Универзитет у Београду

**Факултет организационих наука**

Изабрана поглавља из информационих система

**Развој веб апликације применом Microfrontend архитектуре**

Ментор: Студент:

Проф. др Слађан Бабарогић Вељко Благојевић 2023/3031

Београд, 2024. године

Садржај

[Увод 3](#_Toc176919760)

[Веб апликације и frontend технологије 9](#_Toc176919761)

[Појам microfrontend архитектуре 10](#_Toc176919762)

[Хоризонтална и вертикална подела 12](#_Toc176919763)

[Рутирање и композиција 13](#_Toc176919764)

[Међусобна комуникација microfrontend апликација 14](#_Toc176919765)

[Приступ backend архитектури 16](#_Toc176919766)

[Аутоматизација, тестирање и процес deployment-а 18](#_Toc176919767)

[Анти-патерни microfrontend архитектуре 20](#_Toc176919768)

[Студијски пример 22](#_Toc176919769)

[Опис проблема 22](#_Toc176919770)

[Структура софтверског система 23](#_Toc176919771)

[Имплементација серверског слоја система 25](#_Toc176919772)

[Јава 25](#_Toc176919773)

[Концепти објектно-оријентисаног програмирања 25](#_Toc176919774)

[Spring Boot радни оквир 25](#_Toc176919775)

[Перзистенција података 26](#_Toc176919776)

[Веб сервиси и заштита 26](#_Toc176919777)

[Архитектура 27](#_Toc176919778)

[Имплементација клијентског слоја система 29](#_Toc176919779)

[Главна апликација обмотач 30](#_Toc176919780)

[Међусобно дељење података и комуникација 33](#_Toc176919781)

[Имплементација microfrontend апликација 35](#_Toc176919782)

[Закључак 37](#_Toc176919783)

# Увод

Брза еволуција веб технологија захтевала је усвајање скалабилнијих и одрживијих архитектуралних образаца. Фокус рада је истраживање развоја веб апликације која користи микрофронтенд архитектуру, модеран приступ који проширује принципе микросервисне архитектуре на фронтенд и презентациони слој веб апликације. Декомпозицијом монолитне фронтенд апликације на мање модуле који се могу независно развијати и применити, микрофронтендови нуде побољшану флексибилност, тимску сарадњу и брже развојне циклусе. Апликација је подељена на различите микрофронтендове, од којих је сваки одговоран за одређену функцију или функционалност, омогућавајући паралелни развој и независна ажурирања.

Налази показују да микрофронтенд архитектура значајно побољшава скалабилност и могућност одржавања великих веб апликација. Такође олакшава боље раздвајање брига и омогућава тимовима да искористе различите технологије. Међутим, приступ такође уводи сложеност у смислу интеграције, тестирања и оптимизације перформанси, што се решава кроз најбоље праксе и стратешка решења.

Ова теза закључује да иако микрофронтенд архитектура представља одређене изазове, њене предности у смислу модуларности, аутономије тима и ефикасности на апликацијама са широким могућностима и хоризонталним скалирањем чине ову архитектуру адекватном.

Информациони системи представљају формалан систем за прикупљање, прераду, чување и дистрибуцију и приказ информација. [1]

Један информациони систем, као такав, треба да задовољи већу количину циљева и да буде развијен у условима ограничених финансијских, људских и временских ресурса. [2] Самим тим је од изузетне важности и одабир и приступ методе развоја, поделе рада и одговорности, као и радни ток развоја једног таквог производа. На све ове аспекте може да утиче и одлука софтверске архитектуре, чиме се доказује да та одлука не представља само имплементациони детаљ који утиче само на инжењерски тим, већ на много већи обим организације и одговорног лица који развија и одржава информациони систем кроз сам радни ток, поделу одговорности и висок ниво пословних промена и организационих планова.

Појмови софтверске архитектуре и информационих система иду руку уз руку и уско су повезани. У области информационих система, архитектура софтвера игра кључну улогу у одређивању успеха и ефективности ових система. Информациони систем је организован систем за прикупљање, организацију, складиштење и комуникацију информација. [3] Архитектура софтвера, с друге стране, пружа структурирани оквир који осигурава да је систем усклађен са функционалним и нефункционалним захтевима, омогућавајући систему да испуни своју предвиђену сврху. [4]

1. Структура комплекност: Информациони системи су инхерентно сложени, јер треба да управљају, обрађују и дистрибуирају огромне количине података међу више заинтересованих страна. Архитектура софтвера помаже у управљању овом сложеношћу тако што обезбеђује нацрт који води развој и интеграцију различитих компоненти система. Осигурава да је сваки део информационог система, као што су базе података, кориснички интерфејси и логика обраде, организовани на начин који побољшава кохерентност и смањује редундантност.
2. Скалабилност и флексибилност: Како информациони системи расту у обиму и размери, основна софтверска архитектура мора да прихвати овај раст без угрожавања перформанси. Добро дизајнирана софтверска архитектура омогућава да се информациони системи скалирају додавањем нових модула, интеграцијом са другим системима и надоградњом постојећих функционалности. Такође пружа флексибилност прилагођавања променљивим пословним потребама, технолошком напретку и регулаторним захтевима.
3. Усклађеност са пословним циљевима: Информациони системи су дизајнирани да подрже специфичне пословне циљеве, било да се ради о побољшању оперативне ефикасности, побољшању доношења одлука или олакшавању комуникације. Архитектура софтвера игра кључну улогу у усклађивању техничких аспеката система са овим пословним циљевима. Дефинисањем јасних архитектонских образаца и принципа дизајна, архитектура софтвера осигурава да је информациони систем изграђен на начин који директно подржава стратешке циљеве организације.
4. Осигуравање интегритета система: Интегритет и поузданост информационог система су најважнији, јер директно утичу на тачност и поузданост информација које систем обрађује. Архитектура софтвера успоставља смернице за безбедност, руковање грешкама и интегритет података, обезбеђујући да информациони систем може да издржи потенцијалне претње и да одржава конзистентан рад чак и у неповољним условима.
5. Побољшање одрживости: Информациони системи су дугорочна улагања која захтевају стално одржавање и ажурирања. Робусна софтверска архитектура олакшава лакше одржавање промовишући модуларност, где су компоненте система лабаво повезане и могу се независно ажурирати или заменити. Ова модуларност смањује сложеност одржавања система, минимизирајући застоје и осигуравајући сталну доступност.
6. Подршка интероперабилности: У данашњем међусобно повезаном свету, информациони системи често морају да комуницирају са другим системима, било у оквиру исте организације или између различитих ентитета. Архитектура софтвера обезбеђује интероперабилност усвајањем стандардизованих комуникационих протокола, формата података и интеграционих оквира. Ово омогућава беспрекорну размену података и сарадњу између различитих система, повећавајући укупну вредност информационог система.

Дизајн и имплементација фронтенд архитектуре су интегралне компоненте укупне софтверске архитектуре. [5] Како софтверски системи постају све сложенији, улога фронтенд архитектуре постаје све значајнија у обезбеђивању да су кориснички интерфејс (*UI*) и корисничко искуство (*UX*) у складу са ширим архитектонским циљевима система. Овај одељак истражује замршен однос између фронтенд архитектуре и укупне софтверске архитектуре, наглашавајући како прва доприноси кохезији, перформансама и могућности одржавања софтверског система у целини.

1. Интеграција и кохезија: Фронтенд архитектура служи као мост између корисника и основних функционалности система, енкапсулираних унутар бацкенд архитектуре. Да би софтверски систем функционисао кохезивно, фронтенд мора неприметно да се интегрише са позадинским сервисима, базама података и апликативним интерфејсом (*Application Programming Interface - API*). Софтверска архитектура обезбеђује основну структуру, која може подржавати микросервисе, RESTful API, GraphQL или gRPC са којима фронтенд комуницира. Придржавајући се принципа и образаца успостављених целокупном софтверском архитектуром, фронтенд обезбеђује несметан проток података између корисничког интерфејса и апликативног слоја, што доводи до кохерентног корисничког искуства.
2. Конзистентност у дизајну и обрасцима: Конзистентност у архитектури софтвера је од суштинског значаја за одржавање квалитета и поузданости система. Фронтенд архитектура доприноси овој конзистентности имплементацијом образаца и софтверских патерна, који су усклађени са глобалном архитектонском стратегијом. Ови обрасци не само да организују фронтенд кодну базу на начин којим се може управљати, већ такође обезбеђују да интеракција између фронтенда и бекенда прати предвидљив и стандардизован протокол, смањујући ризик од настанака грешака или било ког типа неусклађености.
3. Оптимизација перформанси: Перформансе софтверског система се често перципирају кроз одзив и брзину фронтенда. Архитектура фронтенда игра кључну улогу у оптимизацији перформанси управљањем начином на који се подаци преузимају, обрађују и приказују на страни клијента. Технике као што су лењо учитавање (*lazy loading*), кеширање (енгл. *caching*) на страни клијента и ефикасно управљање стањем се користе у оквиру архитектуре фронтенд-а да би се минимизирало време учитавања и смањио оптерећење апликативног слоја. Целокупна архитектура софтвера мора да прихвати ове стратегије, често кроз оптимизоване *API* приступне тачке, балансирање оптерећења (енгл. *load balacing*) и *Content Delivery Network - CDN*, како би се осигурало да се побољшања перформанси направљена на нивоу фронтенд-а у потпуности реализују у целом систему.
4. Скалабилност и проширивост: Како се софтверски системи развијају, они морају бити способни за процес скалирања како би задовољили све компликованије захтеве корисника и прилагођавали се новим захтевима клијената. Фронтенд архитектура доприноси скалабилности и проширивости целокупног система усвајањем модуларних и вишекратних компоненти, које се могу лако ажурирати или заменити по потреби. Ова модуларност се огледа у софтверској архитектури кроз праксе као што су сервисно оријентисана архитектура (енгл. *Service oriented architecture - SOA*). Усклађивање између фронтенд и бекенд модуларности омогућава софтверском систему да се ефикасно скалира, било додавањем нових функција, подржавањем већег броја корисника или интеграцијом са додатним системима.
5. Одржавање: Могућност одржавања софтверског система је уско повезана са архитектонским одлукама донетим и на презентационом и на апликативном нивоу. Лоше структурирана архитектура фронтенда може довести до проблема са кодом који временом постаје све тежи за одржавање, промену и рефакторисање, чинећи будућа ажурирања изазовнијим и скупљим. Пратећи најбоље праксе у организацији кода, документацији и тестирању, архитектура фронтенда може значајно смањити овај изазов и побољшати дугорочну одрживост система. Ова могућност одржавања додатно је подржана целокупном софтверском архитектуром, која би требало да обезбеди чврсту основу која олакшава континуирану интеграцију, аутоматизовано тестирање и контролу верзија на фронтенд и бекенд компонентама.
6. Безбедност и интегритет података: Безбедност је најважнија брига у софтверској архитектури, а фронтенд је често прва линија одбране од потенцијалних претњи. Фронтенд архитектура мора да имплементира мере безбедности као што су валидација уноса, аутентификација и ауторизација у координацији са сигурносним протоколима логичког дела програма. Целокупна софтверска архитектура обезбеђује неопходне безбедносне оквире и смернице којих се фронтенд мора придржавати, обезбеђујући да се одржава интегритет података и да је систем заштићен од рањивости као што су *Cross-site scripting - XSS* или *Cross-site request forgery - CSRF*. Овај координисани приступ безбедности осигурава да је цео софтверски систем отпоран на нападе.

Како софтверски системи расту у сложености и обиму, архитектонски обрасци су еволуирали како би се суочили са изазовима развоја, примене и одржавања. Једна таква еволуција је микрофронтенд архитектура, која проширује принципе микросервиса на фронтенд домен. [6] Овај приступ има значајне импликације не само на фронтенд архитектуру већ и на целокупну софтверску архитектуру и, на крају, на ефикасност и прилагодљивост информационих система у целини. Овај одељак истражује везе између микрофронтенд архитектуре, традиционалне монолитне архитектуре фронтенда и њиховог заједничког утицаја на информационе системе.

Микрофронтенд архитектура представља промену парадигме од монолитног фронтенд дизајна ка више модуларном и децентрализованом приступу. Традиционална фронтенд архитектура обично организује кориснички интерфејс (*UI*) као јединствену, кохезивну кодну базу, која је у интеракцији са бекенд слојом преко дефинисаних *API* тачака или сервиса. Насупрот томе, микрофронтенд архитектура разлаже фронтенд на мање, независне јединице које се могу развијати, постављати и одржавати одвојено.  
Ова модуларизација омогућава већу флексибилност и паралелни развој, јер различити тимови могу да раде на различитим карактеристикама фронтенда или компонентама без међусобног мешања. [6] Сваки микрофронтенд може да има сопствени одабир технологија са којима ради (енгл. *tech stack*), *deployment pipeline*-e и уникатан приступ верзионисања кода, омогућавајући агилније и итеративне процесе развоја. Веза између микрофронтенд архитектуре и традиционалне фронтенд архитектуре лежи у начину на који оба служе слоју корисничког интерфејса, али микрофронтендовима нуде скалабилнији и управљивији приступ у раду са великим, сложеним апликацијама које могу да се разграниче на одвојене домене.

Усвајање микрофронтенд архитектуре има дубоке импликације на целокупну архитектуру софтвера. Баш као што микросервис разлаже бекенд на мање, независне услуге, микрофронтендови примењују исти принцип на фронтенд. Ово усклађивање између фронтенд и бекенд архитектуре олакшава кохерентнији и конзистентнији дизајн система, где су оба слоја модуларна и способна да се развијају независно.

Целокупна софтверска архитектура такође мора да подржава примену и оркестрацију микрофронтендова, често путем контејнеризације (нпр. *Docker*) и алата за оркестрацију (нпр. *Kubernetes*), обезбеђујући да систем остане кохезиван упркос својој децентрализованој структури.

Импликације микрофронтенд архитектуре се протежу изван техничког подручја, утичући на дизајн, примену и управљање информационим системима у целини. Информациони системи су дизајнирани да подрже специфичне пословне процесе и циљеве, захтевајући од њих да буду флексибилни, скалабилни и да реагују на промене. Микрофронтенд архитектура доприноси овим циљевима омогућавајући детаљнију контролу над корисничким интерфејсом, омогућавајући да се различити делови информационог система ажурирају или побољшају без ометања целог система.

На пример, информациони систем е-трговине може да користи микрофронтендове за управљање различитим аспектима корисничког искуства, као што су каталог производа, корпа за куповину и процес плаћања. Свака од ових карактеристика може се развијати и одржавати независно, омогућавајући систему да се брзо прилагоди захтевима тржишта, интегрише нове технологије или одговори на повратне информације корисника. Ова агилност је кључна за одржавање релевантности и конкурентности информационог система. [7]

Штавише, раздвајање фронтенд и бекенд архитектуре кроз микрофронтендове и микросервисе омогућава ефикасније скалирање информационих система. Како број корисника или трансакција расте, систем се може хоризонтално скалирати постављањем додатних инстанци специфичних микрофронтенда или микросервиса, уместо скалирања целе апликације. Ова циљана скалабилност побољшава перформансе и ефикасност система, осигуравајући да може да поднесе повећано оптерећење без жртвовања корисничког искуства.

# Веб апликације и frontend технологије

Веб апликације, некада познате као Rich Internet Applications - RIA, значајно су еволуирале. Данас, Software as a Service – SSAS платформе нуде сложене корисничке интерфејсе које омогућавају различите функционалности, од штампања визиткарти до управљања банковних рачунима. [7] Када се упуштају у нови пројекат, програмери морају пажљиво да размотре архитектонски избор и да разумеју изазове који су пред нама.

Једностраничне апликације (Single Page Application - SPA) су широко прихваћене од стране компанија. Састоје се од JavaScript датотека које обухватају цео фронтенд, преузете унапред. Једностраничне апликације комуницирају са API приступним тачкама, избегавајући вишеструка повратна путовања ка серверу и побољшавајући корисничко искуство. Поред тога, једностраничне апликације управљају рутирањем на страни клијента и омогућавају флексибилну логичку дистрибуцију између сервера и клијента. [8]

Током протеклих неколико деценија, појавиле су се дебате о томе да ли су танки или дебели клијенти супериорна решења. Оба приступа имају своје предности и недостатке, у зависности од специфичног контекста апликације. За мулти-платформске апликације, танак клијент и дебели сервер могу бити од предности, омогућавајући да се функција која је једном имплементирана искористи на више места. Насупрот томе, за десктоп апликације чија је једна од главних лимитација сиромашна или никаква веза са остатком мреже, дебели клијент и танак сервер могу бити пожељнији. Међутим, једностраничне апликације показују одређене недостатке. Почетно време учитавања обично је дуже због преузимања целе апликације унапред. [9] Поред тога, једностраничне апликације представљају изазове за оптимизацију претраживача (Search Engine Optimization - SEO), захтевајући алтернативне методе индексирања.

# Појам microfrontend архитектуре

Фронтенд екосистем је еволуирао на велики број приступа и методологија које стоје иза њих, омогућавајући решавање конкретних проблема. Међутим, проблем великих апликација где се посао може поделити на скроз независне тимове је било изузетно тешко. Микрофронтенд је архитектура која помаже у скалирању управо таквих апликација.

Мицрофронтенд је архитектура у повоју, јасно инспирисана архитектуром микросервиса. Главна идеја иза ње је да се "монолит" разбије на мање делове, омогућавајући организацији да расподели количину посла на више аутономних тимова без потребе за успоравањем или смањивањем њиховог проток испоруке.

Поред организационих промена, овај приступ развоју софтвера дозвољава и да различити тимови развијају своје апликације коришћењем потпуно других оквира. Један тим може да користи React оквир, док други сасвим комотно може да имплементира апликацију применом VueJS оквира. Иако је ово валидно и итекако могуће, приступ мешања оквира се смтра анти-патерном и није препоручљив на новим green-field пројектима. Овај и остали анти-патерни биће обрађени при крају тренутног поглавља.

У традиционалном развоју и имплементацији софтверских система, често се користе апликације са централизованом логиком. Овакви монолитни системи обезбеђују једноставнију имплементацију, али захтевају да сви модули буду усаглашени. [10] Такав тип архитектуре се назива монолит. С друге стране, микросервисна архитектура омогућава раздвајање функционалности на мање, независне модуле. Сем Њуман детаљније анализа трансформацију монолита у системе са микросервисном архитектуром, као и све проблеме и изазове који следе одатле. [11] Један од проблема може бити раздвајање базе података, при чему је потребно извршити адекватна дистрибуција. Овим долази и до проблема конзистентности података и далеко комплексинијих образаца и примена патерна трансакција као што су Сага патерн или Two Phase Commit и Three Phase Commit стратегије.

При приступању имплементацији микросервиса или трансформацији монолита у микросервисе потребно је обратити пажњу на следеће принципе: [7]

• Моделовање око пословних домена

• Изолација грешака

• Висока уочљивост

• Могућност независног deployment-а

• Децентрализација свега

• Сакривање имплементационих детаља

• Култура аутомације

Сви ови принципи важе и на примену клијентских апликација развијеним коришћењем микрофронтенд архитектуре.

За даље разумевање, потребно је и разјаснити и појам „главне“ апликације (Master application shell) која представља обмотач око осталих апликација. Специфична је по томе што практично не садржи пословну логику, и њен главни циљ је енкапсулација осталих апликација.

Главна апликације игра кључну улогу у овој архитектури. А неки од њених најбитнијих карактеристика је то што је одговорна за дистрибуцију различитих делова апликације, пружа заједнички простор за учитавање микрофронтендова. Обично, главна апликација укључује траку заглавља и подножја (header и footer), управља аутентификацијом и дељеним процесима. [7]

Визуелно се одгледа само кроз хедер и футер и самим тим је, са аспекта корисничког интерфејса, изузетно једноставна. Још једна битна карактеристика је навигација и рутирање, што ће детаљније бити објашњено у наставку рада.

# Хоризонтална и вертикална подела

Унутар главне апликације постоје више микрофронтенд апликација које могу бити приказане на два начина: помоћу хоризонталне или вертикалне поделе.

Хоризонтална подела представља стил у којем се на истом погледу/екрану приказују више различитих микрофронтенд апликација. Овим приступом је омогућено да више програмерских тимова утичу на исти екран изменама својих респективних апликација. Овиме је и додата флексибилност, узимајући у обзир да се један микрофронтенд може појавити и на више екрана. [12]

Вертикална подела је подела у којој корисник на екрану може да види само један микрофронтенд. Овим је омогућено да један поглед има само једну апликацију, што је практично при подели на пословне домене (унутар Domain Driven Development - DDD).

# Рутирање и композиција

Следећа подела се врши на приступе при композицији микрофронтенд апликација. Постоје три начина, а они су: [6]

* *Client-side composition*
  + Ово је приступ где се композиција апликација врши на самом клијентском рачунару.
  + Главна апликација динамички учитава микрофронтендове користећи JavaScript или HTML улазне тачке, омогућавајући апликацији да дода *Document Object Model* чворове или иницијализује *JavaScript* компоненте.
* *Edge-side composition*
  + Композиција се врши на нивоу *Content Delivery Network - CDN*
* *Server-side composition*
  + Композиција се врши на изворном серверу

Поред одлуке на ком од ова три нивоа ће се вршити композиција, такође је потребно и одабрати и слој на коме ће се сместити логика рутирања. Уколико је употребљена композиција на страни сервера, тада је неопходно користити и рутирање на истом том слоју. Употребом *edge-side* композиције, рутирање се врши помоћу путање прегледача тј. *Unified Resource Locator – URL.* Финална опција је рутирање на клијентском нивоу где ће логика рутирања бити смештена у главној апликацији, и ово је пожељни приступ који допушта највећи ниво флексибилности и могућности проширивања у изузетно комплексним условима. [7]

# Међусобна комуникација microfrontend апликација

Још један од битних одабира архитекте пре почетка пројектовања фронтенд архитектуре апликације је начин на који начин ће микрофронтенд апликације међусобно да комуницирају. Битно је нагласити да је међусобна комуникација анти-патерн тј. образац који представља лошу праксу. Цела концепција микрофронтенд архитектуре је да апликације међусобно не би требало да буду свесне постојања других и да су потпуно изоловане, енкапсулирајући свој пословни домен. Међутим, у реалном свету, ипак постоје ситуације где су такви типови комуникација ипак неопходни.

Први образац који се примењује у оваквој комуникацији је да различите апликације не смеју да имају директну везу, тј. зависност са другим апликацијама. Ово се омогућава тако што се забрањују референце пројеката међусобно. Овај приступ је неопходан како би се одржао принцип независног *deployment*-а.

Један уобичајени приступ укључује коришћење заједничког глобалног стања или *EventBus*, где микрофронтенд апликације могу да објаве и претплате се на догађаје, омогућавајући им да реагују на промене стања или акције које су покренули други микрофронтендови. Ова метода промовише принцип слабог везивања (енгл. *loosely coupled*), пошто микрофронтендови не морају директно да се упућују једни на друге, чиме се чува њихова аутономија. [7]

Друга стратегија је коришћење прилагођених догађаја или механизама за преношење порука, где микрофронтендови емитују догађаје које други микрофронтендови могу да слушају и реагују на одговарајући начин. Овај приступ омогућава циљанију комуникацију, смањујући вероватноћу непотребних промена стања и побољшавајући перформансе.

Ово су били приступи који су практично ексклузивно коришћени при пројектовању хоризонталне поделе. А поред њих могуће је и користити напредније и модерније технологије и принципе као што су *Reactive Streams* и сигнали. [7]

Следеће стратегије међусобне комуникације се могу применити на обе, и хоризонталну и вертикалну поделу. Оне су на нивоу самог интернет прегледача, па се могу глобално користити, невезано од имплементационих одлука и архитектуре.

Једна од њих је примена *query* параметара из саме путање. Ово је решење које се обично користи на једноставнијим примерима и мањим структурама података. Класичан пример би био идентификатор одређеног пословног ентитета, чији детаљи могу да се учитају коришћењем додатног сервиса. Овај приступ није препоручљив са осетљивије податке као што су кориснички креденцијали или шифре.

Приступање локалној меморији интернет прегледача је још један приступ, где је могуће сачувати било какав објекат у меморију сесије или *localStorage,* па дате податке читати унутар било ког дела кода из било које апликације. [13]

Колачићи (енгл. *Cookies*) су мало датотеке и могу бити чуване током једне сесије, због чега такође могу бити коришћени за перзистенцију података које више различитих микрофронтенд апликација може да чита и процесуира.

Коначно, микрофронтендови могу да комуницирају путем дељених сервиса или API-ја, посебно у случајевима када подаци морају да се синхронизују или радње треба да се координирају на више микрофронтендова. Апстраховањем заједничке функционалности у слој дељеног сервиса, микрофронтендови могу међусобно да комуницирају на контролисан и доследан начин, обезбеђујући да се промене у једном микрофронтенду на одговарајући начин одразе на друге.

Сваки од ових комуникационих механизама има свој скуп компромиса, а избор методе треба да буде вођен специфичним архитектонским захтевима и ограничењима апликације. Ефикасна комуникација између микрофронтендова је неопходна за постизање кохезивног корисничког искуства уз очување модуларности и скалабилности које нуди архитектура микрофронтенда.

# Приступ backend архитектури

Поред пажње око имплементационих детаља самог фронтенда, потребно је и фокус дати на бекенд. Иако концептуално, ови слојеви треба да буду независни, развијајући се одвојено, ипак је корисно применити одређене добре праксе на једном слоју у зависности од имплементације другог. Узимајући обзир у то, бекенд слој апликације која користи микрофронтенд архитектуру може да изгледа на један од следећих ограничених приступа.

Први тип комуникације са бекендом је коришћењем „речник сервиса“ (енгл. Service Dictionary).

Ово је приступ који може да се подједнако добро примени и на монолит као и на модуларни монолит. Овај речник није ништа више него статички исписана листа доступних бекенд сервиса које фронтенд може да позива. Овај приступ има више алтернатива, прва би била да се статички направи конфигурациона датотека где је излистана свака приступна тачка доступних сервиса. Овај алтернатива може да се имплементира коришћењем JavaScript Object Notation - JSON датотеке. Други приступ је излагање још једне API приступне тачке која ће вратити листу доступних сервиса. Овај приступ захтева да први, почетни микрофронтенд у вертикалној подели направи тај захтев и самим тиме иницијализује дату листу, а у случају хоризонталне поделе, одговорност позивања тог сервиса и постављање његове повратне вредности преузима главна омотач апликација. Речник сервиса, поред саме листе доступних сервиса може и да поседује друге мета-податке, као што је одређена расподела листа, по одређеном кључу, који може бити пословни домен, ниво приступа применом рола и пермисија. [7]

Други тип комуникације микрофронтенд апликације и слоја пословне логике је API Gateway патерн. Образац који је већ изузетно познат у микросервисној архитектури. Наиме, такозвана капија представља пролазну тачку свих захтева од стране клијента према бекенд слоју. Она апстрахује микросервисну архитектуру излагајући само један сервис који је задужен за примање свих захтева, и његово даље рутирање на адекватан сервис. Ово је управо примена Reverse Proxy софтверског патерна. [14] API Gateway је, поред излагања једног сервиса за комуникација и рутирање, одговоран и за ауторизацију, лимитирање захтева од стране клијената, кеширање, телеметрију, прикупљање метрика и друге централизовање функције. Још једна предност коришћења обрасца капије је тиме што је апстрахована комуникација унутар микросервисне архитектуре. Клијент може да комуницара са капијом путем HTTP RESTful, а унутар микросервисне архитектуре може да се користи скроз другачији протокол или пак извршити транзицију између више протокола а да клијент уопште не буде свестан тога. Google Remote Procedure Call – gRPC је оквир за комуникацију између сервиса који је изузетно популаран између микросервиса јер омогућава висок ниво перформанси, ефикасну бидирекционалну комуникацију и проток информација. [15] Ове предности су омогућене уз помоћ HTTP 2 протокола, за разлику од REST приступа који користи HTTP 1.1. Нов протокол доноси и одређени вид комплексности који може да представља препреку при комуникацији са интернет прегледачем, због тога се не користи за директну комуникацију са фронтендом. Још једна мана капије је једно место грешке, где, уколико тај сервис није доступан, цео слој пословне логике није доступан клијентима. Овај недостатак се надомешћује прављењем реплика унутар Kubernetes кластера, где при падању и отказу једног, други заузима његово место и преузима све будуће захтеве клијената. Како је патерн API капије већ добро устаљен у индустрији, на тржишту постоји велики број решења и имплементационих библиотека које далеко упрошћавају рад са овим видом комуникације клијента и микросервисне архитектуре у позадини. Како је капија посредник између свих захтева, додатним кораком се додаје кратко време проласка кроз сервис капије, што у апликацијама високих перформанси где се да приметити разлика од пар милисекунди латенције, капија може представљати проблем. [7]

Софистициранији приступ обрасца капије је њена имплементација више капија које се деле по пословним доменима, па је овим уведен додатан слој гранулације и контроле. Партиционисање захтева на овај начин може бити корисно где је потребна додатна енкапсулација одређене пословне логике. Овај приступ, у одређеној мери, смањује проблем најслабије карике, тј. једне тачке падања целог система.

Трећи патерн бекенд слоја и комуникације са фронтендом је такозвани Backend for Frontend – BFF. BFF образац је специфичан приступ комуникације између презентационог слоја и слоја бизнис логике, где се прави нови сервис, налик API Gateway-у, али са пар кључних разлика. Једна од ситуација где се користи BFF наспрам капије је у мулти-платформским окружењима. Мобилна верзија апликације, због величине екрана или брзини мреже и процесора, приказује драстично мању количину података, или потпуно други скуп информација. Једноставна капија не може да прави овакву дистинкцију, и потребно је додатно процесуирање и филтрирање података. [16] Још једна примена је и код апликација где се користе извештаји и контролне табле са великом количином информација агрегираних са випе извора. Саму консолидацију и груписање тих података који долазе са више различитих сервиса може да изврши BFF. BFF је такође могуће разложити на више инстанци које гранулирано покривају своје пословне домене.

GraphQL је упитни језик креиран од стране Facebook-а 2015. године. Главна карактеристика је овог манипулативног језика је што клијент може да изабере тачно она поља и атрибуте које жели да прочита или промени. [17] Самим тим се драстично смањује количина терета коју мрежа мора да пренесе између клијента и сервера, као и само оптерећење над базом података. GraphQL је популарна опција међу апликацијама које се приказују на више платформи, због своје могућности прилагођавања траженом количином података, као и у real-time апликацијама које се ажурирају у реалном времену. GraphQL чини још један вид могуће комуникације између фронтенд и бекенд програма у информационом систему.

# Аутоматизација, тестирање и процес deployment-а

Велики аспект микрофронтенд архитектуре се огледа и у значају робусне стратегије аутоматизације. Док микросервиси нуде флексибилност и скалабилност, оне такође уносе сложеност у управљање инфраструктуром, захтевајући снажну континуирану интеграцију (Continuous Integration - CI) и континуирану испоруку (Continuous Delivery - CD). Кључни принципи за ефикасну аутоматизацију укључују одржавање брзих повратних информација, редовно понављање стратегија аутоматизације, оснаживање развојних тимова и успостављање јасних заштитних баријера. Чврста стратегија тестирања, посебно за потпуно тестирање од краја до краја (енгл. End to End Testing), такође је истакнута као кључна за обезбеђивање поуздане примене и минимизирање проблема током испоруке софтвера. Једна од одлука при имплементацији CI/CD грана је одабир верзионисања кода. Микрофронтенд омогућава два приступа. [7] Први је такозвани моно-репо, где постоји један репозиторијум и у њему се налазе сви микрофронтенд пројекти. Овим је могуће лакше централизовати или апстраховати заједничке функционалности и издвојити их на једноставнији начин. Овај приступ је препоручљив и за транзицију са монолитне архитектуре на микрофронтенд, а такође је и једноставнији почетницима или новозапосленима на пројекту. Други приступ верзионисању кода је помоћу поли-репо стратегије, где сваки микрофронтенд има свој посебан, одвојени репозиторијум. Поли-репо стратегија омогућава већу флексибилност при грањању и мање застоја између тимова, али уводи нови вид комплексности и обавеза програмера да дуплирају код и више воде рачуна о конвенцијама именовања. [7]

Поред верзионисања кода, још једна одлука са којом се сусрећу софтверске архитекте и DevOps тим инжењера је начин на који ће микрофронтенд апликација бити испоручена тј. која Deployment стратегија ће бити употребљена.

• Плаво-зелено испоручивање (енгл. Blue Green Deployment): Стратегија испоручивања где се окружење дели на две идентичне секције: плаву и зелену. Стратегија омогућава да се различите микрофронтенд апликације, креиране од стране различитих тимова, испоруке независно једни од других. [18] Стратегија функционише кроз наредне кораке: [19]

1. Апликација опслужује клијенте на плавој половини окружења. Рутер је посредник који све захтеве од стране клијента шаље ка плавој инстанци. Зелена инстанца стоји чекајући.

2. Када је потребно испоручити нову верзију апликације, ажурирање се врши на зелену инстанцу, која тренутно не опслужује захтеве са продукције.

3. Инжењерски тим надгледа да ли је ажурирање на зелену инстанцу протекло уреду, проверавајући конфигурације, телеметрију и остале сигурносне захтеве

4. Када је осигурано стабилно зелено окружење са новом верзијом, тада се врши промена боје, где зелена постаје активна инстанца која опслужује саобраћај са продукције, а плава инстанца остаје на чекању

5. Процес се понавља при сваком ажурирању

• Испоручивање стратегијом канаринца (енгл. Canary Release Strategy): Слично као претходна стратегија, и ова стратегија садржи рутер који служи као посредник чија је улога да навигира захтеве ка адекватној инстанци. Круцијална разлика је у томе што не шаље захтеве коришћењем бинарног алгоритма, где сви захтеви иду искључиво на плаву или на зелену инстанцу, већ један проценат шаље на једну, док други проценат на другу. Стратегија је изузетно популарна при бета тестирању где одређени, обично мањи, број корисника користи нову верзију апликације, чиме инжењерски тим може да провери стабилност ажурираног кода пре него што га испоручи свим клијентима [19]

• Strangler Fig патерн: Образац који је настао намењен процесу трансформације монолитне архитектуре слоја пословне логике у архитектуру микросервиса. Образац који такође може да се примени и на презентациони слој. Образац je архитектонска стратегија у софтверском инжењерству, која олакшава инкременталну миграцију старих система на модерне архитектуре. Овај образац, који је концептуализовао Мартин Фаулер, потиче своје име по биљци смокве давитеља, која обавија и на крају замењује своје стабло домаћина. [20] У пракси, образац укључује постепену замену специфичних функционалности постојећег система новим сервисима. Овај постепени приступ ублажава ризике повезане са заменама целог система одједном омогућавајући коегзистирање и застарелих и нових системима током прелазног периода. [21] Како се нове функционалности развијају и интегришу, застарели систем се прогресивно „дави“ све док се не може у потпуности повући из употребе. [20][21] Овај метод је посебно повољан за сложене системе где је фазна миграција неопходна да би се обезбедио континуитет и минимално ремећење.

# Анти-патерни microfrontend архитектуре

Појам анти-патерна је појам у инжењерским гранама који показује често коришћено решење за понављајуће проблеме, међутим то решење на дуге стазе може бити контра-продуктивно.

Први анти-патерн који се среће код примене архитектуре микрофронтенда је неприкладна идентификација микрофронтенда и настаје када се једна компонента погрешно третира као микрофронтенд. Ова неправилна класификација подрива основне принципе микрофронтенда, који наглашавају независни развој, примену и скалабилност. Компоненти, по дефиницији, недостаје аутономија и енкапсулација потребне да би била прави микрофронтенд, што доводи до повећане сложености и изазова одржавања. Сходно томе, ова пракса може резултирати уско повезаним системима који негирају предности модуларности и независне примене. [22] Јасан сигнал да је компонента погрешно идентификована као микрофронтенд се може видети уколико се при хоризонталној подели, на једном екрану приказује чак десет или више микрофронтенд апликација. [7]

Следећи анти-патерн би био примена вишеструких библиотека и оквира. [23] Иако је могуће да сваки микрофронтенд буде написан у различитој библиотеци или оквирима као што су React, Angular или VueJS, ово је пак изузетно неприкладно. Драстично се повећава комплексност кодне базе, као и што се перформансе смањују јер је потребно изузетно већа количина JavaScript кода да се преузме при почетном учитавању апликације. Наравно, уколико се прави нови пројекат, препоручљиво је користити један оквир, међутим, велики број старих пројеката врше транзицију ка микрофронтенд архитектури, где се не може избећи мешање различитих оквира.

Анти-патерн који је тешко препознати је креирање микрофронтенда који унутар себе има превише или премало одговорности. Искусан инжењер би требало да препозна на ком нивоу гранулације се врши расподела логике између више пројеката.

Честа лоша пракса је коришћење бидирекционог тока података. [7] Овакав вид комуникације је изузетно непрепоручљив унутар микрофронтенд архитектуре, а као његова замена се препоручује унидирекциони ток података, на којем је заснована библиотека као што је *Redux*.

Вишеструко позивање истог сервиса од стране випе различитих микрофронтенд апликација може учинити код изузетно неефикасним. При доласку до овакве ситуације потребно је проверити да ли је извршена квалитетна гранулација између расподеле пројеката, колико је учестао позив, између колико апликација. Једно од решења је обмотавање више микрофронтенд апликација једном компонентом обмотача која ће бити задужена за комуницирање са бекенд сервисом, чиме би се двоструко смањио саобраћај према слоју пословне логике.

Постоји већи број имплементационих библиотека које промовишу и засновани су на микрофронтенд архитектури.

Прва са великим нивоом популарности је *Module Federation*. Уведен у Webpack верзији 5, представља значајан напредак у области микрофронтенд архитектуре. Ова иновативна библиотека олакшава динамичко дељење кода и зависности у више JavaScript апликација, омогућавајући на тај начин децентрализован приступ развоју апликација. Допуштајући да се независни модули развијају, тестирају и постављају аутономно, Module Federation побољшава скалабилност и могућност одржавања. [24] Штавише, подржава беспрекорну интеграцију различитих оквира и верзија, чиме се промовише флексибилност и смањује ризик од сукоба верзија. Овај модуларни приступ не само да оптимизује коришћење ресурса већ и побољшава целокупно искуство програмера поједностављујући управљање сложеним апликацијама великих размера.

Друга изузетно популарна у заједници, и распрострањена библиотека је *single-spa*. Single-spa је пионирска библиотека у микрофронтенд екосистему, дизајнирана да олакша оркестрацију више JavaScript оквира унутар једне апликације. Штавише, single-spa подржава екосистем које не зависи од оквира, промовишући флексибилност и смањујући сложеност повезану са интеграцијом различитих технологија.

# Студијски пример

У наставку ће бити приказан практичан пример примене претходно поменуте архитектуре на веб апликацији за преглед пацијената.

## Опис проблема

Информациони систем треба да решава реалан проблем који се среће у функционисању било каквог друштва. А један од главних стубова сваке модерне цивилизације је њено здравство и медицина. Доктор, као круцијални аспект сваког здравственог система је такође, врло често и уско грло функционисања самог система. Због тога би требало имплементирати и прилагодити информациони систем, како би сам рад лекара био што ефикаснији и без непотребних прекидања. Овим високим захтевима се поставља висока лествица што се тиче самих функционалних и нефункционалних захтева у самом информационом систему са којим један лекар барата. Наиме, при прегледању пацијента, доктор пролази кроз више радних токова, а сви они морају бити интуитивни и да подржавају све добре праксе корисничког искуства.

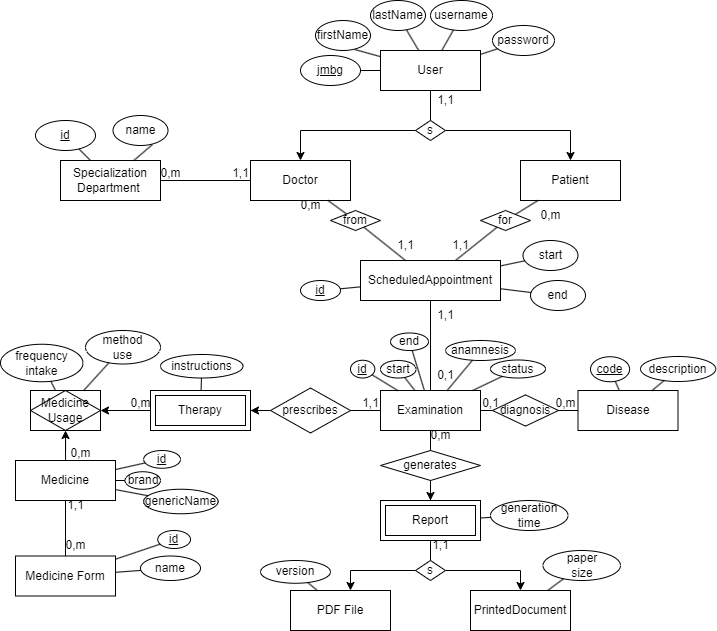
Сам рад доктора се своди кроз следећи ток активности, након што се пријави на информациони систем, доктор може да види списак заказаних термина, на основу тога прозива пацијента и сам преглед почиње. Како пацијент описује своје тегобе, тако лекар уписује анамнезу. Лекар прегледа пацијента, гледа листу болести из међународне листе болести МКБ 10, бира дијагнозу и преписује терапију са лековима и објашњењем њиховог коришћења. По потребни штампа тај документ, да ли у папирној форми, или кроз генерисања PDF датотеке, и на крају по потреби заказује контролу за датог пацијента или гледа термине за остале прегледе.

У овом систему се може видети одређена сегментација функционалности која би се поделила на мање независне сервисе. Ово је подела која може да се изврши и клијентској и/или на серверској страни.

## Структура софтверског система

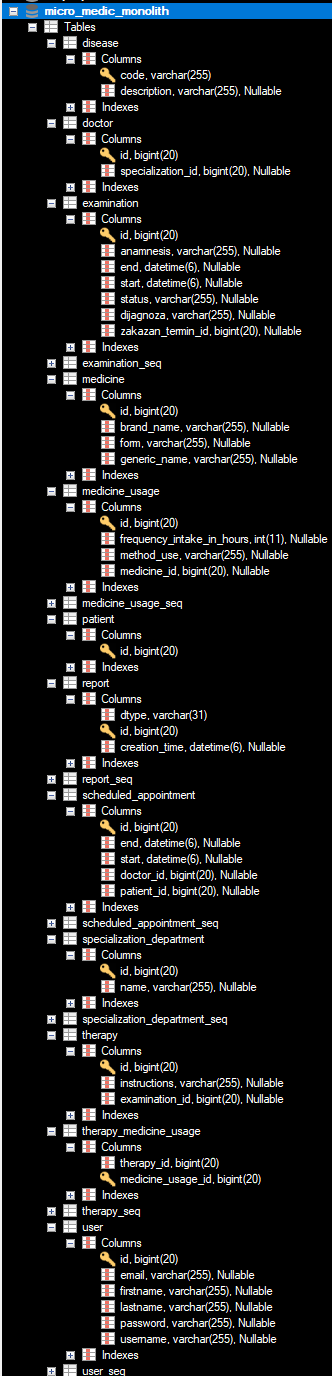
Структура информационог система је сам темељ за његово даље пројектовање и имплементацију. Статички приказ система се може описати путем концептуалног (доменског) модела, као и релационог модела.

На следећој слици је дат проширени модел објекти везе датог студијског примера.



Слика 1 Проширени модел објекти везе система

Један релациони модел се представља путем табела, примарних и спољних кључева, атрибута и типова тих истих атрибута, као и свих ограничења. На следећој слици је дат релациони модел кроз табеле у релационој бази података, са њиховим атрибутима и типовима.



Слика 2 Табеле и атрибути базе података

## Имплементација серверског слоја система

Сам серверски део апликације је имплементиран помоћу Spring Boot радног оквира унутар Java окружења.

### Јава

Јава је технологија која уједно представља и платформу као и објектно-оријентисан језик. Јава је састављена од скупа технологија и алата који се користе за развој програма. Апликација која је написана у Јава програмском језику се може покретати на различитим платформама, што је омогућено уз помоћ Јава Виртуелне Машине (енгл. Java Virtual Machine - JVM).

### Концепти објектно-оријентисаног програмирања

Програми који су написани са објектно-оријентисаном парадигмом користе објекте као основу при пројектовању и развоју самог софтверског решења. Заснива се на различитим техникама, као што су наслеђивање, модуларност, полиморфизам и енкапсулација.

Јава као један од програмских језика који је заснован на објектно-оријентисаној парадигми и користи класе и објекте као своје основне градивне блокове. Класа је шаблон за креирање објеката, обезбеђује почетне вредности за променљиве као и имплементације понашања кроз методе. Објекат је инстанца класе и уобичајени начин његовог креирања је уз помоћ конструктора класе.

### Spring Boot радни оквир

Spring радни оквир (енгл. framework) представља најпопуларнији Јава оквир развијен почетком века који је заснован на концепту инверзије контроле и управљањем контејнерима.

Spring радни оквир има следеће карактеристике:

1. Инверзија контроле (енгл. Inversion of Control - IoC)

2. Аспектно оријентисано програмирање (енгл. Aspect Oriented Programming - AOP) је парадигма којом Spring омогућава програмерима да примене аспекте на свој код како би добили додатну функционалност.

3. Модулирање и слојевита архитектура.

4. Подршку за разне библиотеке, као што су Java Database Connection (JDBC) за приступ базама података, RESTful веб сервисе, Java Message Service за обраду порука, Hibernate за објектно релационо мапирање (енгл. Object Relational Mapping - ORM), и многе друге.

5. Једноставно тестирање.

### Перзистенција података

У развоју софтвера, базе података омогућавају управљање подацима, при чему је релациони модел са SQL-ом најчешће коришћен. Системи за објектно-релационо мапирање (ORM) омогућавају програмерима да раде са подацима као са објектима уместо коришћења SQL-а.

Hibernate је алат за објектно-релационо мапирање у Јава окружењу који аутоматски генерише SQL упите користећи мета податке, омогућавајући програмерима да управљају базом података преко метода унутар алата. Кључне карактеристике укључују мапирање објеката на табеле, управљање везама, трансакцијама, лењо учитавање и кеширање за оптимизацију перформанси.

Spring Data JPA је подскуп Spring Data алата који омогућава стандардизован приступ објектно-релационом мапирању, често користећи Hibernate као имплементацију. Он подржава JPQL за писање упита и нуди функционалности попут сортирања, пагинације и „*Derived Query Methods*“ које омогућавају аутоматско генерисање SQL упита на основу имена метода.

### Веб сервиси и заштита

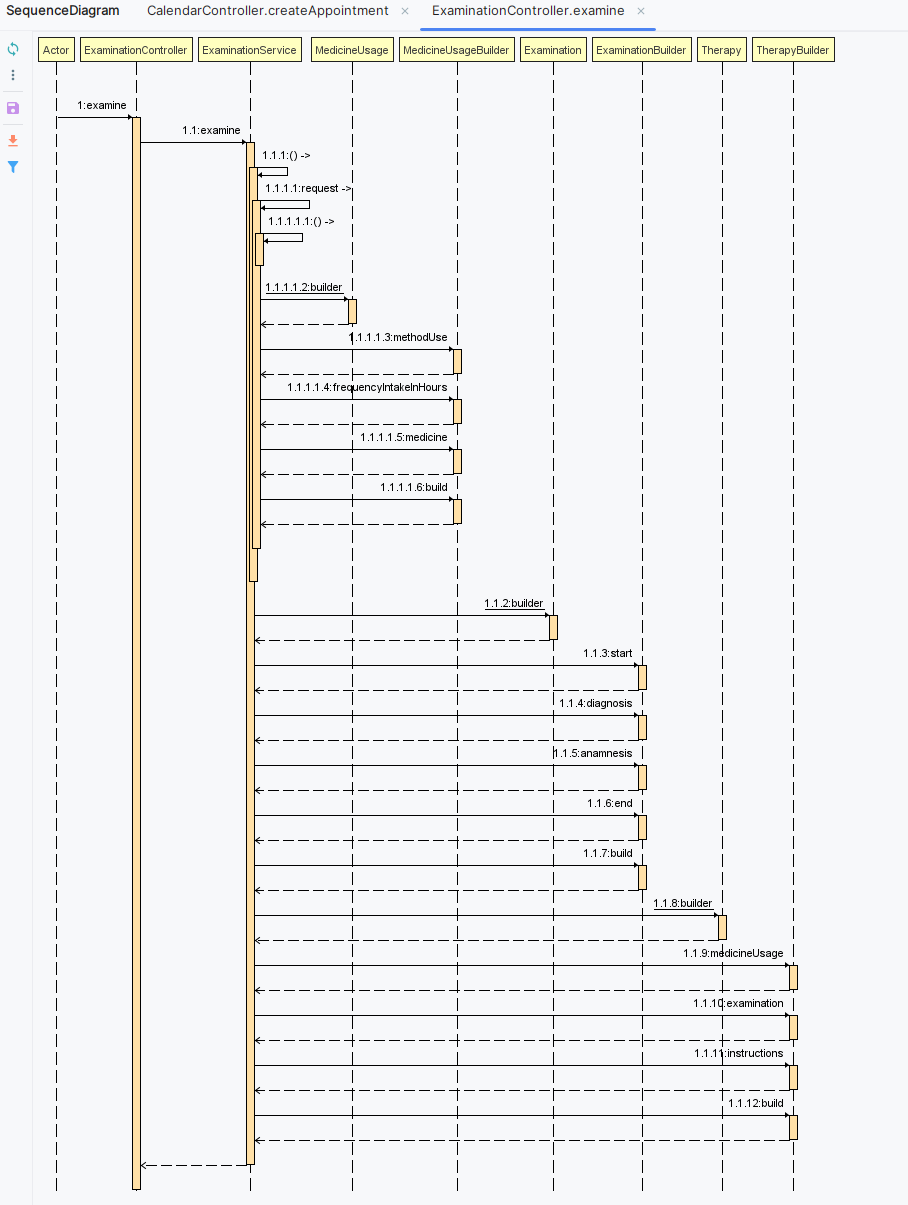
*REST (*енгл. *Representational State Transfer)* је архитектонски стил за пројектовање веб сервиса који користи HTTP за комуникацију и захтева да сервиси буду без стања, при чему се ресурси преносе у XML или JSON формату. Spring Web модул подржава развој RESTful веб сервиса у Јава окружењу користећи Spring MVC, омогућавајући лако дефинисање рута, обраду захтева и генерисање одговора у JSON или XML формату.

Приступ интернету и заштита података су кључни процеси који укључују аутентификацију и ауторизацију корисника, при чему се идентификује корисник и одређује његов ниво овлашћења. JSON Web Token (JWT) је стандард за безбедан пренос података у JSON формату, који омогућава само-одрживу аутентификацију без потребе за сесијама на серверу. Spring Security модул обезбеђује заштиту Јава апликација, омогућавајући интеграцију са JWT за сигурну аутентификацију и ауторизацију корисника у веб апликацијама.

### Архитектура

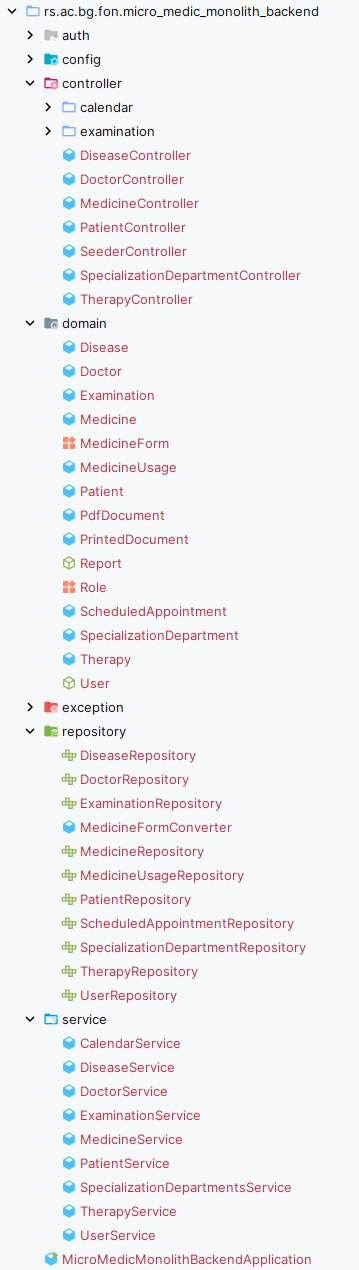
За пројекат је одабрана монолитна архитектура са 3 слоја и једном базом података. Први слој су контролери коју служе да приме захтев и преусмере саобраћај. Други слој су сервиси у којима се одвија пословна логика. Трећи слој је слој репозиторијума који представља слој за приступ бази података.

Наредна слика представља дијаграм секвенци за процес вршења прегледа, узимање анамнезе и давање дијагнозе пацијенту.



Слика 3 Дијаграм секвенци прегледа пацијента

Оваква архитектура је произвела следећу структуру датотека:



Слика 4 Структура датотека серверске апликације

## Имплементација клијентског слоја система

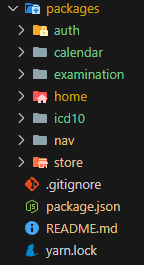
За креирање клијентског дела апликације коришћена је microfrontend архитектура уз помоћ имплемнтационих библиотека ModuleFederation и single-spa.

Прво је потребно разграничити и одредити ниво гранулације саме апликације. Према, претходно споменутим, анти-патернима треба обрадити пажњу на више аспекта. А најбитнији је да апликације буду сегментиране као праве засебне пословне целине, а не компоненте.

Са тим на уму, јасно је могуће издвојити microfrontend апликацију која је заслужена за сам процес слушања пацијента, постављања анамнезе, откривање дијагнозе и препоручивања терапије. Ово све представља једну пословну целину на којој један инжењерски тим може засебно да ради и да је развија.  
Друга апликација која би могла да се издвоји је сам календар, тј. временски дијаграм на коме су приказани заказани термини.  
Трећа microfrontend апликација би могла бити апликација за приказ и претрагу дијагноза.  
И још један издвојен microfrontend би био задужен за аутехнтификацију, јер би садржао странице везане за пријављивање на систем, регистрацију, манипулацију личним профилом и остало.  
На крају остаје и део система који је задужен за управљањем заглавља и подножја, као и главна апликација обмотач која ће све да садржи унутар себе.

За пројекат је изабран приступ верзионисања кода названим моно-репозиторијум, где се сав програмски код ставља у један git репозиторијум. Наравно свака апликација је засебна унутар њега.

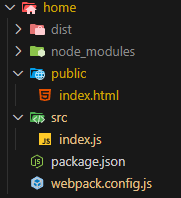
Наспрам претходне анализе, репозиторијум за верзионисање кода ће имати следећу структуру фасцикли:



Слика 5 Структура фасцикли клијентске апликације

### Главна апликација обмотач

Главна апликација која у себи мора да садржи сву логику за обједињавање осталих microfrontend апликација унутар ње, се у овом случају назива ‘home’.



Слика 6 Структура датотека главне апликације

Она је сачињена од неколико битних ставки. Прва је конфигурација WebPack 5, која је дата на наредној слици:



Слика 7 Конфигурација WebPack-а

Унутар дате конфигурације, потребно је обратити пажњу на више ствари. Прва је да се ова апликација покреће на порту 3001. Иако се остале апликације покрећу на другим портовима, при самом конзумирању, крајњем кориснику је битан само овај порт, јер ће кроз њега, прегледач динамички повући остале HTML фрагмете са JavaScript кодом.

Друга битна ставка унутар ове конфигурације је улазна тачка апликације, која представља JavaScript датотеку звану index.js. Наиме, та датотека је задужена за регистрацију свих microfrontend апликација које ће се користити унутар система.

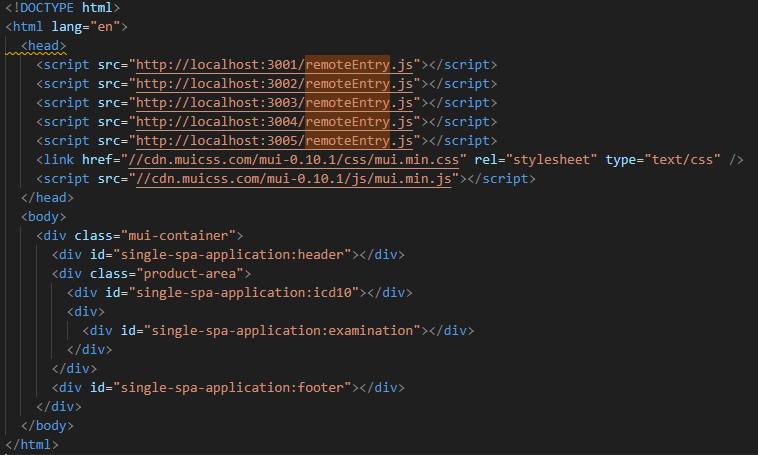


Слика 8 Регистрација апликација

Применом single-spa библиотеке, можемо да динамички, на основу путање (дефинисане адресе) учитамо у меморију стране компоненте које су експортоване као microfrontend апликације.

Location параметар је потребан унутар ове главне апликације обмотач, како би знала да врши рутирање. У овом примеру је коришћен рутирање на страни клијента.

И на крају, само конструисање HTML елемената је извршено унутар index.html датотеке, где је дата структура пројекта.

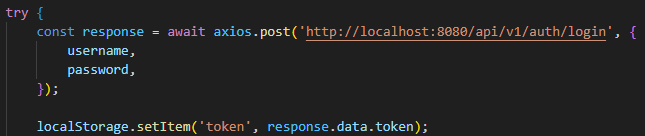


Слика 9 HTML структура целокупног пројекта

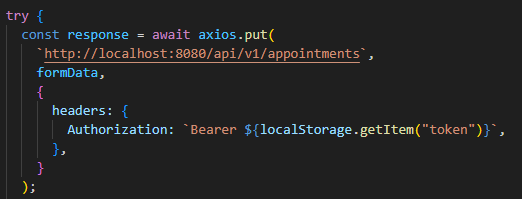
Поред што је дефинисана структура, у *head* тагу су позване скрипте које заправо представљају улазне тачке за све остале microfrontend апликације.

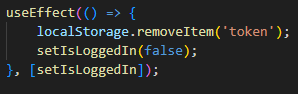
### Међусобно дељење података и комуникација

Следећи изузетно битан пројекат је фасцикла звана *store* која садржи стање читаве апликације. Осврћући се на поменуте приступе при дељењу података између microfrontend апликација, речено је да је лоша пракса да се превелики број података дели, и да је то сигнал да није извршена адекватна сегментација. Међутим, на датом примеру је потребно чувати и користити кроз више апликација исти JWT токен. Овај случај коришћења је сасвим валидан и може се имплементирати на сваки од начина поменутих из претходних поглавља. Али једна од имплементација би била кроз локални меморијски простор прегледача, где би синтакса изгледала овако:



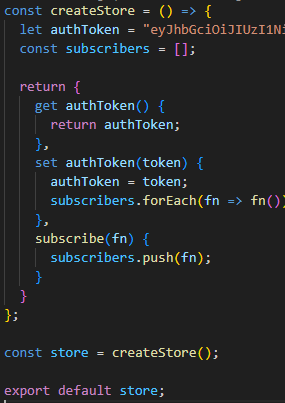
Слика 10 Примена локалне меморије прегледача





Други приступ је коришћење програмске логике за чување стања, применом библиотеке за управљање стањем, као што је Redux или MobX. Како је овде пример изузетно једноставан, а и зарад чистијег и очигледнијег показног примера, искоришћена је самостална имплементација ове функционалности која се налази у засебном пројекту названом *store*.

Ова апликација мора да имати идентичну поставку WebPack конфигурације, као и свака друга, док сама имплементација чувања и мењања токена коришћеног за аутентификацију изгледа овако:

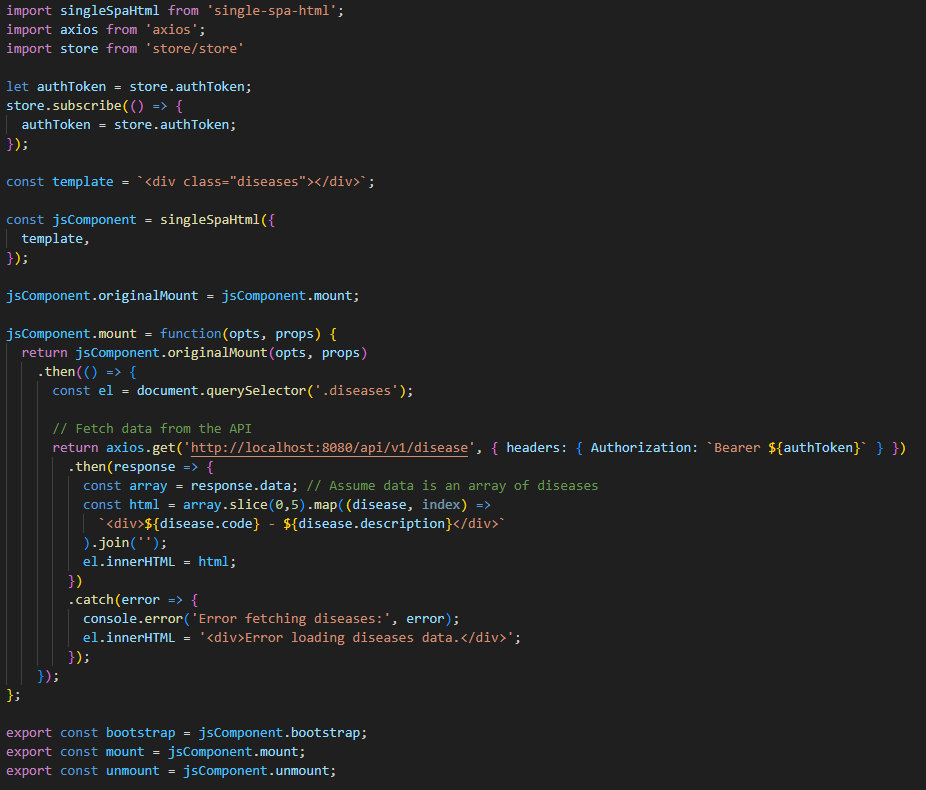


Слика 11 Имплементација store-а

Касније се други microfrontend-ови могу „претплатити“ на store и на тај начин приступити његовим подацима, као и у реалном времену ослушкивати промене које се одвијају у њему.

### Имплементација microfrontend апликација

Апликација која је задужена да прикаже листу болести коришћењем Међународне класификације болести 10 је имплементирана путем „чистог“ HTML и JavaScript кода. Па, поред стандардне конфигурације WebPack-а коју смо претходно објаснили, овако изгледа њена сама иницијализација.



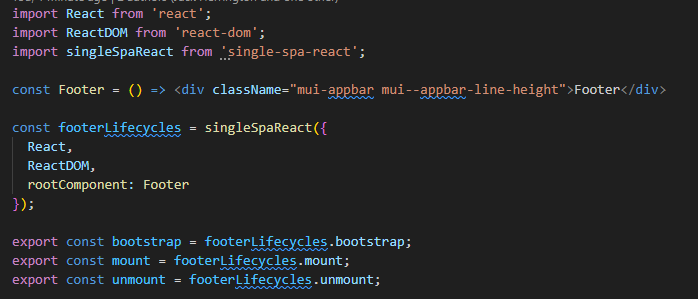
Слика 12 Имплементација HTML microfrontend-а

Како се користи single-spa библиотека, потребно је искористити њену подваријанту која је специјализована за рад са „čistim“ HTML кодом, па је употребљена single-spa-html библиотека.

Она излаже интерфејс који захтева да се приложе три методе, а то су: bootstrap, mount (метода која се извршава при самом постављању компоненте на главну апликацију обмотач), и unmount (метода која се извршава када компонента није више део Document Object Model-а).

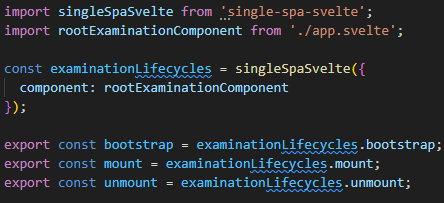
Ово је пракса која се користи и за друге радне оквире.

Како је подножје имплементирано путем React радног оквира, може се видети сличан интерфејс, али друге подбиблиотеке специјализоване за рад са React радним оквиром.



Слика 13 Имплементација React microfrontend-а

Апликација која се користи при вршењу самог прегледа пацијента и писања анамнезе и дијагнозе је направљена у виду форме. Та форма је имплементирана путем *Svelte* технологије и њена иницијализација три „lifecycle“ методе изгледа на начин како је приказано на следећој слици:



Слика 14 Имплементација Svelte microfrontend-а

# Закључак

Microfrontend архитектура представља нову парадигму у развоју веб апликација, која омогућава поделу сложеног корисничког интерфејса на мање, независно развијене и имплементиране компоненте. Овај приступ нуди бројне предности, међу којима су повећана флексибилност у развоју, бржа испорука нових функционалности и боља изолација грешака. Захваљујући својој модуларности, microfrontend архитектура омогућава тимовима да раде независно, уз коришћење различитих технологија и алата, што резултира већом скалабилношћу и одрживошћу кода. Међутим, ова архитектура доноси и изазове, као што су сложенија интеграција компоненти, управљање међузависностима и повећана сложеност у управљању стањем апликације. Упркос тим изазовима, microfrontend приступ се све више усваја у великим и динамичним развојним окружењима због своје способности да олакшава раст и еволуцију апликација у дугорочном периоду. У закључку, microfrontend апликације представљају ефикасно решење за модерне веб апликације које захтевају скалабилност, модуларност и брзу испоруку, али њихова успешна имплементација захтева пажљиво планирање и искуство у дизајну дистрибуираних система.