlaidība (vidējais atbildes laiks HTTP pieprasījumam)
4,516 s	Heap 85,59 MB Non-heap 114,79 MB	128 req/s

Tabula 5.7. atsevišķi parāda atbilžu laiku uz dažādu tipu HTTP pieprasījumiem. Katrs no tiem tika veikts 10 reizes.

5.7. tabula

Endpoint	Tips	Laiks
/category/{categoryId}	GET	0,146 s
/order/{productId}	POST	0,134 s
/products/{id}	PUT	0,115 s
/products/{id}	DELETE	0,071 s

5.2.4. GraalVM natīvā kompilācija

5.8. tabula

Startēšanas laiks	Atmiņas izmantošana	Caurlaidība (vidējais atbildes laiks HTTP pieprasījumam)
2,767 s	Heap 66,39 MB Non-heap 102,64 MB	125,33 req/s

5.2.5. Dokumentācija

Micronaut ietvara pieejamais dokumentācijas apjoms ir mazāks, salīdzinot ar citiem diviem ietvariem. Lai gan oficiālā dokumentācija ir labi strukturēta un piedāvā vairākas apmācības ar skaidriem piemēriem, dažreiz pietrūkst citus apmācības avotus, kas plašāk skaidrotu konkrētus izstrādes aspektus, lai palīdzētu atrisināt radījušas problēmas. Tā kā šis ietvars ir jaunāks, tam nav tik plašas kopienas kā Spring Boot ietvaram, tāpēc atrast vajadzīgo informāciju var būt grūtāk. Piemērām https://www.baeldung.com platformā Micronaut dokumentācijas ir ievērojami mazāk, salīdzinot ar populārāku Spring Boot ietvaru.

5.2.6. Drošība

Implementēt Micronaut drošības mehānismus izradījās grūtāk, nekā Spring Boot ietvarā. Spring Boot piedāvā iebūvētu autentifikācijas funkcionalitāti, kas tiek implementēta, pievienojot vajadzīgas drošības atkarības. Micrnaut, savukārt, balstās uz "lightweight" pieejas un cenšas nepārslogot kopējo sistēmu, ielādējot vairākas klases. Tāpēc autentifikācijai vajadzīgas klases un pupas, tādas kā AuthenticationProviderUserPassword un BCryptPasswordEncoderService ir jādefinē manuāli. Micronaut jaunākās versijas tika pārstāts atbalsts klasēm *io.micronaut.security.authentication.providers.PasswordEncoder;* un *io.micronaut.security.authentication.providers.BCryptPasswordEncoder;* Tāpēc lai izmantotu BCrypt algoritmu ir jāimportē ārējas klases, piemēram Spring Boot ietvara BCrypt klasi vai klasi org.mindrot.jbcrypt.BCrypt, vai arī jādefinē to pašam. Tas sarežģī izstrādi, tomēr ļauj saņemt augstākas vietnes veiktspējas radītājus, jo importējamas bibliotēkas un metodes mazāk ietekmē startēšanas laiku un atmiņas izmantošanu. Micronaut Security piedāvā gan JWT, gan OAuth2 tipu drošības risinājumus.

5.2.7. Priekšrocības un trūkumi

Galvenā Micronaut ietvara priekšrocība ir tā revolucionāra pieeja atkarību injekcijām, kas rezultējas mazākā palaišanas laikā un zemākā resursu patēriņā, salīdzinot ar Spring Boot ietvaru, kas balstās uz refleksijas mehānismiem. Micronaut demonstrē vislabākos rezultātus caurlaidībā, kas nozīmē, ka šis ietvars ir spējīgs visātrāk apstrādāt HTTP vaicājumus. Tā MVC modelis ir līdzīgs vairākiem citiem ietvariem, kas atvieglo migrāciju. Tomēr tas pēc noklusējuma neietver sevī vairākas metodes un bibliotēkas, kas dažreiz prasa papildus konfigurāciju. Piemēram Lombok anotāciju rīka konfigurācija Micronaut ietvara autoram kļuva par izaicinājumu. Micronaut arī neatbalsta savu PasswordEncoder interfeisu, kas šajā darba ietvaros tika izmantots paroļu apstrādei. Tātad izstrādātājam ir izvēlē konfigurēt to manuāli vai izmantot Spring Boot ietvara bibliotēku.

```java
@Singleton
public class BCryptPasswordEncoderService implements PasswordEncoder {

    PasswordEncoder delegate = new BCryptPasswordEncoder();

    @Override
    public String encode(@NotBlank @NotNull CharSequence rawPassword) {
        return delegate.encode(rawPassword);
    }

    @Override
    public boolean matches(@NotBlank @NotNull CharSequence rawPassword,
                           @NotBlank @NotNull String encodedPassword) {
        return delegate.matches(rawPassword, encodedPassword);
    }
}
```

Attēls 5.1. "Micronaut paroļu apstrādes konfigurācija"

5.3. Quarkus ietvara rezultāti

5.3.1. Uzstādīšana un konfigurēšana

Quarkus ietvara projekts nedaudz atšķiras pēc savas struktūras no Spring Boot un Micronaut projektiem. IntellijId izstrādes vidē Quarkus projektam tiek automātiski noģenerēts Dockerfile, jo konteinerizācijas izmantošana ļauj vislabāk izmantot Quarkus ietvara priekšrocības. Tika izmantota Quarkus 3.21.0. versija.

Projektam tika pievienotas sekojošas atkarībās:

- quarkus-rest (3.21.0. versija) vajadzīga REST API funkcionalitātes izmantošanai
- quarkus-elytron-security-jdbc (3.21.0. versija) vajadzīga drošai lietotāju datu glabāšanai
- quarkus-jdbc-postgresql (3.21.0. versija) vajadzīga, lai integrētu PostgreSQL datubāzi
- quarkus-arc (3.21.0. versija) vajadzīga atkarību injekcijai
- quarkus-hibernate-orm (3.21.0. versija) vajadzīga Hibernate ORM datubāzu rīku izmantošanai
- quarkus-hibernate-orm-panache (3.21.0. versija) vajadzīga Panache repozitoriju izmantošanai
- quarkus-qute (3.21.0. versija) vajadzīga Qute dzinēja skatu izmantošanai
- quarkus-rest-qute (3.21.0. versija) vajadzīga, lai apstrādātu Qute skatus caur rest endpointiem

- quarkus-security (3.21.0. versija) vajadzīga, lai implementētu Quarkus drošības mehānismus
- quarkus-micrometer-registry-prometheus (versija 3.21.0) vajadzīga metriku savākšanai
- lombok (1.18.36. versija) vajadzīga Lombok anotāciju izmantošanai

5.3.2. Darbietilpība

5.9. tabula

Izstrādes laiks	Konfigurācijas koda apjoms	Funkcionalitātes koda apjoms
54 stundas	151 rindiņa	385 rindiņas

5.3.3. Veiktspēja

Metriku saņemšanai tika izmantots Quarkus Micrometer rīks.

5.10. tabula

Startēšanas laiks	Atmiņas izmantošana	Caurlaidība (vidējais atbildes laiks HTTP pieprasījumam)
5,695 s	Heap 121.84 MB Nonheap 106,13 MB	70,28 req/s

Tabula 5.11. atsevišķi parāda atbilžu laiku uz dažādu tipu HTTP pieprasījumiem. Katrs no tiem tika veikts 10 reizes.

5.11. tabula

Endpoint	Tips	Laiks
/category/{categoryId}	GET	0,216 s
/order/{productId}	POST	0,266 s
/products/{id}	PUT	0,214 s
/products/{id}	DELETE	0,211 s

5.3.4. GraalVM natīvā kompilācija

5.12. tabula

Startēšanas laiks	Atmiņas izmantošana	Caurlaidība (vidējais atbildes laiks HTTP pieprasījumam)
0,643 s	Heap 67,43 MB Non-heap 50,76 MB	68,23 req/s

5.3.5. Dokumentācija

Quarkus dokumentācija ir pārsvarā pietiekami skaidra un labi organizēta. Pēc ietvara izstrādātāju vārdiem, viņi piedāvā "Developer Joy" pieeju, kuras mērķis ir atvieglot izstrādi, padarot katru ietvara īpašību par intuitīvu un viegli konfigurējamu. [12] Pēc autora pieredzes, vajadzīgo apmācību par Quarkus ietvara konfigurāciju bija atrast nedaudz vieglāk, salīdzinot ar Micronaut ietvaru. Tomēr tā joprojām nav tik plaša, ka Spring Boot ietvaram, un tā kā Quarkus ir visjaunākais no visiem trijiem, tā kopiena nav tik plaša un pieredzoša.

5.3.6. Drošība

Quarkus Security dod iespēju sakonfigurēt autentifikācijas funkcionalitāti *application.properties* failā ar vajadzīgām īpašībām. Tiek atbalstīti vairāki autentifikācijas veidi:

- Pamata (Basic) autentifikācija
- Formu bāzēta autentifikācija
- Savstarpēja (Mutual) TLS autentifikācija

Šajā projektā tika izmantota formu bāzēta autentifikācija, jo tā ir tuva tradicionālai Java Servlet formu bāzētai autentifikācijai, tomēr pastāv atšķirības. Tā nesaglabā lietotāja datus, jo Quarkus neatbalsta HTTP sesijas. Lietotāja autentifikācijas informācija tiek glabāta šifrētās sīkdatnes, ko var nolasīt caur atslēgu. [13]

5.3.7. Priekšrocības un trūkumi

Quarkus ietvaram ir vairākas priekšrocības, salīdzinot ar citiem Java balstītiem ietvariem:

- Fokuss uz konteinerizāciju – jau “no kastes” piemērots un optimizēts mākoņrisinājumu implementācijai, modernas vides, kā IntellijId jau automātiski ģenerē Dockerfile Quarkus projekta izveides laikā.
- Live Coding – izradījās ļoti noderīga funkcija, kas nopietni samazina izstrādes laiku, jo vairākkārtējā projekta pārstartēšana aizņem nemaz laika, pat tādiem minimālistiskiem ietvariem, kā Micronaut.
- Ietvars nodrošināja vismazāko atmiņas patēriņu GraalVM natīvas kompilācijas režīmā, demonstrējot savas priekšrocības konteinerizācijas vidē. Tām arī ir bijis vismazākais startēšanas laiks GraalVM režīmā, tomēr ir jāņem vērā, ka natīvā attēla būvēšana aizņem vairāk laikā.

No trūkumiem varētu pieminēt to, ka ietvars neparāda īpaši labus veiktspējas rezultātus, darbojoties uz klasiskas JVM virtuālas mašīnas.

5.4. Koda struktūras salīdzinājums

Kopumā visu triju ietvaru koda struktūra sanāca ļoti līdzīga. Modeļi definē datubāzes entītijas un to attiecības. Kontrolieri (vai resursi) definē endpointus un servisu metožu izmantošanu. Servisos tiek realizēta galvenā programmas funkcionalitāte, ieskaitot darbības ar datubāzi. DTO ļauj pārnest datus starp dažādiem programmas slāņiem. Spring Boot un Micronaut ietvaros pastāv arī repozitoriji, kas palīdz veikt operācijas ar datubāzi. Savukārt Quarkus ietvara vietnē izmantots Active Record paterns un repozitoriju funkcionalitāte tika realizēta modeļos.

Piemēram attēlā 5.2. ir kategorijas dzēšanas metode Quarkus ietvarā. Piekļuves kontrole tiek nodrošināta ar @RolesAllowed anotāciju (@Secured Micronautā un @PreAuthorize Spring Bootā). Ja lietotājs nav ielogots, vai viņam nav administratora lomas, programma atgriezīs 403 Forbidden. Šajā metodē, kā arī visos parējos tiek veikta kļūdainu pieprasījuma apstrāde. @Transactional anotācija norāda, ka metode izpildās transakcijas ietvaros - ja kaut kas neizdodas, izmaiņas tiks atceltas.

```java
@DELETE
@Path("/categories/{id}")
@Transactional
@RolesAllowed("ADMIN")
public Response deleteCategory(@PathParam("id") Long id) {
    try {
        categoryService.deleteCategory(id);
        return Response.seeOther(URI.create("/")).build();
    } catch (Exception e) {
        String encodedError = URLEncoder.encode(e.getMessage(), StandardCharsets.UTF_8);
        return Response.seeOther(URI.create("/?error=" + encodedError)).build();
    }
}
```

Attēls 5.2. "Kontroliera metodes implementācija"

Visiem trijiem ietvariem sanāca atšķirīga drošas autentifikācijas konfigurācija.

Spring Boot drošības bibliotēkas piedāvā iebūvēto klasi AuthenticationConfiguration un PasswordEncoder, kas ļauj viegli sakonfigurēt drošības mehānismus.

```java
@Configuration
@EnableWebSecurity
@EnableMethodSecurity
public class SecurityConfig {
    @Autowired
    private CustomUserDetailsService userDetailsService;
    @Bean
    public static PasswordEncoder passwordEncoder() {
        return new BCryptPasswordEncoder();
    }
    @Bean
    public AuthenticationManager authenticationManager(AuthenticationConfiguration authConfig) throws Exception {
        return authConfig.getAuthenticationManager();
    }
    @Bean
    public SecurityFilterChain filterChain(HttpSecurity http) throws Exception {
        http
                .authorizeHttpRequests(authorize ->
                        authorize
                                .requestMatchers("/css/**", "/js/**", "/images/**").permitAll()
                                .requestMatchers("/", "/register/**").permitAll()
                                .requestMatchers("/category/**").permitAll()
                                .requestMatchers("/products/search").permitAll()
                                .requestMatchers("/", "/register", "/register/**", "/login").permitAll()
                                .requestMatchers("/actuator/**", "/prometheus").permitAll()
                                .requestMatchers("/admin/**").hasRole("ADMIN")
                                .anyRequest().authenticated()
                )
                .formLogin(form ->
                        form
                                .loginPage("/login")
                                .loginProcessingUrl("/login")
                                .defaultSuccessUrl("/")
                                .permitAll()
                )
                .logout(logout ->
                        logout
                                .logoutSuccessUrl("/")
                                .permitAll()
                );
        return http.build();
    }
}
```

Attēls 5.3. "Spring Boot drošības konfigurācija"

Micronaut ietvars ar savu minimalistisko pieeju prasa vairāku manuālo konfigurāciju, tāpēc tām bija jāizveido atsevišķas klases, kas realizē šo funkcionalitāti. Tā ietvēra sevī *AuthenticationProviderUserPassword* un *BCryptPasswordEncoderService* klašu izveidi.

```java
@Singleton
public class AuthenticationProviderUserPassword<B> implements HttpRequestAuthenticationProvider<B> {

    private final UserRepository userRepository;
    private final PasswordEncoder passwordEncoder;

    public AuthenticationProviderUserPassword(UserRepository userRepository, PasswordEncoder passwordEncoder) {
        this.userRepository = userRepository;
        this.passwordEncoder = passwordEncoder;
    }

    @Override
    public AuthenticationResponse authenticate(
            @Nullable HttpRequest<B> httpRequest,
            @NonNull AuthenticationRequest<String, String> authenticationRequest
    ) {
        String identity = authenticationRequest.getIdentity();
        String secret = authenticationRequest.getSecret();

        Optional<User> userOptional = userRepository.findByUsername(identity);

        if (userOptional.isPresent()) {
            User user = userOptional.get();
            if (passwordEncoder.matches(secret, user.getPassword())) {
                return AuthenticationResponse.success(user.getUsername(), List.of(user.getRole().getName()));
            }
        }

        return AuthenticationResponse.failure(AuthenticationFailureReason.CREDENTIALS_DO_NOT_MATCH);
    }
}
```

Attēls 5.4. "AuthenticationProviderUserPassword klase" [37]

Quarkus ietvars ietver sevī iebūvētas autentifikācijas mehānismus un, atšķirībā no Spring Boot ietvara, kurā autentifikācija tiek nodrošināta caur anotācijām un pupu injekcijām, autentifikācijas mehānisms tiek nodefinēts application.properties failā ar sekojošām īpašībām:

quarkus.http.auth.form.enabled=true
quarkus.http.auth.form.login-page=/login
quarkus.http.auth.form.error-page=/login?error=true
quarkus.http.auth.form.landing-page=/
quarkus.http.auth.form.username-parameter=j_username
quarkus.http.auth.form.password-parameter=j_password
quarkus.http.auth.form.post-location=/j_security_check

HTML lapu struktūra visām vietnēm sastāv no *index.html*, *products.html*, *orders.html*, *login.html* un *register.html* failiem. Gan Thymeleaf, gan Qute dzinēji piedāvā diezgan intuitīvu sintaksi.

HTTP metožu filtri tika pievienoti, lai programma varētu apstrādāt PUT un DELETE pieprasījumus.

Tabulā 5.13. var redzēt koda sadalījumu starp dažādu failu grupām.

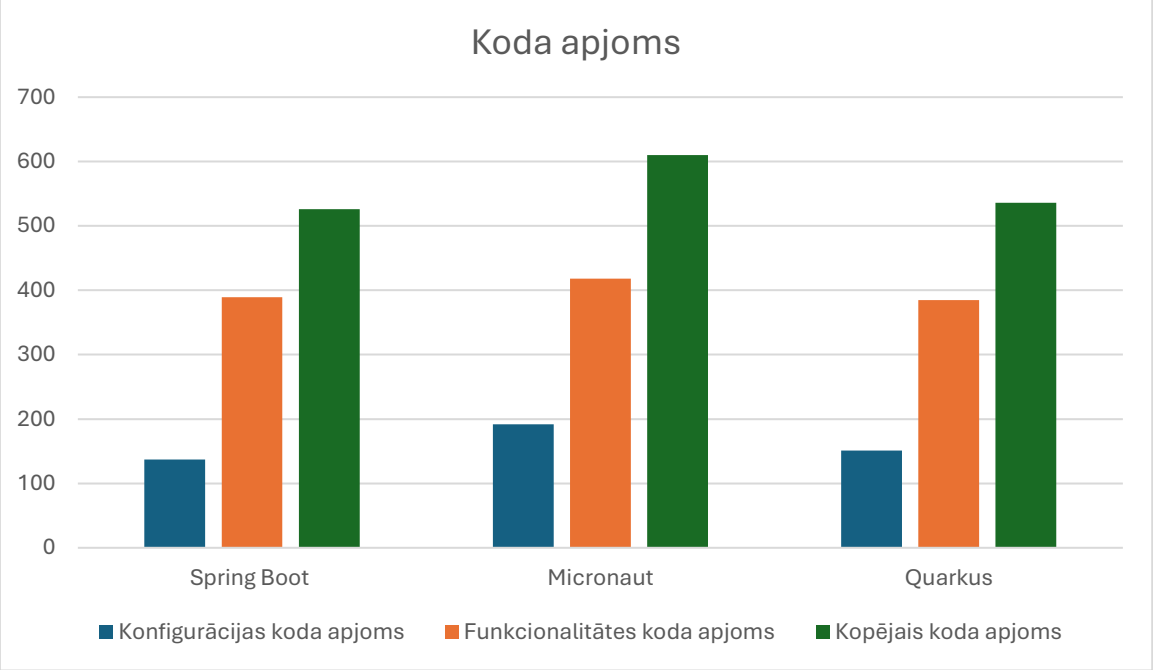
5.13. tabula

	Spring Boot	Micronaut	Quarkus
Konfigurācijas kods	Kontrolieri – 51 rindiņa, DTO – 4 rindiņas, Modeļi – 15, Repozitoriji – 5 rindiņas, Servisi – 6 rindiņas, SecurityConfig – 18 rindiņas, Startēšanas funkcija – 22 rindiņas, application.properties – 16 rindiņas	Kontrolieri – 60 rindiņa, DTO – 12 rindiņas, Modeļi – 15, Repozitoriji – 5 rindiņas, Servisi – 3 rindiņas, AuthenticationProvider – 17 rindiņas, BCryptPasswordEncoderService – 9 rindiņas, Startēšanas funkcija – 22 rindiņas, application.properties – 8 rindiņas, application.yml – 25 rindiņas HttpMethodConversionFilter - 16	Resursi – 89 rindiņa, DTO – 4 rindiņas, Modeļi – 14, Servisi – 11 rindiņas, application.properties – 33 rindiņas
Funkcionalitātes kods	Kontrolieri – 176 rindiņas, DTO – 21 rindiņa, Modeļi – 59 rindiņas, Repozitoriji – 14 rindiņas, Servisi – 119 rindiņas	Kontrolieri – 201 rindiņas, DTO – 15 rindiņa, Modeļi – 67 rindiņas, Repozitoriji – 15 rindiņas, Servisi – 120 rindiņas,	Resursi – 164 rindiņas, DTO – 24 rindiņa, Modeļi – 75 rindiņas, Servisi – 122 rindiņas

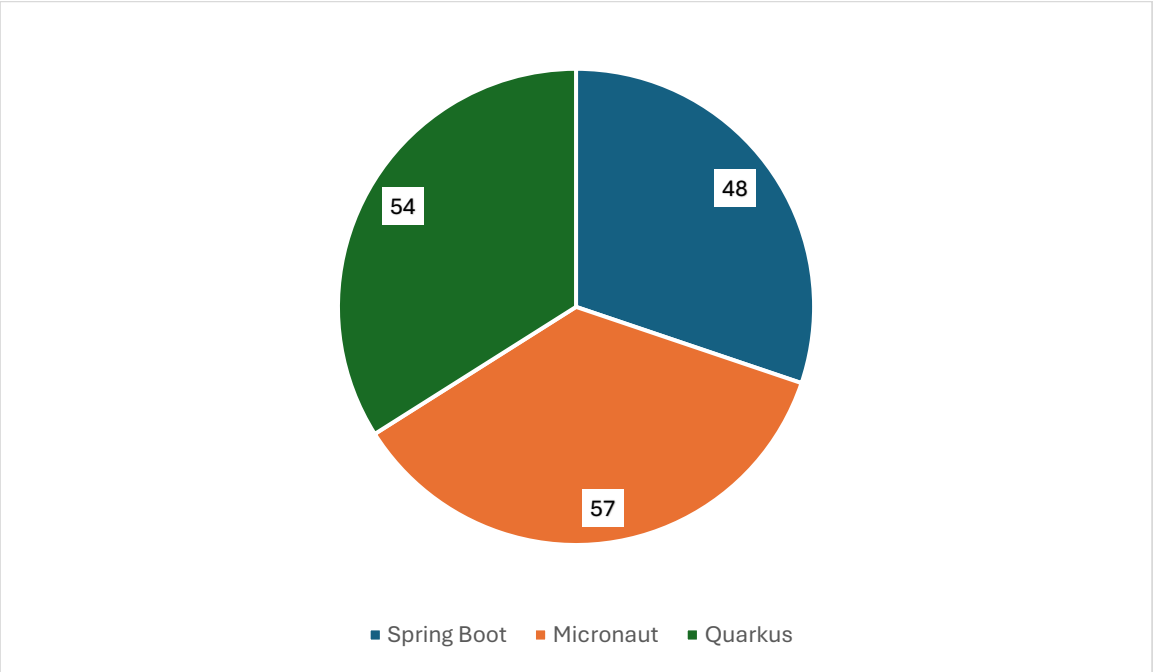
6. REZULTĀTU ANALĪZE

6.1. Izstrāde

Izstrāde ir viena no galvenajām salīdzināšanas metrikām, jo tās ērtums, produktivitāte un optimizācija tieši ietekmē ietvara lietošanas pieredzi. Ar to ir domāts koda apjoms, kopējais izstrādes laiks un koda sadalījums starp dažādu vietnes funkcionalitāti. Kopumā vismazāk laika aizņēma Spring Boot vietnes izstrāde darba autora iepriekšējas pieredzes un visplašākās dokumentācijas dēļ. Pateicoties tā popularitātei, šis ietvars ir ļoti labi integrējams ar dažādām bibliotēkām un piedāvā stabilus, ar laiku pārbaudītus risinājumus. Micronaut izstrāde izradījās grūtāka, jo dažreiz prasīja sarežģītu konfigurāciju. Quarkus vietnes izveide sākumā liekas sarežģīta tā atšķirīgu tehnoloģiju dēļ, tomēr Quarkus piedāvā izstrādātājam draudzīgo pieeju, kas atvieglo un paātrina izstrādi un sarežģītu nianšu saprašanu.



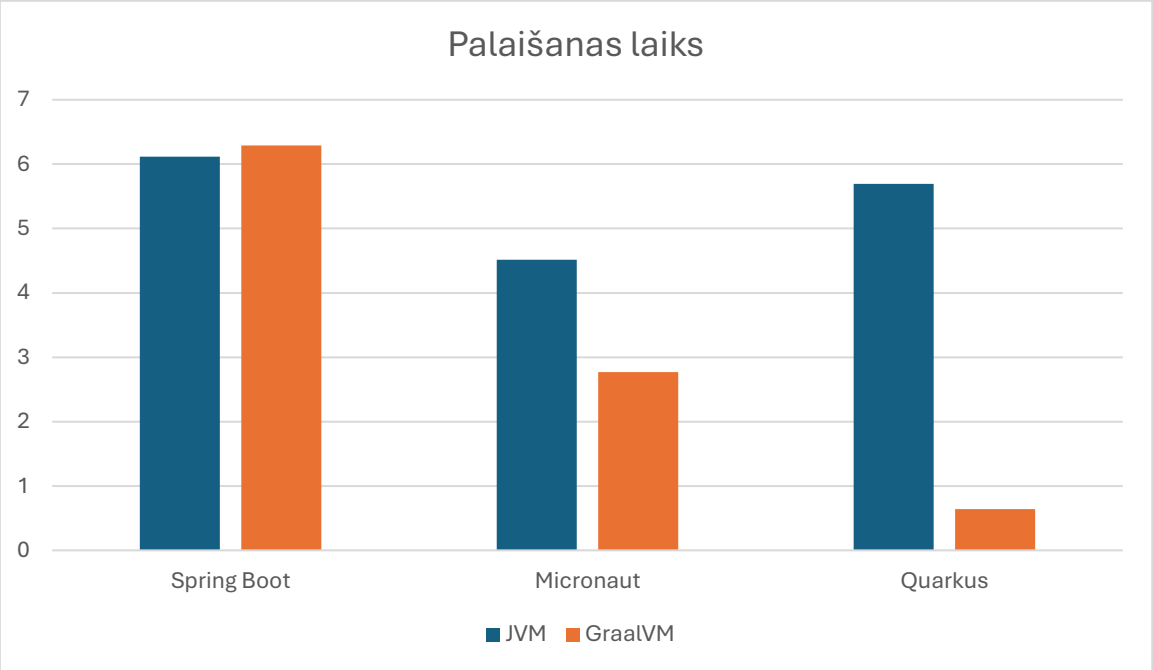
Attēls 6.1. "Koda apjoms katrā no ietvariem"



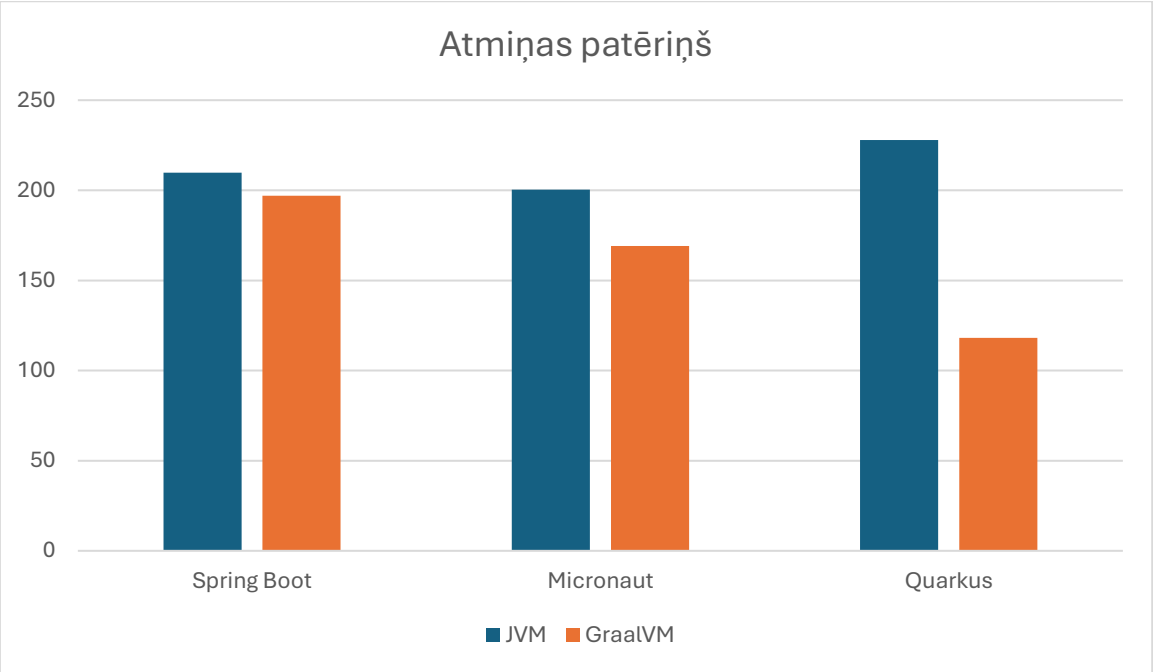
Attēls 6.2. "Izstrādes laiks katram no ietvariem"

6.2. Veiktspēja

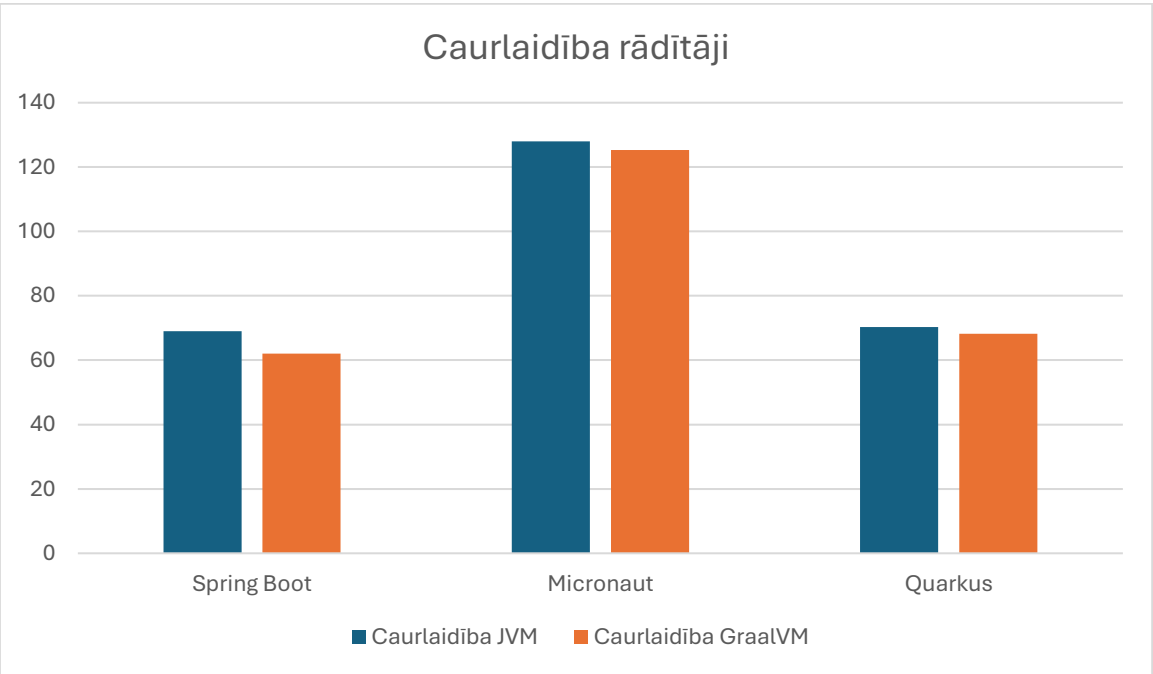
Apkopojot ietvaru veiktspējas radītājus, tika saņemti sekojošie rezultāti:



Attēls 6.3. "Projekta palaišanas laiks katram no ietvariem"



Attēls 6.4. "Atmiņas patēriņš katram no ietvariem"



Attēls 6.5. "Caurlaidība katram no ietvariem"

6.3. Galējie rezultāti

Pamatojoties uz veikto trīs Java ietvaru – Spring Boot, Micronaut un Quarkus – praktisko salīdzinājumu, iespējams sniegt konkrētus, pierādījumos balstītus secinājumus par katra ietvara piemērotību, stiprajām un vājajām pusēm.

6.3.1. Izstrādes ērtums un darbietilpība

- Spring Boot izcēlās ar visērtāko konfigurāciju un plaši dokumentētu ekosistēmu. Projekta struktūra bija skaidra, pateicoties MVC pieejai un piedāvātām bibliotēkām, kas ļāva ātri uzsākt darbu.
- Micronaut izstrāde sākumā likās sarežģītāka un prasīja precīzāku izpratni par kompilācijas laika DI mehānismiem, kā arī iespējamiem autentifikācijas risinājumiem. To sarežģīja arī dokumentācijas trūkums, taču līdzīga modulārā arhitektūra deva iespēju vieglāk sakonfigurēt vietnes galveno funkcionalitāti.
- Quarkus izstrādi uzsākt bija nedaudz sarežģītāk nekā Spring Boot, tā atšķirīgas projekta struktūras dēļ, taču pateicoties tā ērtai un izstrādātājam draudzīgai dokumentācijai, kodēšanas cikls bija ļoti ātrs un intuitīvs. Live Coding funkcija arī nopietni paātrināja izstrādes procesu.

Koda apjoms ietvariem atšķiras ne tik daudz un nevar tikt uzskatīts par noteicošo rādītāju. Nedaudz vairāk koda rindiņu, salīdzinot ar citiem ietvariem, prasīja Micronaut tās minimālistiskas pieejas dēļ, kas lika konfigurēt manuāli funkcijas, kas citos ietvaros nāk jau "no kastes'.

6.3.2. Veiktspēja un resursu patēriņš

- Spring Boot vietnes startēšana notika salīdzinoši lēni un GraalVM natīva kompilācija nerādīja ievērojami labāku rezultātu, salīdzinot ar standarta JVM kompilāciju. Tas arī kopumā aizņēma visvairāk operatīvas atmiņas. Caurlaidība Spring Boot ietvaram sanāca ievērojami zemākā par Micronaut ietvaru.
- Micronaut startēja būtiski ātrāk un izmantoja mazāk RAM, pateicoties savam DI mehānismam. Tas rādīja vēl labākus rezultātus ar GraalVM natīvo kompilāciju. Caurlaidības rādītāji Micronaut ietvaram ir bijuši visaugstākie un pārspēja citus ietvarus gandrīz divās reizēs.
- Quarkus ar GraalVM native image startēja visātrāk un patērēja vismazākus resursus. Tas padara to nepārspējamu bezservera scenārijos, taču ar standartu JVM kompilāciju Quarkus prasīja vairāk RAM pat par Spring Boot ietvaru un palaišanas laiks arī ir bijis samērā lēns, un caurlaidība abu kompilāciju gadījumos ir bijusi tikai nedaudz augstākā nekā Spring Boot ietvaram.

6.3.3. Drošība un autentifikācija

Visi ietvari piedāvāja stabilus drošības risinājumus ar formu bāzētu autentifikāciju un lomu pārvaldību:

- Spring Boot ar Spring Security nodrošināja visdetalizētākās autentifikācijas un autorizācijas iespējas.
- Micronaut risinājums bija vieglāks un efektīvāks, taču prasīja precīzāku konfigurāciju.
- Quarkus integrācija ar Quarkus Security moduli un OIDC paplašinājumiem bija labi piemērota konteineru un mākonī izvietotām sistēmām, kā arī ļāva sakonfigurēt autentifikāciju *application.properties* failā bez papildus klašu izveides.

6.3.4. Dokumentācija un kopienas atbalsts

- Spring Boot pārliecinoši uzvar šajā aspektā – dokumentācija, piemēri, un tēmu klāsts dažādos forumos ir visaptverošs.
- Micronaut dokumentācija ir strukturēta, bet ne tik detalizēta.
- Quarkus dokumentācija ir moderna un labi uzturēta, bet mazāk plaša, salīdzinot ar Spring Boot dokumentāciju.

SECINĀJUMI

Šī bakalaura darba ietvaros tika veikta visaptveroša Java valodā balstītu tīmekļa ietvaru izpēte, apvienojot teorētisko analīzi ar praktisko REST API vietnes izstrādi. Balstoties uz rūpīgu Spring Boot, Micronaut un Quarkus ietvaru salīdzinājumu, var izdarīt vairākus būtiskus secinājumus, kas atklāj katra ietvara piemērotību konkrētām vajadzībām.

Pirmkārt, Spring Boot saglabā savu vadošo pozīciju kā visaptverošs un uzticams ietvars ar plašu ekosistēmu, daudzpusīgu atbalstu un gataviem risinājumiem visiem tipiskākajiem tīmekļa vietņu izstrādes uzdevumiem. Tas ir īpaši piemērots lielāka mēroga, uzņēmuma līmeņa lietotnēm, kur nepieciešama pilnīga infrastruktūra, elastīga konfigurācija un ilgtspējīgs kopienas atbalsts. Tomēr tā refleksijā balstītā arhitektūra un lielā resursu prasība padara to mazāk piemērotu konteinerizētai un bezservera videi, kā arī nelielām mikroservisu lietotnēm ar stingrām prasībām pēc resursu patēriņa.

Savukārt Micronaut demonstrēja būtisku progresu resursu efektivitātes ziņā. Tā compile-time metadatu apstrāde un minimālā refleksijas izmantošana nodrošina ievērojami īsāku startēšanas laiku un mazāku atmiņas patēriņu. Micronaut izrādījās īpaši piemērots mikroservisu arhitektūrai un situācijām, kur nepieciešams ātrs mērogojums vai bezservera izvietošana. Tas piedāvā līdzīgu programmēšanas pieredzi kā Spring Boot, taču ar mūsdienīgāku, veiktspējai draudzīgāku mehānismu.

Quarkus savukārt pārsteidza ar visaugstāko inovāciju līmeni. Tā "Supersonic Subatomic Java" pieeja un pilnīgs GraalVM natīvās kompilācijas atbalsts padarīja to par spēcīgāko kandidātu mūsdienu mākoņvides un konteinerizācijas prasībām. Pateicoties tā zemajam resursu patēriņam un ātrajam startēšanas laikam, Quarkus ir īpaši piemērots Kubernetes un bezservera arhitektūrām. Papildus tam tā Dev UI, Live Coding un Dev Services rīki būtiski uzlabo izstrādātāju produktivitāti. Tomēr natīvās kompilācijas izmantošana prasa rūpīgu sagatavošanu, un tā var nebūt piemērota visām lietošanas vidēm, bet ar standarta JVM kompilāciju netiek realizētas visas šī ietvara priekšrocības.

Praktisku testu rezultāti apstiprināja teorētisko analīzi: Spring Boot demonstrēja izcilu funkcionalitātes pilnīgumu, Micronaut – efektivitāti, ātrumu un augstāko caurlaidību, savukārt Quarkus – nepārspējami zemo startēšanas laiku un atmiņas patēriņu konteineru vidē ar natīvo kompilāciju. Visi trīs ietvari veiksmīgi atbalstīja REST API izstrādi, datubāzu integrāciju un drošības risinājumus. Katram no tiem ir savas priekšrocības un trūkumi, tāpēc izvēle starp tiem būtu balstāma uz konkrētās lietojumprogrammas prasībām.

Noslēgumā jāsecina, ka Spring Boot joprojām ir spēcīgākais izvēles rīks tradicionālajai izstrādei un liela mēroga sistēmām ar kompleksu funkcionalitāti, Micronaut – vispiemērotākais vieglām un ātrām mikroservisu lietotnēm, savukārt Quarkus – līderis modernās, konteinerizētās infrastruktūrās. Šī darba rezultāti ļauj izstrādātājiem pieņemt apzinātu un tehniski pamatotu ietvara izvēli, pamatojoties uz konkrēta projekta prasībām un kontekstu.

IZMANTOTĀ LITERATŪRA UN AVOTI

1. “Spring Boot in Action, 2nd Edition” Autors: Craig Walls
2. JetBrains izstrādātāju aptauja 2023. gadā: https://www.jetbrains.com/lp/devecosystem-2023/java/ (Pēdējo reizi skatīts: 24.05.2025)
3. Micronaut mājaslapa Saite: https://micronaut.io/ (Pēdējo reizi skatīts: 24.05.2025)
4. “Micronaut in Action: Designing, Developing, and Deploying Resilient, Cloud-Native Microservices“ Autors: Aarav Joshi
5. Oficiāla Quarkus dokumentācija. Saite: https://quarkus.io/ (Pēdējo reizi skatīts: 24.05.2025)
6. “Quarkus Cookbook: Kubernetes-Optimized Java Solutions.” Autors: Alex Soto Bueno
7. Oficiāla Spring Boot dokumentācija. Saite: https://docs.spring.io/spring-data/data-commons/docs/1.6.1.RELEASE/reference/html/repositories.html (Pēdējo reizi skatīts: 24.05.2025)
8. Oficiāla Quarkus dokumentācija. Saite: https://quarkus.io/guides/hibernate-orm-panache (Pēdējo reizi skatīts: 24.05.2025)
9. Raksts par REST API arhitektūru. Autors: Howie Mann Saite: https://mannhowie.com/rest-api (Pēdējo reizi skatīts: 24.05.2025)
10. Pētījums par Java būvēšanas rīku popularitāti. Autors: Karsten Silz Saite: https://betterprojectsfaster.com/guide/java-tech-popularity-index-2024-q1/build/ (Pēdējo reizi skatīts: 24.05.2025)
11. Raksts par Java ekosistēmas stāvokli 2024. gadā. Autors: New Relic kompānija Saite: https://newrelic.com/resources/report/2024-state-of-the-java-ecosystem (Pēdējo reizi skatīts: 24.05.2025)
12. Oficiāla Quarkus dokumentācija. Saite: https://quarkus.io/developer-joy/ (Pēdējo reizi skatīts: 24.05.2025)
13. Oficiāla Quarkus dokumentācija. Saite: https://quarkus.io/guides/security-authentication-mechanisms (Pēdējo reizi skatīts: 24.05.2025)
14. Spring Boot ietvara izlaidumu lapa Github vietnē. Saite: https://github.com/spring-projects/spring-boot/releases (Pēdējo reizi skatīts: 24.05.2025)
15. Oficiāla Play dokumentācija. Saite: https://www.playframework.com/documentation/3.0.x (Pēdējo reizi skatīts: 24.05.2025)
16. Raksts par Micronaut ietvaru Baeldung vietnē. Autors: Michael Pratt Saite: https://www.baeldung.com/micronaut (Pēdējo reizi skatīts: 24.05.2025)
17. Vikipēdijas raksts par Quarkus ietvaru. Saite: https://en.wikipedia.org/wiki/Quarkus (Pēdējo reizi skatīts: 24.05.2025)
18. Vikipēdijas raksts par Vert.x ietvaru. Saite: https://en.wikipedia.org/wiki/Vert.x (Pēdējo reizi skatīts: 24.05.2025)
19. Dropwizard mājaslapa. Saite: https://www.dropwizard.io/en/stable/getting-started.html (Pēdējo reizi skatīts: 24.05.2025)

20. Oficiāla Spring Boot dokumentācija. Saite:https://docs.spring.io/spring-boot/reference/using/auto-configuration.html#using.auto-configuration.replacing (Pēdējo reizi skatīts: 24.05.2025)
21. Raksts Spring Boot mājaslapā. Autors: Josh Long Saite: https://spring.io/blog/2014/03/07/deploying-spring-boot-applications (Pēdējo reizi skatīts: 24.05.2025)
22. Oficiāla Spring Boot dokumentācija. Saite: https://spring.io/projects/spring-data-r2dbc (Pēdējo reizi skatīts: 24.05.2025)
23. Oficiāla Spring Boot dokumentācija. Saite: https://spring.io/projects/spring-cloud (Pēdējo reizi skatīts: 24.05.2025)
24. Oficiāla Spring Boot dokumentācija. Saite: https://spring.io/guides/gs/testing-web (Pēdējo reizi skatīts: 24.05.2025)
25. Raksts par Micronaut IOC. Autors: Robert Pudlik. Saite: https://softwaremill.com/introduction-to-micronaut-ioc-basics/ (Pēdējo reizi skatīts: 24.05.2025)
26. Raksts par Micronaut reaktīvo aplikāciju veidošanu. Autors: Alexander Obregon Saite: https://medium.com/@AlexanderObregon/building-reactive-applications-with-micronaut-and-rxjava-4778103b32f8 (Pēdējo reizi skatīts: 24.05.2025)
27. Raksts par Cloud-Native Java programmu izstrādi ar Micronaut ietvaru. Autors: Graeme Rocher Saite: https://www.infoq.com/articles/native-java-micronaut/ (Pēdējo reizi skatīts: 24.05.2025)
28. Apmācības raksts par Micronaut vietnes uzraudzību. https://ruuben.medium.com/monitoring-micronaut-applications-with-prometheus-and-grafana-657f75207f51 (Pēdējo reizi skatīts: 24.05.2025)
29. Oficiāla Quarkus dokumentācija. Saite:https://quarkus.io/guides/building-nativeimage (Pēdējo reizi skatīts: 24.05.2025)
30. Oficiāla Quarkus dokumentācija. Saite: https://quarkus.io/guides/rest (Pēdējo reizi skatīts: 24.05.2025)
31. Oficiāla Quarkus dokumentācija. Saite: https://quarkus.io/developer-joy/ (Pēdējo reizi skatīts: 24.05.2025)
32. Oficiāla Quarkus dokumentācija. Saite: https://quarkus.io/guides/smallrye-fault-tolerance (Pēdējo reizi skatīts: 24.05.2025)
33. Oficiāla Quarkus dokumentācija. Saite: https://quarkus.io/guides/getting-started-testing (Pēdējo reizi skatīts: 24.05.2025)
34. Oficiāla Quarkus dokumentācija. Saite: https://quarkus.io/guides/observability (Pēdējo reizi skatīts: 24.05.2025)
35. Raksts par natīvu kompilāciju Spring Boot ietvaram Baeldung vietnē. Autors: Ralf Ueberfuhr. Saite: https://www.baeldung.com/spring-native-intro (Pēdējo reizi skatīts: 24.05.2025)
36. Apmācības raksts par Spring Boot DI geeksforgeeks forumā. Autors: “deepakanduri”. Saite: https://www.geeksforgeeks.org/spring-dependency-injection-with-example (Pēdējo reizi skatīts: 24.05.2025)
37. Oficiāla Micronaut dokumentācija. Saite: https://guides.micronaut.io/latest/micronaut-security-basicauth-gradle-kotlin.html (Pēdējo reizi skatīts: 24.05.2025)
38. Oficiāla Spring Boot dokumentācija. Saite: https://spring.io/projects/spring-security#overview (Pēdējo reizi skatīts: 24.05.2025)

PIELIKUMI

Šajā sadaļā ir pievienotas vairākas Java klases: kontrolieri, resursi, modeli utt., kā arī HTML failu konfigurācijas.

1. pielikums. Spring Boot ietvara produktu kontrolieris

```
@Controller
public class ProductController {

    private final ProductService productService;
    private final CategoryService categoryService;

    @Autowired
    public ProductController(ProductService productService, CategoryService categoryService) {
        this.productService = productService;
        this.categoryService = categoryService;
    }

    @GetMapping("/category/{categoryId}")
    public String getProductsByCategory(@PathVariable Long categoryId, Model model) {
        model.addAttribute("products", productService.getProductsByCategory(categoryId));
        model.addAttribute("category", categoryService.getCategoryById(categoryId));
        model.addAttribute("product", new ProductDto());
        return "products";
    }

    @GetMapping("/products/edit/{id}")
    @PreAuthorize("hasRole('ADMIN')")
    public String showUpdateForm(@PathVariable Long id, Model model) {
        ProductDto product = productService.getProductById(id);
        model.addAttribute("product", product);
        model.addAttribute("products",
productService.getProductsByCategory(product.getCategoryId()));
        model.addAttribute("category", categoryService.getCategoryById(product.getCategoryId()));
        return "products";
    }

    @PostMapping("/products")
    @PreAuthorize("hasRole('ADMIN')")
    public String createProduct(@ModelAttribute ProductDto productDto, RedirectAttributes
redirectAttributes) {
        try {
            productService.createProduct(productDto);
            return "redirect:/category/" + productDto.getCategoryId();
        } catch (Exception e) {
            return "redirect:/category/" + productDto.getCategoryId();
        }
    }
    @PutMapping("/products/{id}")
    @PreAuthorize("hasRole('ADMIN')")
```

```
    public String updateProduct(
            @PathVariable Long id,
            @ModelAttribute ProductDto productDto,
            RedirectAttributes redirectAttributes) {
        try {
            productDto.setId(id);
            productService.updateProduct(id, productDto);
            return "redirect:/category/" + productDto.getCategoryId();
        } catch (Exception e) {
            return "redirect:/category/" + productDto.getCategoryId();
        }
    }

    @DeleteMapping("/products/{id}")
    @PreAuthorize("hasRole('ADMIN')")
    public String deleteProduct(@PathVariable Long id, RedirectAttributes redirectAttributes) {
        try {
            ProductDto product = productService.getProductById(id);
            Long categoryId = product.getCategoryId();
            productService.deleteProduct(id);
            return "redirect:/category/" + categoryId;
        } catch (Exception e) {
            return "redirect:/";
        }
    }

    @GetMapping("/category/{categoryId}/search")
    public String searchProductsByCategory(@PathVariable Long categoryId, @RequestParam String
query, Model model) {
        model.addAttribute("products", productService.searchProducts(query));
        model.addAttribute("product", new ProductDto());
        model.addAttribute("searchQuery", query);
        model.addAttribute("category", categoryService.getCategoryById(categoryId));
        return "products";
    }
}
```

2. pielikums. Micronaut ietvara produktu kontrolieris

```
@Controller
public class ProductController {

    private final ProductService productService;
    private final CategoryService categoryService;
    private final SecurityService securityService;

    @Inject
    public ProductController(ProductService productService, CategoryService categoryService,
SecurityService securityService) {
        this.productService = productService;
        this.categoryService = categoryService;
        this.securityService = securityService;
    }
```

```
@Get(uri = "/category/{categoryId}", produces = MediaType.TEXT_HTML)
@View("products")
@PermitAll
public Map<String, Object> getProductsByCategory(@PathVariable Long categoryId) {
    Map<String, Object> model = new HashMap<>();
    model.put("products", productService.getProductsByCategory(categoryId));
    model.put("category", categoryService.getCategoryById(categoryId));
    model.put("product", new ProductDto());

    Authentication authentication = securityService.getAuthentication().orElse(null);
    if (authentication != null) {
        model.put("authenticated", true);
        model.put("username", authentication.getName());
        Collection<String> rolesCollection = authentication.getRoles();
        String userRole = null;
        if (rolesCollection != null && !rolesCollection.isEmpty()) {
            userRole = rolesCollection.iterator().next();
        }
        model.put("userRole", userRole);
    } else {
        model.put("authenticated", false);
        model.put("username", null);
        model.put("userRole", null);        }
    return model;
}

@Get(uri = "/products/edit/{id}", produces = MediaType.TEXT_HTML)
@View("products")
@Secured("ADMIN")
public Map<String, Object> showUpdateForm(@PathVariable Long id) {
    ProductDto product = productService.getProductById(id);
    Map<String, Object> model = new HashMap<>();
    model.put("product", product);
    model.put("products", productService.getAllProducts());
    model.put("category", categoryService.getCategoryById(product.getCategoryId()));
    return model;
}

@Post(uri = "/products", consumes = MediaType.APPLICATION_FORM_URLENCODED)
@Secured("ADMIN")
public HttpResponse<?> createProduct(@Body ProductDto productDto) {
    try {
        productService.createProduct(productDto);
        return HttpResponse.redirect(URI.create("/category/" + productDto.getCategoryId()));
    } catch (Exception e) {
        e.printStackTrace();
        return HttpResponse.redirect(URI.create("/category/" + productDto.getCategoryId()));
    }
}
```

```
    @Put(uri = "/products/{id}", consumes = MediaType.APPLICATION_FORM_URLENCODED)
    @Secured("ADMIN")
    public HttpResponse<?> updateProduct(@PathVariable Long id, @Body ProductDto productDto) {
        try {

            productDto.setId(id);
            productService.updateProduct(id, productDto);

            return HttpResponse.redirect(URI.create("/category/" + productDto.getCategoryId()));
        } catch (Exception e) {
            e.printStackTrace();
            Long categoryId = productDto.getCategoryId();
            if (categoryId == null) {
                try {
                    categoryId = productService.getProductById(id).getCategoryId();
                } catch (Exception ex) {
                    ex.printStackTrace();
                    return HttpResponse.redirect(URI.create("/"));
                }
            }
            return HttpResponse.redirect(URI.create("/category/" + categoryId));
        }
    }

    @Delete(uri = "/products/{id}")
    @Secured("ADMIN")
    public HttpResponse<?> deleteProduct(@PathVariable Long id) {
        try {
            ProductDto product = productService.getProductById(id);
            Long categoryId = product.getCategoryId();
            productService.deleteProduct(id);
            return HttpResponse.redirect(URI.create("/category/" + categoryId));
        } catch (Exception e) {
            e.printStackTrace();
            return HttpResponse.redirect(URI.create("/"));
        }
    }



    @Get(uri = "/category/{categoryId}/search", produces = MediaType.TEXT_HTML)
    @View("products")
    @PermitAll
    public Map<String, Object> searchProductsByCategory(@PathVariable Long categoryId,
@QueryValue(defaultValue = "") String query) {
        Map<String, Object> model = new HashMap<>();

        model.put("category", categoryService.getCategoryById(categoryId));
        model.put("products", productService.searchProductsByCategory(categoryId, query));
        model.put("product", new ProductDto());

        if (query != null && !query.trim().isEmpty()) {
            model.put("searchQuery", query);
```

```
        }

        Authentication authentication = securityService.getAuthentication().orElse(null);
        if (authentication != null) {
            model.put("authenticated", true);
            model.put("username", authentication.getName());
            Collection<String> rolesCollection = authentication.getRoles();
            String userRole = null;
            if (rolesCollection != null && !rolesCollection.isEmpty()) {
                userRole = rolesCollection.iterator().next();
            }
            model.put("userRole", userRole);
        } else {
            model.put("authenticated", false);
            model.put("username", null);
            model.put("userRole", null);
        }
        return model;
    }
}
```

3. pielikums. Quarkus ietvara produktu kontrolieris

```
@Path("/")
public class ProductResource {

    private final ProductService productService;
    private final CategoryService categoryService;
    private final Template products;

    @Inject
    public ProductResource(ProductService productService, CategoryService categoryService,
Template products) {
        this.productService = productService;
        this.categoryService = categoryService;
        this.products = products;
    }

    @Inject
    SecurityIdentity securityIdentity;

    @GET
    @Path("/category/{categoryId}")
    @Produces(MediaType.TEXT_HTML)
    public TemplateInstance getProductsByCategory(@PathParam("categoryId") Long categoryId) {
        return products
                .data("products", productService.getProductsByCategory(categoryId))
                .data("category", categoryService.getCategoryById(categoryId))
                .data("product", new ProductDto())
                .data("identity", new IdentityInfo(securityIdentity))
                .data("searchQuery", null);
    }
    @GET
```

```
    @Path("/products/edit/{id}")
    @Produces(MediaType.TEXT_HTML)
    @RolesAllowed("ADMIN")
    public TemplateInstance showUpdateForm(@PathParam("id") Long id) {
        ProductDto product = productService.getProductById(id);

        var category = categoryService.getCategoryById(product.getCategoryId());
        return products
                .data("products", productService.getProductsByCategory(product.getCategoryId()))
                .data("category", category)
                .data("product", product)
                .data("identity", new IdentityInfo(securityIdentity))
                .data("searchQuery", null);

    }

    @POST
    @Path("/products")
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
    @Transactional
    @RolesAllowed("ADMIN")
    public Response createProduct(@BeanParam ProductDto productDto) {
        try {
            productService.createProduct(productDto);
            return Response.seeOther(URI.create("/category/"+ productDto.getCategoryId())).build();
        } catch (Exception e) {
             return Response.seeOther(URI.create("/")).build();
        }
    }

    @PUT
    @Path("/products/{id}")
    @Consumes(MediaType.APPLICATION_FORM_URLENCODED)
    @RolesAllowed("ADMIN")
    @Transactional
    public Response updateProduct(@PathParam("id") Long id, @BeanParam ProductDto productDto)
{
        try {
            productDto.setId(id);
            productService.updateProduct(id, productDto);
            return Response.seeOther(URI.create("/category/" + productDto.getCategoryId())).build();
        } catch (Exception e) {
            return Response.seeOther(URI.create("/category/" + productDto.getCategoryId())).build();
        }
    }
    @DELETE
    @Path("/products/{id}")
    @RolesAllowed("ADMIN")
    @Transactional
    public Response deleteProduct(@PathParam("id") Long id) {
        try {
            ProductDto product = productService.getProductById(id);
            Long categoryId = product.getCategoryId();
```

```
            productService.deleteProduct(id);
            return Response.seeOther(URI.create("/category/" + categoryId)).build();
        } catch (Exception e) {
                return Response.seeOther(URI.create("/")).build();
        }
    }

    @GET
    @Path("/category/{categoryId}/search")
    @PermitAll
    @Produces(MediaType.TEXT_HTML)
    public TemplateInstance searchProductsByCategory(
            @PathParam("categoryId") Long categoryId,
            @QueryParam("query") String query) {


        var category = categoryService.getCategoryById(categoryId);

        if (query == null || query.trim().isEmpty()) {
            return products
                    .data("products", productService.getProductsByCategory(categoryId))
                    .data("product", new ProductDto())
                    .data("category", category)
                    .data("searchQuery", null)
                    .data("identity", new IdentityInfo(securityIdentity));
        }

        return products
                .data("products", productService.searchProductsByCategory(categoryId, query))
                .data("product", new ProductDto())
                .data("category", category)
                .data("searchQuery", query)
                .data("identity", new IdentityInfo(securityIdentity));
    }
}
```

Bakalaura darbs „Java valodā balstītu tīmekļa ietvaru salīdzinājums” izstrādāts LU Eksakto zinātņu un tehnoloģiju fakultātē Datorikas nodaļā.

Ar savu parakstu (drošs elektroniskais paraksts) apliecinu, ka pētījums veikts patstāvīgi un izmantoti tikai tajā norādītie informācijas avoti

Autors: (drošs elektroniskais paraksts) Daniels Janovskis (stud. apl. Nr. dj21030)

Darba vadītājs: (drošs elektroniskais paraksts) (pasniedzējs, Sergejs Rikačovs)

Recenzents: *asociētais profesors Edgars Celms*

Darbs augšupielādēts LUIS’ā 26.05.2025.