# Универзитет у Београду

## Математички факултет


Мастер рад

---

# Примена реактивног дистрибуираног програмирања на примеру онлајн банкарских сервиса

---

*Студент*:
Драгутин Илић

*Ментор*:
др Иван Чукић

*Ментор:*

**др Иван Чукић, доцент**

*Математички факултет, Универзитет у Београду*

*Чланови комисије:*

**др Александар Картељ, доцент**

*Математички факултет, Универзитет у Београду*

**Анђелка Зечевић, асистент**

*Математички факултет, Универзитет у Београду*

# Примена реактивног дистрибуираног програмирања на примеру онлајн банкарских сервиса

**Апстракт** – Систем онлајн банкарства *BANKX22* развијен као део овог рада је веб систем отвореног кода који нуди сервисе који се могу наћи у уобичајним финансијским апликацијама попут онлајн плаћања, размене новца из једне валуте у другу, узимања кредита, као и проверавање стања на рачуну и увид у историју трансакција. Систем је имплементиран да задовољи принципе реактивног дистрибуираног система попут добре одзивности и еластичности, поузданости, односно издржљивости, лабаве спрегнутости сервиса и транспарентне комуникације компонената. За имплементацију клијентске стране коришћене су савремене *Javascript* библиотеке *React, Redux* и *ReduxToolkit.* Клијентска страна комуницира са сервером преко развојног окружења *.Net Web API* употребом *REST* протокола. За имплементацију серверске стране искоришћено је развојно окружење *Akka .Net* које имплементира модел система актера, погодног за олакшану имплементацију разних реактивно дистрибурианих принципа. Складиштење података се врши употребом *MS SQL* сервера, док се за приступ података користи библиотека *Entity Framework Core*. Делови система који врше двосмерну комуникацију у реалном времену, попут сервиса за нотификације, користе библиотеку *SignalR*.

# Application of reactive distributed programming on the example of online banking services

**Abstract** – The online banking system *BANKX22* developed as part of this master thesis is an open-source web system that offers popular services that can be found in common financial applications such as online payment, money exchange, taking loans, checking account balances and insight into transaction history. The system is implemented to meet the principles of reactive, distributing system such as good responsiveness and elasticity, reliability, that is durability, loose coupling of services and transparent communication of components. Modern Javascript libraries React, Redux and ReduxToolkit were used for implementation of client side of system. Communication between client side and server side is done through .Net Web API framework by REST protocol usage. Server side is implemented in Akka .Net framework which implements model of actor system, suitable for implementation of reactive distributed principles. Data is stored inside MS SQL server, while for data access is used Entity Framework core library. Some part of the system which needs two-way real-time communication, like notification service, use SignalR library.

# Садржај

# 1 Увод

Потреба за програмирањем на реактиван начин се посебно јавља у финансијским системима. Поред потребе ових система да осигурају корисницима сигурност при обављању финансијских трансакција неопходно је да буду и поуздани. Поузданост треба остварити и уколико неки делови,било хардверски или софтверски, откажу. Могућност динамичког додавања и уклањања ресурса у случају повећаних односно смањених захтева је такође важна одлика једног финансијског система.

Банке и друге финансијске институције имају жељу да привуку и опслуже што већи број корисника. Да би тако нешто остариле, систем који нуде мора да настави са радом чак иако има огроман број корисника који треба да опслужи. Систем би требало да има добар одзив према свим тим корисницима, односно да одговори захтевима корисника благовремено, под било којим околностима. Пошто свака машина може отказати у било ком тренутку, такав систем би требало расподелити на више машина. Овај фундаментални захтев за расподелом има као последицу потребу за применом нових архитектуалних образаца. Коришћењем ових образаца постиже се да систем реагује благовремено ка својим корисницима тј. буде одзиван (*енг. responsive*), да реагује на отказ и остане доступан тј. буде издржљив (*енг.* resilient), да може реаговати на променљиве услове оптерећења односно да буде еластичан (*енг.* elastic), да реагује на улаз тако што ће бити вођен порукама (*енг.* message-driven) [2]. Још две карактеристике које бисмо желели да остваримо код имплементације оваквог система су одрживост и проширивост.

У овом раду је представљен систем онлајн банкарства *BANKX22* који се хвата у коштац са свим поменутим изазовима. Коришћењем адекватних, у наставку описаних технологија, систем имплементира одређене архитектуалне обрасце помоћу којих су остварене већи број наведених карактеристика једног таквог реактивног система. Рад је конципиран тако да у првом поглављу читаоцу приближи елементарне концепте, дефиниције и особине којима се треба водити при пројектовању и имплементацији једног реактивног дистрибуираног система. У следећем поглављу су описани обрасци пројектовања који се најчешће користе да би била остварена својства из првог поглавља. Потом се помињу и укратко обајшњавају технологије коришћене за имплементацију система онлајн банкарства *BANKX22*. Последња два поглавља се тичу описа архитектуре, односно имплементације самог система где се на делу могу видети претходно описане карактеристике и обрасци реактивног дистрибуираног система.

# 2 Манифест реактивног програмирања

## 2.1 Реаговање на корисника

Под појмом корисник мислимо како на човека који користи неки сервис посредством другог програма (рецимо претраживача) тако и на други компјутер који делује у име човека директно или индиректно. Битно својство сервиса је ***благовремено слање одговора*** на захтев корисника уколико је то могуће [2]. Одзивни сервиси се фокусирају на пружању брзих и конзистентних времена одгровора, утврђујући поуздане горње границе тако да испоруче конзистентне квалитете услуга. Ово је омогућено низом техника, алата и образаца, неких описаних у овом раду. Ако је ово својство остварено онда је једноставније обрађивање грешака, веће поверење крајњег корисника сервиса па самим тим и већа вероватноћа даље интеракције корисника са сервисом.
Систем би требало да буде у стању да одговори захтевима корисника и при различитом оптерећењу [1]. Ово својство се назива ***еластичност.*** Оно се постиже повећањем или смањењем ресурса додељених сервису који је под оптерећењем. Такви системи не би требало да имају спорне тачке и уска грла што за последицу има дељење и реплицирање компоненти и дистрибуцију улаза међу њима.

## 2.2 Реаговање на грешке

Грешке које се јављају у систему могу бити софтеверске, хардверске и људске. Истек времена без добијеног одговора односно тајмаут се такође може представити као грешка. Претпоставка са којом треба поћи приликом пројектовања и имплементације система није “*ако се грешка јави*” него “*када и колико често*” ће се јављати. Корисника не интресује разлог грешке нити којег је типа, њега се тиче само то да ће морати да настави без добијеног одговора. То за последицу има да ће корисник вероватно користити други сервис. ***Поузданост***, односно ***издржљивост*** система је још једна од кључних карактеристика која указује на то колико брзо систем може да се опорави од грешке [2]. Избегавање грешака никада неће бити потпуно могуће, зато пажњу треба усмерити на толеранцију грешака. Издржљиви системи имају могућност не само да поднесу грешке него и да опораве стање сервиса пре него што је настала грешка. Технике представљене у овом раду које се тичу решавања оваквих проблема су образац *прекидач кола (енг. circuit breaker)* и механизам *надзора (енг. supervision)*.

## 2.3 Вођење порукама

Реактивни систем се ослања на ***асинхроно прослеђивање порука*** како би се успоставила граница између компонената које на тај начин осугиравају лабаву спрегнутост[1], изолованост и локацијску транспарентност[2]. Ове границе такође омогућавају да се грешке делегирају као поруке. Употребом експлицитног прослеђивања порука омогућује се управљање оптерећења, еластичности и контроле тока. То се постиже формирањем и посматрањем редова порука у систему и применом техника зване "*back-pressure"*[3]. Неблокирајућа комуникација допушта примаоцима да конзумирају ресурсе само док су активни што доводи до мање конзумације ресурса од стране система.

## 2.4 Губитак строге конзистентности

Да бисмо кренули са овом темом неопходно је изложити теорему Ерика Бруера (*енг. Eric Brewer's CAP theorem*) која тврди да било који систем који путем мреже дели податке може имати највише два од три пожељна својства [13]:

1) Конзистентност (*енг. Consistency - C*) је еквивалентна поседовању једне актуелне копије података
2) Висока доступност (*енг. High availability - A*) подацима
3) Толеранција у случају дељења мреже (*енг. Network partitions - P*)

На основу ње можемо закључити да у случају да се деси дељење мреже, конзистентност или доступност мора бити жртвована. У случају да се десе промене над подацима током дељења мреже могу настати неконзистентности. Једини начин да се ово спречи је да се забране измене чиме је угрожена доступност. У дистрибуираним системима дакле не можемо применити ACID[4] особине трансакција на којима почивају традиционалне релационе базе података јер желимо да задовољимо принципе издржљивости, скалабилности и одзивности. Ови принципи се требају применити на све делове система како бисмо добили жељене предности, али се на тај начин елиминишу строге гаранције трансакција на којима су грађени традиционални системи. Дистрибуиране системе бисмо требали пројектовати тако да задовоље скуп принципа назван *BASE*:

---

[1] Лабава спрегнутост (*енг. Loosely coupled*) компоненти или сервиса је својство где свака од компоненти има мало или нема уопште знања о дефиницији и имплементацији других компоненти и омогућава лаку замену те компоненте неком другом алтернативом која пружа исте сервисе [15].

[2] Локацијска транспарентност (*енг. Location transparency*) представља својство да изворни код за слање порука изгледа исто без обзира на којој локацији се прималац поруке налази [2]. Компоненте комуницирају на униформан начин дефинисан експлицитиним слањем порука

[3] Back-pressure - могућност преоптерећене компоненте да пошаље поруку надлежним компонентама како би оне могле да смање оптерећење. На тај начин систем грациозно одговара на оптерећење, а не урушава се под њим.

[4] ACID - Атомичност (*енг. atomicity*), конзистентност (*енг. consistency*), изолованост (*енг. isolation*), издржљивост (*енг. durability*)

1) У основи доступан - операције читања и писања су доступне што год је више могуће, али без гаранције о конзистентности;
2) Меко стање - без гарантоване конзистентности, после неког времена, постоји одређена вероватноћа да имамо информације о стању пошто можда још увек није постигнута евентуална конзистентност;
3) Евентуално конзистентан - ако систем функционише и корисник чека одређено време, биће у стању евентуално да зна стање базе, тако да ће даља читања бити конзистентна са његовим очекивањима [10].

*BASE* је био важан корак у разумевању која својстава су корисна, а која недостижна. Још једна препорука која је предложена била је ACID 2.0[5]:

- *Ако се свака акција представи тако да може бити примењена у гомили (асоцијативност) и у било ком поретку (комутатувност) и тако да ако је применимо више пута није штетна (идемпотентност), тада крајњи резултат не зависи од тога која реплика прихвата промене и у ком редоследу се ажурирања шире кроз мрежу- чак је и понављање слања прихватљиво ако пријем још увек није потврђен [2].*

[5] ACID 2.0 је предложен од стране Пета Хеланда на конференцији 2014 а односи се на својства асоцијативности, комутативности, идемпотентности и дистрибуираности

# 3 Обрасци

## 3.1 Образац једноставне компоненте

Дефиниција овог обрасца је: ***Компонента ће радити само једну ствар, али ће је радити у целости*** [2]. Овај образац треба применити када систем обавља више функционалности или су функције система толико комплексне да морају бити подељене у различите компоненте. Образац је изведен из *принципа јединствене одговорности*[6]. Ако пођемо од целог система потребно је да на основу свих његових функционалности почнемо да га разбијамо на мање компоненте где ће свака да обавља једну функционалност. Ако овај процес применимо рекурзивно над добијеним компонентама можемо добити задовољавајућу грануларност и добити хијерархију компонената које задовољавају образац јединствене компоненте и које је онда лакше имплементирати, тестирати и одржавати.

Образац јединствене компоненте је један од основних постулата који је коришћен приликом пројектовања и имплементације онлајн банкарских сервиса *BANKX22*. Можемо за пример узети било који од сервиса система. Ако, на пример, погледамо сервис који враћа кориснику све расположиве валуте, можемо уочити да се сваки захтев корисника система шаље акцији на серверу која је лоцирана у контролеру валута (*енг. CurrenciesController*) и чија једина одговорност је да тај захтев прослеђује у виду поруке одговарајућем актеру модела[7]. Актер има задатак да примљену поруку обради, тако што ће на основу добијеног токена из поруке дохватити идентификацију корисника и на основу ње, добити све валуте које су расположиве кориснику. Како описани посао не одговара обрасцу јединствене компоненте, добијање идентификације корисника се као посебна одговорност прослеђује актеру чија је то једина одговорност, док се посао читања валута из базе података прослеђује посебном актеру. Оваквим приступом је постигнуто да сваки од актера који учествују у обради захтева раде једну ствар и то у целости, актер који чита идентификацију корисника из базе (*енг. UserIdRetrieverActor*), актер који чита валуте за датог корисника из базе (*енг. CurenciesStorageActor*) и актер који врши синхронизацију комуникације између претходно поменута два актера и контролера (*енг. CurrenciesGetterActor*).

[6] Принцип јединствене одговорности (*енг. Single Responsibility Principle*) је принцип који наводи да сваки модул, класа или функција у програму мора имати одговорност над тачно једним делом функционалности тог програма и њу треба да енкапсулира [26].

[7] Актер представља основни конструкт модела система актер који гарантује енкапсулираност и сигурност извођења операције у вишенитном окружењу (за више детаља погледати поглавље 4.4)

## 3.2 Образац језгра грешке

Дефиниција обрасца: ***У хијерархији супервизије, чувати важна стања апликације или функционалности близу корена, а делегирати ризичне операције ка лишћу*** [2]***.*** Овај образац се надограђује на образац једноставне компоненте и примењив је када год су компоненте са различитим вероватноћама неуспеха и различитом поузданошћу захтева, комбиноване у истом систему. Неке функције система никада не смеју да падну, док су друге неопходно изложене неуспеху. Образац је толико значајан у реактивном програмирању да је развојно окружење модела актера, коришћено за имплементацију серверске стране онлајн банкарског система *BANKX22*, названо *Akka* што у ствари представља палиндром од Језгра Актера (*енг. Actor Kernel*) и реферише на овај образац. Суштинска функција обрасца језгра грешке је да интегрише операциона ограничења система у његову декомпозицију проблема базирану на одговорности.

Примена овог обрасца је свеприсутна у систему онлајн банкарства *BANKX22*. Као добар пример можемо навести актера који комуницира са екстерним системом како би добио информације о тренутним курсу мењања из једне валуте у другу. Како услуге екстерних сервиса могу бити непоуздане и ризичне, јер одржавање и квалитет рада екстерног сервиса није у нашој надлежности, актер који комуницира са њим је лист у дрвету хијерархије актера који су надлежни за процес размене средстава из једне валуте у другу. Супервизор овог актера, актер који га је креирао и који комуницира са њим, је задужен за даљи ток уколико дође до грешке. Уколико информације о курсу нису из неког разлога добијене из екстерног система супервизор покушава да их добије из локалне базе у којој су биле сачуване приликом неког од претходних размена и уколико те информације постоје и задовољавају одређене услове биће враћене као валидне за тренутну размену. Како супервизор има већу одговорност тако је он ближи корену у дрвету структуре актера.

## 3.3 Образац “пусти-да-се-сруши”

Дефиниција обрасца: ***Радије рестартуј целу компоненту него да обрађујеш интерне падове*** [2]***.*** Идеја овог обрасца је да су неке ретке грешке, како софтверске тако и хардверске, обично скупе за дијагнозу и поправку, тако да се преферира опоравак система рестартовањем његових делова. Отклањање грешака хијерархијским рестартовањем омогућава поједностављивање самог модела грешака и у исто време доводи до веће робустности система који онда има шансу и да преживи непредвиђене грешке. Последице овог обрасца су да свака компонента мора бити толерантна на рестарте и падове у било ком тренутку, као што рецимо нестанак струје може настати било када, да компонента мора бити енкапсулирана тако да се грешке не могу проширити на друге компоненте, да мора бити толерантна на пад компоненте са којом комуницира, да сви ресурси које је компонента користила морају бити повраћени након рестарта, да сваки захтев послат компоненти мора бити самоописив тако да се његово обрађивање може наставити након рестартовања са што мање утрошка. Да би се одређена компонента

рестартовала након грешке, потребно је саму грешку уочити на неки начин. Једна од техника која се користи је да компонента након њеног креирања крене да шаље периодичне поруке свом супервизору да је све у реду. У случају да супервизор не добије поруку тог типа примењује “пусти-да-се-сруши” образац. У ретким случајевима може да се деси да компонента не ради исправно, али ипак шаље потврдне поруке свом супервизору. Из тог разлога се у садржај те поруке умећу информације о квалитету рада сервиса као што су стопа падова, латентност одговора итд. Ове поруке се називају “*откуцаји срца”.* Још један начин на који супервизор може бити информисан о грешци је да компонента сама дијагностикује неку грешку и пошаље поруку супервизору. То се може десити на пример у случају ако компонента комуницира са неком екстерном библиотеком која испаљује изузетак у случају грешке. Ова два начина регистровања грешака се могу користити у комбинацији како би покривеност била боља.

Развојно окружење *Akka .NET* користи овај образац као један од основних механизама одбране од грешака. Сваки актер система има уграђен механизам да приликом настанка неке непредвиђене грешке јави свом супервизору о њој [1]. Уколико сурпервизор одлучи да грешка нема утицаја на њега или на друге подређене актере (због пажљивог пројектовања система и употребе обрасца језгра грешке) рестартује актера од кога је стигла порука о грешци. Ово заправо значи да ће бити креирана нова инстанца актера и да ће бити додељена иста референца која је енкапсулирала стару инстанцу. Прецизнији ток догађаја који се одвија током рестартовања је да се прво врши суспендовање актера (нема више обраде порука док не добије одобрење за наставак рада) и рекурзивно се суспендују сва деца тог актера. Потом се позива предефинисани метод актера *PreRestart* који шаље поруке прекида сваком актеру детету и позива предефинисани метод *PostStop.* Затим се чека да сва деца актери прекину са радом и када последње дете пошаље поруку о престанку рада креира се нова инстанца актера. По креирању нове инстанце позива се предефинисани *PostRestart* метод на новој инстанци и шаље се захтев за рестартовање сваком актеру детету [4]. Свако дете прати исти след догађаја. На крају актер може да настави са радом. Комбинација овог обрасца са неком од техника чувања тренутног стања актера се користи у систему онлајн банкарства *BANKX22* и представља главни начин на који се систем носи са непредвиђеним понашањем неког његовог дела.

## 3.4 Образац прекидач кола

Дефиниција обрасца: ***Штити сервисе прекидајући конекцију ка њиховим корисницима током дужих услова отказа*** [2]***.*** Овај образац описује на који начин се безбедно могу повезати други делови система тако да се грешке не рашире међу њима. У случају када је компонента претрпана захтевима или крене да пада, ток захтева до те компоненте може бити намерно прекинут. На овај начин се постиже да прималац захтева има времена да се опорави од могућег пада због оптерећења, а пошиљалац може да одлучи да је захтев неуспешан, а да не чека негативни одговор. Постоје два стандардна начина да се имплементира овај образац. Први је тај да прекидач кола врши комуникацију

са компонентом која прима захтеве. Када добије одговарајући број негативних захтева од компоненте прекидач кола прекида коло и сам креће да враћа негативне поруке пошиљаоцу. Након одређеног времена проверава стање оптерећења компоненте примаоца и уколико је повољно поново затвара коло дозвољавајући захтевима да стигну до компоненте. Овај начин је пример општег прекидача кола. Други начин је да постоји специфичан прекидач кола, рецимо за сваког корисника, и да он мери којом су брзином захтеви тог клијента упућивани компоненти која прима захтеве. Када брзина комуникације пређе одређени лимит, прекидач кола прекида коло и креће са одбијањем захтева. Постоји могућност да се одређеним клијентима забрани приступ дуготрајно уколико им константно брзина комуникације прелази лимит. На тај начин шаље се упозорење да треба погледати имплементацију компоненте пошиљаоца и лимитирати брзину којом она шаље захтеве. Имплементацијом овог обрасца обезбеђујемо да систем буде издржљивији него када бисмо сваки случај грешке обрађивали појединачно. Цена прекидача кола је та да сви позиви пролазе кроз још један корак порвере и да тајмаути морају бити планирани, стога га не бисмо требали имплементирати на високом нивоу грануларности.

Пример примене овог обрасца на систему онлајн банкарства *BANKX22* можемо видети у актеру који врши добијање информација о размени валута са екстерног система. Овај актер користи прекидач кола као вид заштите у случају да је екстерни систем преоптерећен и да му је време одзива лоше или ако је недоступан. У тим случајевима прекидач је конфигурисан тако да уколико добије пет неуспешних одговора од екстерног система улази у затворено стање и уместо да и даље шаље захтеве екстерном систему сам враћа негативне одговоре. Прекидач кола ће бити у затвореном стању један минут након чега прелази у полу отворено стање. Када следећи захтев стигне биће прослеђен екстерном систему и уколико је захтев успешан прекидач кола прелази у отворено стање и наставља са уобичајним радом. Уколико је захтев неуспешан прекидач се враћа у затворено стање још један минут и потом опет проверава доступност екстерног система. Овим приступом је систем онлајн банкарства *BANKX22* у знатној мери заштићен од пада екстерног система за добијање информација о конверзији валута.

## 3.5 Активни-пасивни образац копирања

Дефиниција обрасца: ***Чувај више копија сервиса који се извршава на различитим локацијама, али дозволи измене стања на једној локацији у једном тренутку*** [2]***.*** Образац је добио овај назив како би означио да за опоравак једног сервиса, ако у потпуности престане са радом, укључујући и губитак његовог складишта података, потребно је расподелити његову функционалност и скуп података са којима ради на више физичких локација. Како би оваква имплементација могла да ради потребан је одређен вид координације, поготово за операције које мењају трајна стања сервиса. Циљ овог обрасца је да у једном тренутку дозволи промену стања на само једној копији. Тиме постижемо да измене стања не захтевају консензус између копија. Битан је консензус у одабиру активне реплике. Како би било могуће знати локације свих копија сервиса потребно је имплементирати регистар сервиса који чува адресу сваке копије и коме друге компоненте

могу приступити. При подизању система, компонента која има више копија уписује њихове адресе у регистар. На овај начин је олакшано новим репликама да буду додаване, а старе замењиване динамички. Механизам избора активне копије може бити доступан клијентима који га ослушкују и имају благовремене информације о промени активне реплике. Мана оваквог приступа је што су клијент и сервис уско спрегнути у смислу дељења протокола. Други начин би могао да буде да свака копија има могућност прослеђивања захтева активној копији. Тиме избегавамо строгу спрегнутост, али постоји могућност да редослед пристизања захтева буде нарушен ако су захтеви прослеђени од различитих реплика па губимо детерминизам код обраде захтева. Постоје два типа отказа која се могу јавити код овог обрасца. Може се десити да дође до отказа пасивне копије и тада обично долази до повећања мрежног саобраћаја како би ова копија која је рестартована добила најновије стање у којем се налазе остале копије. Други случај који може да се догоди је да дође до отказа активне копије. Тада ће постојати период у коме неће бити активне копије, зато што је потребно време да се установи губитак активне копије и да се прошири знање да ће нова инстанца бити потребна. Ова координација задатака мора бити обављена пажљиво како бисмо били сигурни да се стара инстанца не може умешати са будућим операцијама ако постане доступна поново након дељења мреже. Због поменутих ситуација могу постојати периоди када ће овај сервис бити недоступан, све док се не изабере нова активна реплика која је у конзистентном стању и при томе пазећи да се не дође у ситуацију постојања две активне реплике. Овај образац је одличан када нам је потребно очување конзистентности по цену доступности сервиса у случајевима отказа. Систем онлајн банкарства *BANKX22*, помоћу развојног окружења *Akka .NET*, користи модул који се зове *Cluster* и који дозвољава репликацију како сервиса тако и целог система. Ове реплике, назване чворови, се могу сместити на различите физичке локације и употребом функционалности које нам пружа *Cluster* модул се лако имплементира активни-пасивни образац копирања.

## 3.6 Обрасци комуникације

У реактивним дистрибуираним системима посебну пажњу треба обратити на пројектовање добре комуникације међу компонентама система на свим нивоима грануларности.

Образац који је доста заступљен и примењен у свим доменима је ***образац захтева и одговора*** (*енг. Request-response pattern*) који наводи да сваки захтев треба да садржи адресу пошиљаоца захтева, како би прималац могао да одговори на захтев. Овај образац је нашао своју примену у бројним протоколима као што је рецимо *HTTP* [8] где је адреса пошиљаоца захтева уграђена у механизме протокола. Образац се користи експлицитно у моделу актера где је приликом комуникације између актера потребно укључити и референцу актера у састав поруке која саджи адресу актера који шаље захтев.

[8] *HTTP* (*енг. Hypertext Transfer Protocol*) апликациони слој веб протокола за слање документа хипермедије као што је *HTML*

Још један образац који треба примењивати је ***образац самосталне поруке*** (*енг. Self-Contained Message pattern*) који наглашава да свака порука треба да садржи све информације потребне за обраду захтева као и да се разуме његов одговор [2]. То заправо значи да ако је могуће, најбоље је у поруци укључити све информације као што су адреса пошиљаоца, јединствени број поруке, коме је порука намењена и сам садржај поруке. На тај начин пошиљалац и прималац би били ослобођени од чувања стања о процесу комуникације што би значајно допринело одзивности као и издржљивости система. Систем онлајн банкарских сервиса *BANKX22* користи *HTTP* како би корисник посредством корисничког интерфејса комуницирао са слојем *.NET Web API* система (за више детаља погледати 4.5 .Net Web API интерфејс за програмирање апликација), док слој *Web API* комуницира са осталим сервисима система употребом обрасца самосталне поруке. У неким случајевима када је садржај поруке превелики, рецимо ако су укључени неки медији као на пример слике или фајлови, тада није пожељно послати све у једној поруци због могућег застоја у мрежи. У тим случајевима се одржавање стања комуникације треба свести на минимум. То се постиже слањем метаинфомрација у првој поруци, а затим, ако се добије потврдан одговор од примаоца, шаље се садржај поруке.

Образац који је постао златни стандард у *Akki* се назива ***упитини образац*** (*енг. Ask pattern*). Образац наводи да обрађивање одговора треба делегирати додељеној компоненти која је креирана са тим разлогом [1]. Након што се обави циклус “захтев-одговор” често се дешава да тренутни процес мора да настави са наредним корацима. Да бисмо ово омогућили актер би могао да чува мапу корелација индентификација са информацијама које се тичу процеса наставка или би могао да створи новог актера чији би задатак био да настави са процесом по пристизању одговора. Овај приступ је уграђен у развојно оркужење модела актера где је додељена одговарајућа синтакса овом обрасцу. Након пристизања одговора дете актер које је одређено за обраду одговора може да пошаље родитељу поруку успеха, поруку неуспеха или поруку истека тајмаута. Ова информација се смешта у објекат који се назива ***будућа вредност*** *(енг. Future*). Будуће вредности омогућавају имплементацију родитеља без блокирања током чекања резултата.

У систему онлајн банкарских сервиса *BANKX22* образац комуникације је нашао своју примену у комуникацији контролера и актера. Како је контролеру неопходно да сачека на одговор и проследи га корисничком интерфејсу он путем упитног обрасца шаље захтев у виду поруке актера на даљу обраду и добија објекат будућност који након разрешавања у конкретан одговор враћа корисничком интерфејсу. Морамо водити рачуна да не претерамо са коришћењем горе поменутих образаца јер се то може одразити на перформансе система. Рецимо ако се строго придржавамо обрасца захтева и одговора, тада ће одговори увек стизати са сервиса коме је захтев упућен. Ако у комуникацији са клијентом постоји сервис који представља посредника у комуникацији са другим сервисима, који је контролор захтева и прослеђивач захтева на праву адресу, тада ако се придржавамо обрасца захтева и одговора, сваки одговор ће бити прослеђиван клијенту такође преко посредника. Ово може да изазове ефекат “уског грла” поготово ако су одговори неки мултимедијални садржаји.

Да до овога не би дошло можемо применити ***образац тока напред*** (*енг. Forward flow pattern*) који наглашава да ток порука треба да буде директан према одредишту тамо где је то могуће [3]. То би значило да одговори различитих сервиса не иду преко посредника

него директно клијенту који је упутио захтев. Образац тока напред је примењен у делу система онлајн банкарских сервиса *BANKX22* који је имплементиран коришћењем модела актера. Након што је контролер послао иницијалну поруку систему моделу актера, референца контролера се пропагира унапред све до актера који врши финалну обраду захтева и резултат шаље директно контролеру.

У систему као што је систем онлајн банкарства *BANKX22* морамо водити рачуна на који начин обављамо трансакције. Како је систем дистрибуиран тако стандардни механизми попут закључавања података којима се постиже атомичност трансакције неће бити могући, поготово ако се деси дељење мреже. Трансакција као што је пребацивање новца са једног рачуна на други може захтевати улазне податке из више различитих компоненти и такође утицати на модификацију више компоненти. У оваквој ситуацији погодан је ***сага образац*** (*енг. Saga pattern*). Дефиниција сага обрасца је да треба поделити дистрибуиране трансакције које имају дуги животни век у више брзих локалних са одговарајућим акцијама у случају неуспеха [27]. Другим речима, треба извршити креирање компоненти са кратким животним веком које би обавиле извршавање низа акција дистибуираних преко више компоненти. Оваква компонента служи као оркестратор у преносу новца и обезбеђује одговарајуће акције у случају да из неког разлога до трансфера не може доћи. Она обезбеђује да се трансакција изврши или да се прекине обезбеђујући атомичност и евентуалну конзистентност, а не блокирајући компоненте које учествују у преносу. Још једна карактеристика саге је да она садржи сво знање о процесу преноса, тј. компонете које учествују у преносу су ослобођене те одговорности. У случају промена ово је пожељна карактеристика јер се све промене обављају на једном месту. Сва стања која има сага компонента требају бити сачувана, тако да ако дође до отказа саге могу лако да буду враћена приликом рестартовања.

Приликом комуникације између сага компоненте и компоненте на којој се налазе одговарајућа средства може доћи из неких разлога до губитка порука. Како бисмо обезбедили поузданије извршавање трансакције потребно је да сага понавља слање иницијалне поруке све док не стигне потврдан одговор, да у ту поруку укључи информацију о идентификацији самог захтева како би компонента могла да га искористи и спречи вишеструко скидање средстава за исти захтев и да компонента мора увек да врати одговор чак и за поновно послате поруке. Овакав вид комуникације се назива ***образац пословног руковања*** (*енг. Business Handshake pattern*) зато што је круцијално да одговор компоненте имплицира успешно процесиран захтев [3].

## 3.7 Обрасци контроле тока

Како би дистрибуирани реактивни систем био отпоран на променљива оптерећења потребни су механизми који би спречили отказ компонената због превеликог броја захтева који су јој упућени. Један такав механизам се назива ***образац вучења*** (*енг. Pull pattern*). Образац вучења наводи како потрошач захтева треба да тражи од произвођача већи скуп података [2]. На тај начин потрошач има контролу над количином захтева коју је у стању да

обради без преоптерећивања. Број тражених захтева може бити прилагодљив динамички како би се постигао оптималан рад потрошача захтева у било ком окружењу, како приликом развоја и тестирања, тако и у самој продукцији. У моделу актера, актер произвођач може креирати одређен број актера потрошача и тај број прилагођавати у односу на укупан број захтева и ресурса како бисмо постигли оптималне перформансе и како актер произвођач не би морао узалудно да чека док му не стигне захтев од актера потрошача за новим скупом захтева.

Постоје ситуације када није могуће контролисати брзину слања, односно примања података, па самим тим није могуће применити образац вучења. Рецимо ако постоји комуникација система са неким екстерним сервисом над којим немамо контролу. Тада се може применити ***образац управљивих редова*** (*енг.* The Managed Queue pattern). Дефиниција обрасца управљивих редова је да треба управљати улазним редовима и реаговати складно њиховом нивоу испуњености. Идеја је да се структура података која се назива ред имплементира између компоненте која прима захтеве и екстерне компоненте која их шаље. Они онда могу да врше надгледање и вођење перформанси система порука. Иако овакав механизам претставља одличну линију одбране од преоптерећења, може се десити да је брзина захтева коју шаље екстерна компонента толико велика да ће доћи до пуњења меморије реда. У једном тренутку систем више неће имати расположиву меморију коју може да додели реду па би морао да угаси неку другу компоненту или да падне.

У таквој ситуацији боље је применити ***образац одбацивања*** (*енг. Drop pattern*) који наводи да захтев боље одбацити него пустити да неконтролисано неуспе. Тако да је боље да компонента где је ред имплементиран има могућност одбацивања захтева када дође до попуњености меморије реда него пустити систему да се носи са неконтролисано неуспелим захтевом. Како бисмо контролисали одбацивање захтева добро решење је имплементирати алгоритам одбацивања на основу односа уноса и вађења из реда. На основу тог односа се може израчунати вероватноћа одбацивања захтева. Што је ред више попуњен, то значи да је брзина пристизања захтева већа од брзине вађења захтева из реда, то је вероватноћа одбацивања већа. Овим се постиже да ће доћи до уједначавања захтева убачених у ред и обрађиваних захтева, а такође се брзина слања може прилагодити или перформансе обраде захтева.

## 3.8 Обрасци управљања и чувања стања

Већина компонената система ће морати на неки начин да управљају својим стањима. Та стања је у многим случајевима потребно чувати ради опоравка у случају отказа. Како би управљање стањима било ефикасније пожељно је одвојити га од домена бизнис логике и комуникације. За ову сврху можемо користити ***образац објекта домена*** (*енг. The Domain Object pattern)* који описује начин на који можемо одржати јасну границу између интреса пословне логике, управљања стања и комуникације [2].

Ако користимо образац објекта домена тада је свака промена стања упарена са догађајем који се шаље клијенту који је затражио промену. Ови догађаји садрже целу

историју на који начин је објекат домена еволуирао па их можемо искористити као метод очувања промене стања. То постижемо тако што изводимо промену стања применом одређеног догађаја и правимо их трајним чувајући те догађаје у логу. Овакав приступ се назива ***образац догађаја као извора*** (*енг. Event-Sourcing pattern*). Име је добио зато што саме догађаје правимо изворима истина за сачуване објекте домена, а не трансформимишемо их у ажурирања једног складишта [1]. Како су догађаји чувани у уређеном поретку у логове, довољно је њихово надовезивање након пада како би се компонента вратила у одговарајуће стање.

Систем онлајн банкарства *BANKX22* користи погодности развојног окружења *Akka .NET* у виду *UntypedPersistedActor* актера који пружа имплементиране механизме за управљање и чување стања. Више детаља о употреби ће бити пружени у наставку текста.

# 4 Технологије

## 4.1 Библиотека *React*

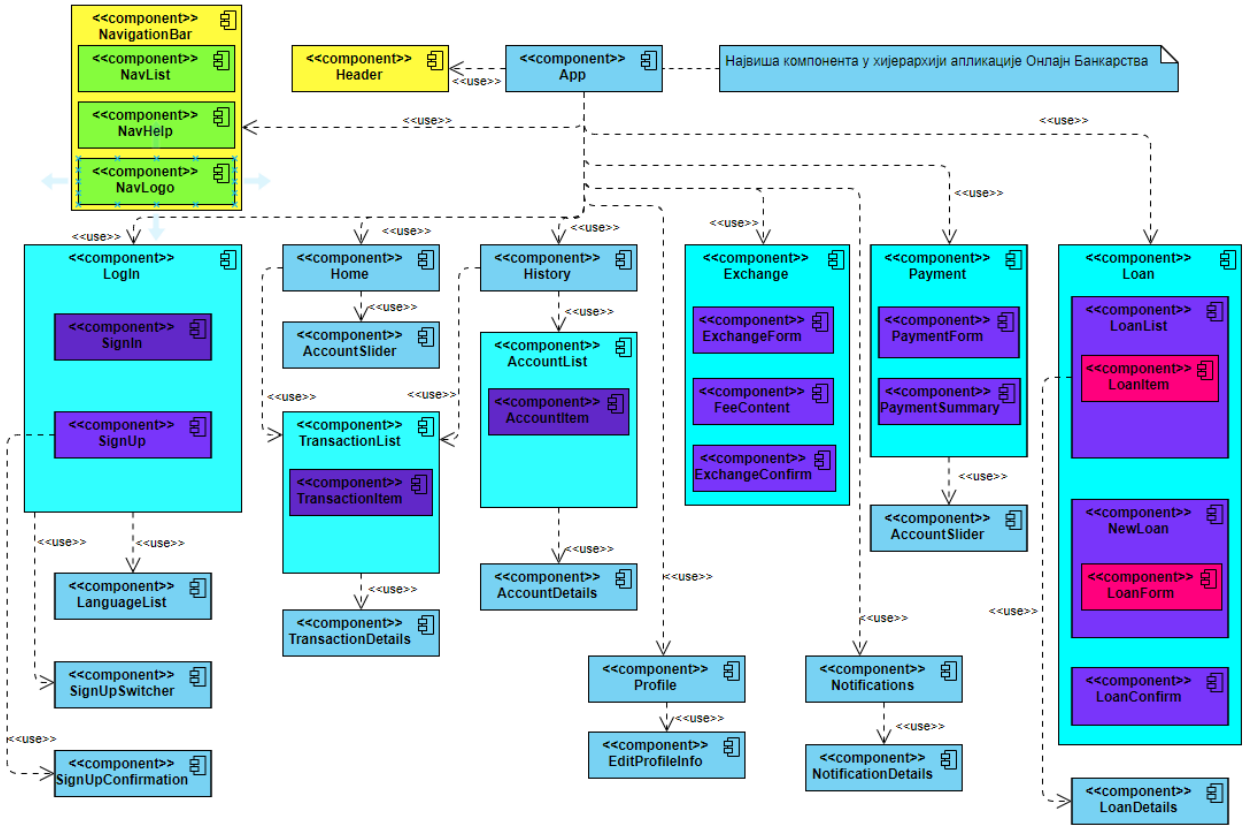
*React* је бибилотека написана у програмском језику *Javascript* која служи за прављење корисничког интерфејса. *React* је декларативна библиотека која је базирана на писању компоненти коришћењем реактивних принципа [7]. Због ових карактеристика коришћена је приликом израде клијентског дела система онлајн банкараства *BANKX22*.

Када правимо апикацију користећи библиотеку *React* ми у ствари апликацију замишљамо као хијерархију компоненти. На врху хијерархије се налази компонента која представља целу апликацију, а потом се она дели на основу својих функционалности на друге компоненте. Тако понављајући рекурзивно процес долазимо до компоненти на дну хијерархије код којих свака компонента представља једну функционалност апликације. Овим поступком је у ствари праћен образац једноставне компоненте (3.1 Образац једноставне компоненте) и образац језгра грешке (3.2 Образац језгра грешке).

Свака компонента има два објекта која су неизоставна и представљају суштину имплементације компонената. Први је објекат стања у коме се чува и ажурира локално стање компоненте и на основу кога библиотека врши поновни приказ те компоненте. Објекат стања се иницијализује у циклусу креирања компоненте и на основу њега се врши иницијални приказ, а затим, када се неки део стања промени врши се поновно исцртавање те компоненте са ажурираним стањем. Други објекат се користи као начин комуникације између компонената на различитим нивоима хијерархије и назива се *props.* Када родитељ компонента жели да обавести дете компоненту, она у *props* објекат додаје одговарајуће својство са жељеном вредношћу. Сваки пут када дете компонента прими нову вредност она се поново приказује кориснику. Уколико бисмо желели да се комуникација одвија у супротном смеру, да дете иницира комуникацију са родитељем, потребно је да приликом имплементације, родитељ проследи детету помоћу *props* објекта функцију коју ће дете компонента позвати када се деси одређени догађај и на тај начин обавести родитеља о жељеној акцији.

Видимо да овакав начин комуникације одудара од принципа комуникације заснованог на порукама који нам налаже манифест реактивног програмирања. Ипак ако узмемо у обзир чињеницу да се клијентска страна апликације углавном налази на једној машини, тако да локална транспарентност није толико битна карактеристика па самим тим и комуникација заснована на порукама није значајна у овом случају.

На Слици 4.1 дат је хијерархијски приказ клијентског дела апликације онлајн банкарства *BANKX22*. За овакву структуру апликације идеално је коришћење библиотеке *React*. *App* компонента представља „корен“ стабла у дрвету компоненти. Како функционалност апликације разлажемо на одговарајуће сервисе тако добијамо следећи ниво хијерархије стабла. Ако гледамо појединачно сваки сервис можемо опет применити образац једноставне компоненте и добити компоненте које представљају појединачне функционалности сваког сервиса. Рецимо ако погледамо *Payment* компоненту на слици можемо приметити да је она разложена на компоненте *PaymentForm* и *PaymentSummary.*




*Слика 4.1 Дијаграм компоненти апликације онлајн банкарства BANKX22 имплементираних у библиотеци React*

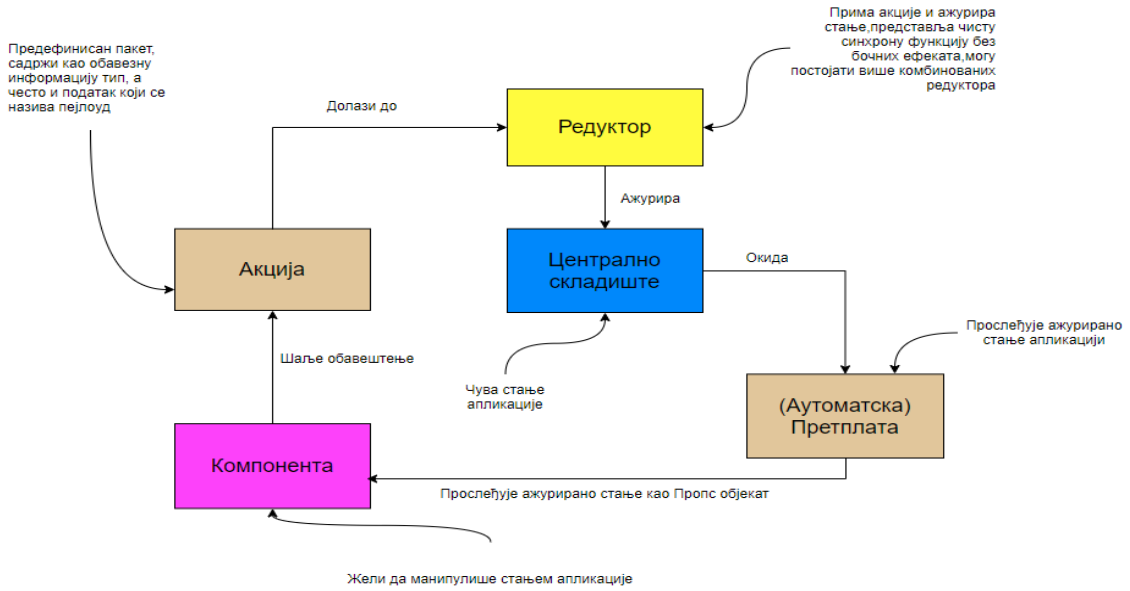
Понављајући поступак компоненте можемо у *React-y* врло ефикасно и лако разлагати до оптималног нивоа грануларности. На Слици 4.1 видимо да компонента *Payment* користи компоненту *AccountSlider* како би у слајдеру приказала списак свих рачуна којима корисник тренутно располаже. Ова компонента је такође коришћена од стране *Home* сервиса. *React* у овим ситуацијама прави две одвојене копије компоненти, тако да промена стања код једне компоненте неће утицати на промену стања код друге. Овим је обезбеђено својство *непромењивости објекта* које је једно од центарлних концепата како у функционалном тако и у реактивном програмирању.

## 4.2 Библиотека *Redux*

*Redux* представља библиотеку написану у *Javascript* језику која помаже у одржавању глобалног стања апликације, односно стања које је потребно у многим деловима апликације. Састоји се од скупа образаца и алата који олакшавају разумевање о томе када, како и зашто је стање у апликацији ажурирано и како се сама апликација понаша када је стање промењено [8].

Систем онлајн банкарства *BANKX22* садржи одређен број података који се требају одржавати на глобалном нивоу. За податке као што су кориснички идентитет, рачуни корисника и обављене трансакције није довољно локално стање које нам пружа *React*-ова компонента јер се ови подаци користе у више различитих сервиса. Такође, константно довлачење ових компонената са серверског дела апликације би било прескупо и непотребно. За податке оваквог типа идеална је употреба библиотеке *Redux*.

Напоменули смо да је главни значај ове библиотеке олакшавање одржавања глобалног стања. Сходно томе једна од главних компоненти библиотеке *Redux* је централно складиште у коме се то стање чува. Стање може бити промењено само посредством специјалне функције која се назива *редуктор (енг. reducer)*. *Редуктор* добија као параметар специјално дефинисан пакет података који се зове акција и на основу типа акције ажурира одређен део стања централног складишта[9]. Акција је *Javascript* објекат који се састоји од типа, а често и од података по конвенцији названих *payload*. Сваку акцију одашиље *Redux*-ова компонента након потребе за променом глобалног стања апликације. Приморавањем да свака промена буде описана акцијом омогућава се јасно разумевање о томе шта се догађа унутар апликације. Како бисмо приликом писања апликације одржавали добре принципе порграмирања као што су енкапсулираност, одвајање брига и свели дуплирање кода на минимум, пожељно је акције одашиљати из компоненти посредством специјалних функција које се зову креатори акције (енг. *action creator*). Као што само име функције каже, њен циљ је да креира и врати одређену акцију која ће бити послата редуктору. Описан процес је приказан на Слици 4.2.




*Слика 4.2 Процес ажурирања глобалног стања клијентског дела апликације коришћењем библиотеке Redux*

[9]Библиотека *Redux* прати концепте функционалне парадигме тако да не постоји директно мењање података. Када се у тексту наводи да се стање ажурира мисли се да се креира потпуно нов објеката који представља стање тако што се непромењени делови стања копирају, а додају промењени и ново стање се ставља на врх складишта као тренутно стање апликације.

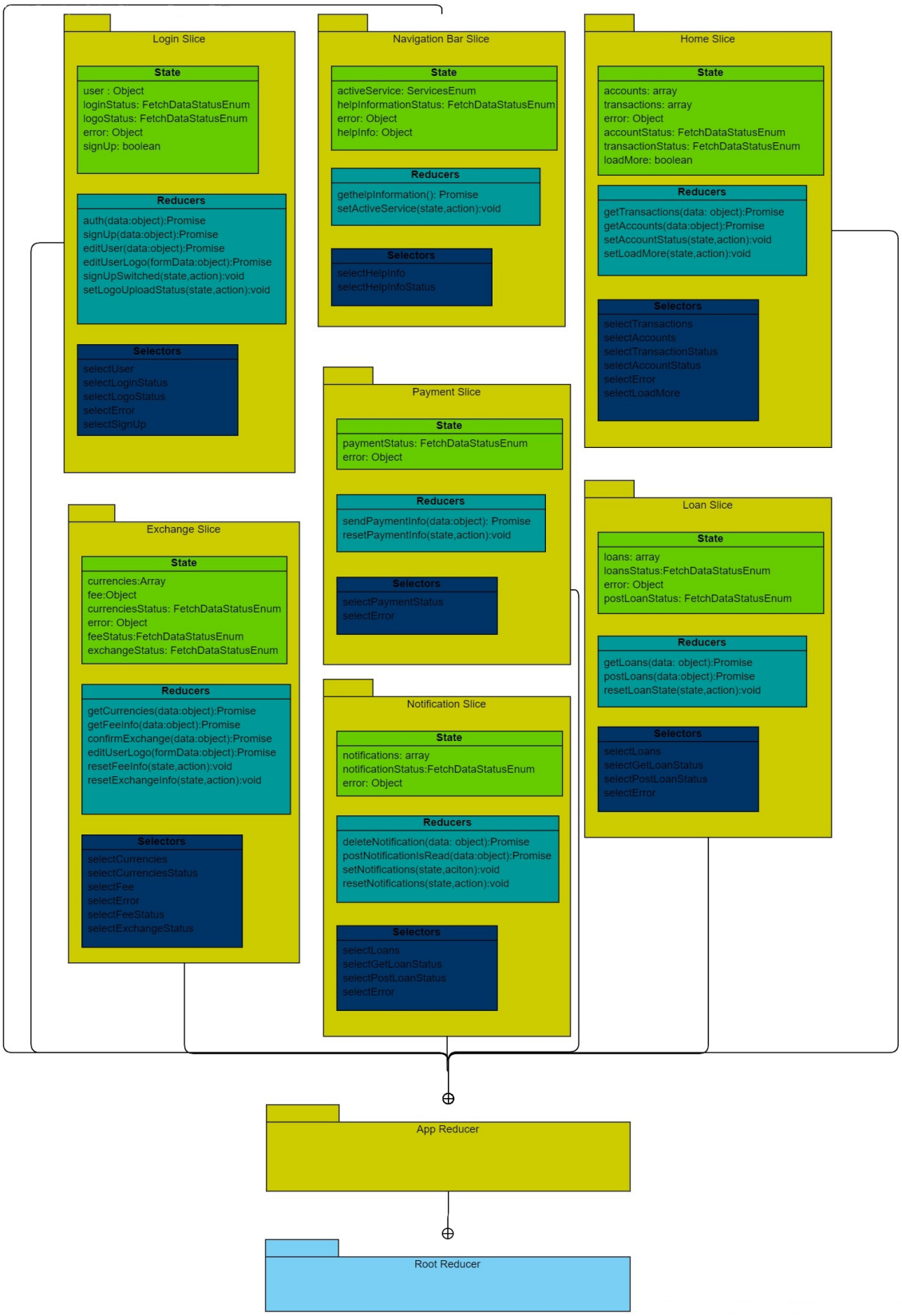
## 4.3 *Redux*-ов скуп алата

*Redux*-ов скуп алата (*енг. Redux Toolkit*) представља вишу апстракцију која помаже да се поједноставе најчешћи случајеви употребе који се јављају у библиотеци *Redux* [9]. Коришћењем овог алата добијамо читкији код јер користимо предефинисана правила и уграђене функције овог скупа.Такође долази и до елеминисања кода који се бави конфигурацијом *Redux*-овог складишта и *посредника (енг. Middleware-а)*. *Redux*-ов скуп алата нам пружа неколико функција које можемо користити и додаје зависности неких пакета који се често користе заједно са *Redux*-ом. Оваквим приступом осигуравамо да ће захтев бити извршен тачно једном.

Када желимо да *Redux* повежемо са апликацијом први корак би требао бити креирање и конфигурисање складишта. Том прилоком се функција која врши креирање прослеђује као параметар редуктору који ће ажурирати стања и скуп опција као што су на пример *посредници*[10] који ће бити коришћени, алати за лакши развој и праћење стања апликације и многи други. Веома често се за многе апликације користи исти скуп опција. Да би се програмер апликације ослободио обавезе ручног конфигурисања *redux*-ов скуп алата укључује функцију *configureStore* која врши аутоматско иницијално конфигурисање и тиме ослобађа програмера да пише конфигурације које се најчешће употребљавају у апликацијама.

Систем онлајн банкарства *BANKX22* користи концепт *парчића (енг. slice)* који је главна карактеристика *Redux-о*вог скупа алата. Сваки *парчић* представља део стања главног складишта који има своје име, иницијално стање и скуп редуктора којим се манипулише тај део стања. Такође сваком *парчићу* можемо придружити и скуп селектора помоћу којег *React* компонента лакше приступа неком сегменту стања. На овај начин је извршена јасна подела и организација кода на основу бизнис логике апликације и програмер је приморан да се придржава добирх образаца при самом развоју. Ови *парчићи* се спајају у једну целину коришћењем комбинованог редуктора који се назива *AppReducer. AppReducer* редуктор би се могао проследити као такав функцији која конфигурише централно складиште. Ипак у систему онлајн банкарство *BANKX22* се јавља потреба за враћањем глобалног стања на иницијалне вредности у одређеним случајевима. Рецимо један случај би био одјављивање корисника из апликације. Како приликом поновног пријављивања корисник не би затекао неке сервисе у неконзистентном стању редуктор *AppReducer* “обмотавамо*”* са кореним редуктором који има функционалност ресетовања стања и њега шаљемо складишту. На Слици 4.3 можемо видети скуп свих *парчића* који се користе у систему онлајн банкарство *BANKX22*.

[10] Неизоставни посредник сваке апликације која комуницира са сервером је “thunk middleware” који служи за асинхроно извршавање неких акција прослеђених редуктору од стране компоненте.




*Слика 4.3 Приказ свих парчића као и скуп редуктора, селектора и иницијалних стања коришћених у систему онлајн банкарства BANKX22*

## 4.4 Модел актера

*Модел актера* (*енг. Actor model*) је модел конкурентних израчунавања у којима се свака комуникација јавља између ентитета који се називају *актери* путем прослеђивања порука актера која шаље поруку и редова који су сандучићи за пошту на страни актера примаоца [1][4]. Овакав концепт се први пут јавља у програмском језику *Erlang*.

*Актери* задовољавају сва три постулата манифеста реактивног програмирања, вођени су порукама, асинхрони су и неблокирајући. Организација актера је таква да задовољавају надгледану хијерархију за управљање неуспесима на различитим нивоима важности. Када се изузетак јави унутар актера та инстанца може бити настављена, рестартована, стопирана чак иако се грешка јавила унутар друге асинхроне нити.

Корисничко окружење *Akka .NET* које је базирано на концепту модела актера обезбеђује проксије кроз које интеракција свих актера мора да тече. Тај прокси се зове референца на актер (*енг. ActorRef*). То за последицу има да сам актер нема потребе да зна физичку локацију актера коме шаље поруку. Ово својство се назива локацијска транспарентност (*енг. location transparency*). Локацијска транспарентност нам омогућава да, рецимо, стартујемо новог актера и преусмеримо све поруке ка њему у случају да тренутни актер престане са радом у сред конверзације без потребе за мењањем кода саме поруке. Мана оваквог приступа је та да актер који шаље поруку мора имати референцу актера који прима ту поруку, односно произвођач поруке и потрошач су уско спрегнути. Предност оваквог приступа је та да отказ једног актера не утиче на рад другог зато што је једина тачка приступа референца актера.

Извршавање унутар актера је синхроно зато што се обавља са једном нити и друга нит не може да изврши позив унутар актера директно. Овим приступом је обезбеђена енкапсулација променљивог стања унутар инстанце актера тако да нема захтевања катанаца како би се лимитирао истовремени приступ променљивој. Модел актера подржава како вертикално скалирање (коришћење више језгара унутар једне машине), тако и хоризонтално скалирање (коришћење других ресурса који су на мрежи). Актери пружају механизме супервизије у подршци толеранције грешака. Због наведених својстава имају све карактеристике за изградњу реактивног дистрибуираног система.

Развојно окружење *Akka .NET* дозвољава да се програмери система усредсреде на бизнис процесе уместо да сами пишу кôд који пружа поуздано понашање, толеранцију грешака и високе перформансе. *Akka* нам пружа вишенитно понашање без потребе коришћења конструкта конкурентности као што су атомичност, закључавања и проблема који се тичу видљивости меморије. Самим тим омогућава писање конкурентих, паралелних и дистрибуираних система лакшим.

Механизам рада актера је следећи:

1) Актер додаје примљену поруку на крај низа,

2) У случају да није већ заказан за извршавање, означава да је спреман за извршавање,
3) Специјална компонента звана *Планер* узима тог актера и почиње да га извршава,
4) Актер *узима* поруку са почетка реда,
5) Врши модификовање унутрашњег стања и шаље поруку другом актеру,
6) Након тога *Планер* отказује актера [4].

Како бисмо постигли овакав начин рада сваки актер има *сандуче* (ред где се поруке стављају), *понашање* (стање актера), *поруке* (делови података који репрезентују сигнал), *извршно окружење* (машинерија која узима актера код кога постоје поруке на које може да реагује и позива кôд који обрађује поруке) и *адресу.* Овако изграђен актер може решити изазове као што је направити апликацију високих перформанси и са конкурентношћу, како решавати грешке у срединама са више нити, тако и заштитити апликацију од замки које носи конкурентност.

Развојно окружење *Akka .NET* поред основног градивног елемента апликације који представља актера пружа још много погодности да се развије један систем какав бисмо желели да буде *BANKX22.* Једна од погодности је и систем рутирања који је имплементиран у овом развојном окружењу. *Рутери* су специјални актери чији је главни циљ да пристиглу поруку проследе једном или више других актера на што ефикаснији начин. Постоји доста различитих типова рутера имплементираних у овом развојном окружењу. Њиховим коришћењем се значајно повећава пропусност система као и брзина обраде и добијања одговора. Рутер може да исту поруку пошаље већем броју актера, лоцираних на различитим машинама, и рецимо да чека док најбржи не пошаље одговор. На тај начин и ако откаже нека машина, одговор може стићи од актера лоцираног на машини која је функционална.

Веома битна погодност која је имплементирана у развојном окружењеу *Akka .NET* јесте транспарентно слање порука између актера на различитим физичким локацијама. Ово је остварено *Akka.Remoting* и *Akka.Clustering* модулима. Ови модули омогућавају да се ништа у имплементацији кода не мења, ако желимо да комуницирамо са актерима на другим системима. Довољно је правилно конфигурисати систем. *Akka.Clustering* је модул који се надограђује на *Akka.Remoting* и који омогућава поделу система по групама који се зову кластери. *Akka.Clustering* има уграђене механизме откривања отказа неког система унутар кластера, као и механизам детектовања ком актеру унутар неког система треба да стигне порука која је послата ван тог кластера. Чињеница која значајно олакшава употребу ових модула је та да је већина ствари могућа конфигурацијом. Тако је могуће преконфигурисати начин комуникације и начин рада самог кластера у зависности од окружења у којем се апликација налази као и у зависности од тренутне оптерећености апликације. Употребом ових модула наша апликација добија могућност аутоматског скалирања у случајевима смањене, односно повећане оптерећености.

Многи актери које имплементира систем онлајн банкарства *BANKX22* (или било који други систем који користи развојно окружење *Akka .NET*) мора имати могућност чувања свог стања у случајевима отказа. Уобичајно се стање актера чува у меморији. У случајвима престанка рада овако чувано стање се губи и након поновног покретања актер опет креће од иницијалног стања. У неким случајвима је ово недопустиво понашање.

Развојно окружење *Akka .NET* нам пружа модул *Akka.Presistence* који се бави чувањем стања техником која се назива ***извор догађаја*** (*енг. event sourcing* описаног у поглављу 3.8 Обрасци управљања и чувања стања). Овим приступом се сваки догађај који се деси у актеру записује у одговарајуће складиште, а када актер жели да поврати жељено стање догађаји се из базе читају секвенцијално и долази до реконструкције стања поновним извршавањем догађаја [17]. Овакав приступ користе популарани програми за контроле верзија као што је Гит. Предности оваквог начина чувања података је што се избегава проблем *неподударања објектно релационе импеденце*[11] и олакшавају способности провере и праћења који су поготово битни у финансијским системима где је битна историја и порекло трансакција.

У случају да неки актер чува огромну количину догађаја може се начинити временски снимак (*енг. snapshot*) где се након одређеног броја догађаја чува само стање актера па приликом реконструкције стања почиње се од снимљеног стања, а не од иницијалног. Тиме се могу повећати перформансе система. Детаљнији приказ употребе развојног окружења *Akka .NET* ће бити описан у поглављу Имплементација апликације.

## 4.5 .Net Web API интерфејс за програмирање апликација

Приликом пројектовања система онлајн банкарства *BANKX22* дошло се до питања комуникације између корисничког дела система који се налази у прегледачу и серверског дела који је базиран на моделу актера. Природно решење би било да се комуникација обавља путем *HTTP* протокола неким архитектуалним стилом као што је *REST* (*енг. REST-Representational state transfer*) [18]. Проблем је што модел актера на високом нивоу апстракције зна да комуницира само путем порука које се не придржавају *HTTP* протокола и *REST* смерница. Из тог разлога је као мост између клијентског дела и развојног окружења *Akka .NET* постављено развојно окружење које се зове *.NET Web API* интерфејс за програмирање апликација (*енг. Web API .NET Core*) написан у C# програмском језику [6].

Када клијент пошаље захтев за неким ресурсом, тај захтев пролази кроз неки од комуникационих канала који се зове крајња тачка (*енг. endpoint*). Крајње тачке су имплементиране у веб интерфејсу за програмирање апликација путем специјалних класа које се зову контролери. У сваком контролеру се налазе методи који се називају акције и који представљају крајње тачке [5]. Када захтев стигне развојно окружење зна коју од акција да позове уз помоћ система за рутирање. Постоји конфигурациони део развојног оркужења где се врши конфигурација система за рутирање и бројних других подешавања. На овом развојном окружењу сва конфигурисања и подешавања се остварују ***принципом уметања зависности***[12]. На тај начин можемо извршити уметање и конфигурисање система актера развојног окружења *Akka .NET*. Пошто смо овим приступом омогућили повезивање веб

[11] Неподударање објектно релационе импеденце (енг. object-relational impedance mismatch) – објекти у меморији се не пресликавају директно у табеле релационе базе [16].

[12] Принцип уметања зависности (енг. dependency injection principle) – техника у којој један објекат прима друге објекте од којих зависи и који представља једну од форми технике која се назива инверзија контроле [19].

интерфејса за програмирање апликација и система актера, потребно је још да се захтев који је пристигао са корисничког дела проследи систему актера. Већ је напоменуто да ће захтев обрађивати акција одговарајућег контролера, па је могуће и врло једноставно да та акција пошаље поруку актеру из система који је задужен за сервисирање захтева.

Уколико дође до повећане оптерећености система услед повећаног броја захтева, треба применити технику скалирања. У таквим ситуацијама се показало да је боље и изводљивије применити хоризонтално скалирање, односно креирање више инстанци сервера. Проблем који се ту јавља је да ће сваки нови сервер додати и нову инстанцу система актера које су међусобно независне. Срећом, применом *Akka.Cluster* модула из развојног окружења *Akka .NET* можемо комбиновати више система актера заједно. Добра пракса је да се постави један централни систем актера који је задужен за најзахтевније процесе, а да други системи актера који настају скалирањем буду лаке категорије (понашају се као нека врста чувара (*енг. proxy*)) и да порслеђују захтеве централном систему.

# 5 Архитектура система

## 5.1 Опис система

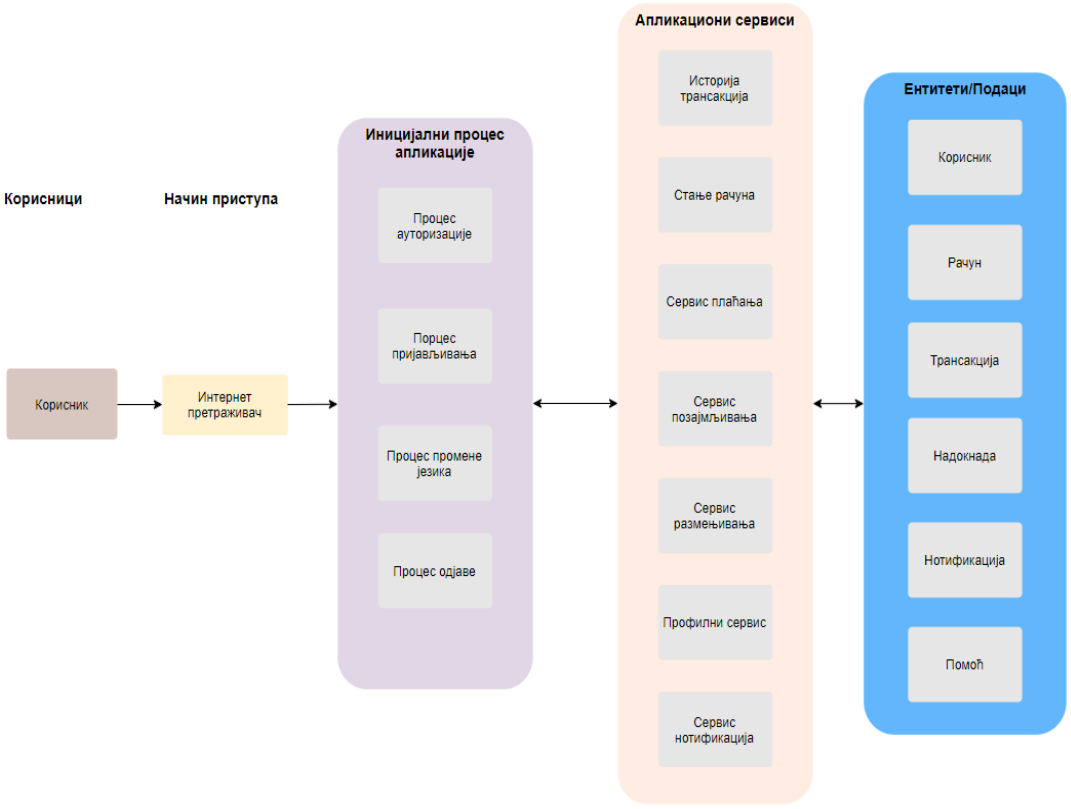
Систем је намењена корисницима који желе да обављају услуге банкарства путем мреже. Систем је отвореног кода и доступан је на *GitHub* репозиторијуму на адреси https://github.com/DragutinIlic88/Master.git. Корисник има могућност регистрације у случају да апликацију користи први пут, односно пријављивања уколико је већ раније извршио пријаву.

Након пријављивања кориснику се нуде могућности прегледа тренутног стања на свим рачунима које поседује, приказ историје свих трансакција које је обавио на неком од рачуна, могућност вршења жељене уплате, конверзије средстава из једне валуте у другу, као и могућност узимања кредита. Поред ових основних сервиса систем шаље разне нотификационе поруке кориснику које се тичу промоција и погодности које банка нуди као и обавештења и упозорења о неплаћеним обавезама. Сваки корисник има приступ свом профилу на коме може поставити, односно изменити информације које се тичу адресе становања, корисничког имена, слике профила, броја телефона и имејла. Уколико се јави потреба да се потражи помоћ приликом обављања неког сервиса или пријављивања неке грешке постоји секција где се може видети контакт телефон и имејл службе.

Архитекура апликације прати принципе дистрибуираног реактивног пројектовања система омогућавајући кориснику добру одзивност, толеранцију на грешке и кварове који се могу јавити, добру еластичност и скалабилност апликације у случају повећане, односно смањене употребе апликације. Више о поменутим својствима можете прочитати у поглављу Манифест реактивног програмирања. Ако се деси да неки од сервиса није доступан, а није кључан за тренутну жељену услугу, корисник неће бити ометен у обављању те услуге. Рецимо ако је из неког разлога недоступан сервис профила корисника или сервис који испоручује нотификације, корисник ће и даље моћи да изврши плаћање, преглед трансакција, конверзију валута или подизање кредита. Може се наравно десити да дође до отказивања оног сервиса који би корисник хтео да искористи. Тада систем може преусмерити извршавање на другу реплику сервиса која се може налазити како на истој тако и на другој машини и тиме омогућити несметано обављање жељене услуге.

Према описаним сервисима, односно посматрајући бизнис логику апликације, природно се намећу неколико битних ентитета који ће се користити. Први битан ентитет је корисник. Ентитет корисника ће бити креиран првим пријављивањем. Том приликом ће садржати све податке који су били неопходни да буду унети приликом процеса пријаве и мора бити сачуван у бази. Одређени подаци садржани у њему ће бити коришћени приликом поступка ауторизације, односно провере могућности корисника да ли је могуће да изврши одговарајућу акцију. Неки подаци могу бити измењени путем профилног сервиса. Рецимо уколико је корисник променио број мобилног или адресу становања, тај податак би требало

изменити ради валидности. Следећи битан ентитет је рачун. Он се провлачи кроз све сервисе. Рецимо уколико се проверава баланс, потребно је изабрати рачун са кога желимо да проверимо стање. Ако се врши уплата, конверзија валута или се жели дићи кредит потребно је одабрати рачун са кога ће се скинути средства односно на коме ће лећи средства. Неизоставан ентитет сваког финансијског система представља трансакција. Трансакција се користи у сервисима прегледа историје трансакција где се могу видети детаљи свих обављених трансакција, приликом конверзије валута, плаћања и подизања кредита где се формирају та три типа трансакција. Приликом размене новца се користи ентитет надокнаде. Он садржи курсну листу за тај датум, односно провизију коју ће банка наплатити од корисника. Још два ентитета која се користе у апликацији су нотификација и помоћ који се користе у нотификационом сервису и пружају детаље поруке која је стигла, односно контакт информације сервисне службе. Описани ентитети и процесни ток се могу видети на Слици 5.1.

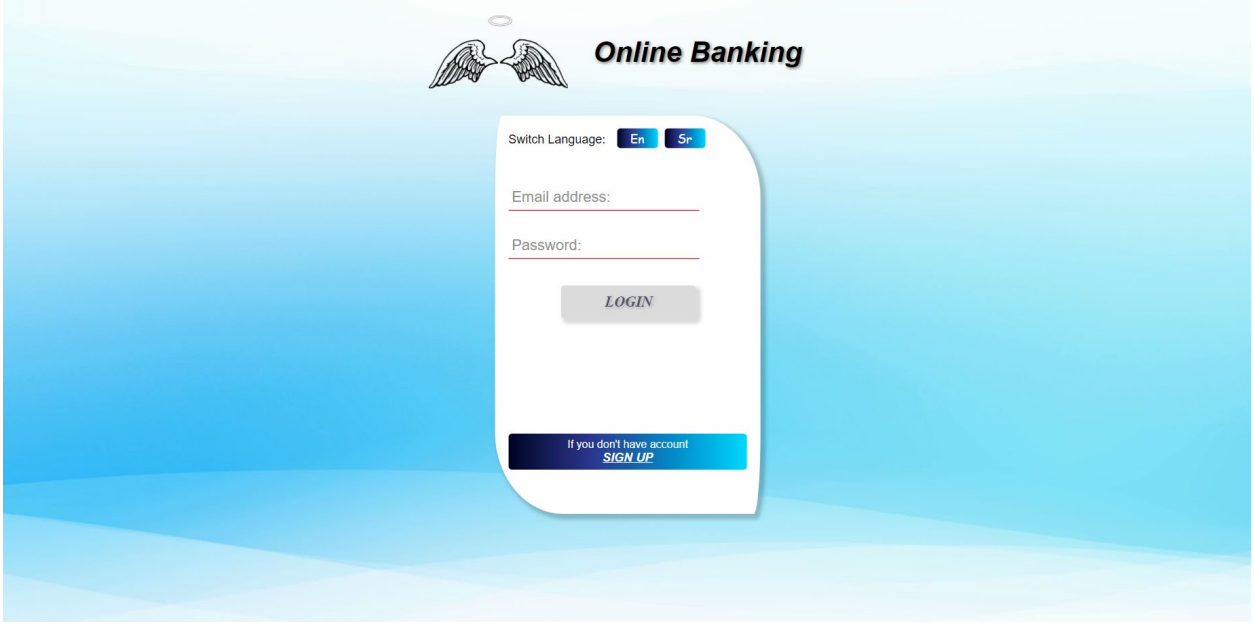



*Слика 5.1 Приказ бизнис архитектуре и порцеса приступа онлајн банкарском систему BANKX22*

## 5.2 Клијентска страна

### 5.2.1 Страна за пријављивање

Када корисник оде на адресу где је постављена апликација биће му приказана страна за пријављивање (Слика 5.2 Страница за пријављивање).




*Слика 5.2 Страница за пријављивање*

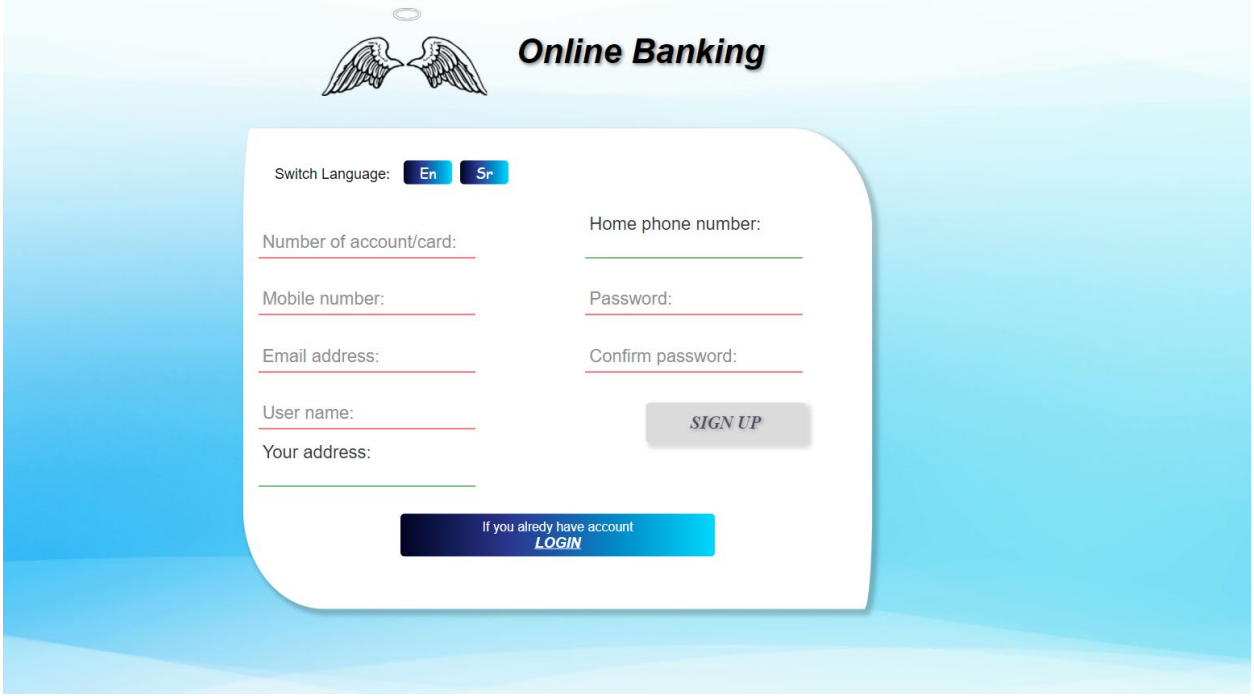
На овој страници корисник може извршити пријаву уношењем имејл адресе и шифре. Одавде корисник може такође извршити избор језика притиском на једну од понуђених опција. Тренутно је подржан српски и енглески језик, али апликација је пројектована тако да се доста једноставно могу додати нови језици. Довољно је додати нови *JSON* фајл у фолдер са именом језици (*енг. languages).*

### 5.2.2 Страна за регистровање

Уколико корисник не поседује налог на располагању му је опција да креира нови. Притиском на ту опцију приказана му је формулар са подацима које је потребно да унесе како би регистрација била успешна (Слика 5.3 Страница за регистрацију). Обавезно поље

је број рачуна или картице. Корисник мора имати постојећи рачун или картицу већ регистровану у банци како би могао да користи систем онлајн банкарских сервиса *BANKX22*.

Остала обавезна поља су број мобилног, имејл адреса, корисничко име, шифра и поље за потврду шифре. Поред обавезних поља корисник има опцију да унесе и адресу становања и број кућног телефона. Ове информације ће бити сачуване као део профила корисника и на основу њих оператер или лични саветник банке ће имати алтернативни начин како може контактирати корисника или где му може послати потребна средства. Уколико је корисник завршио регистрацију успешно, биће враћен на страницу за пријављивање.

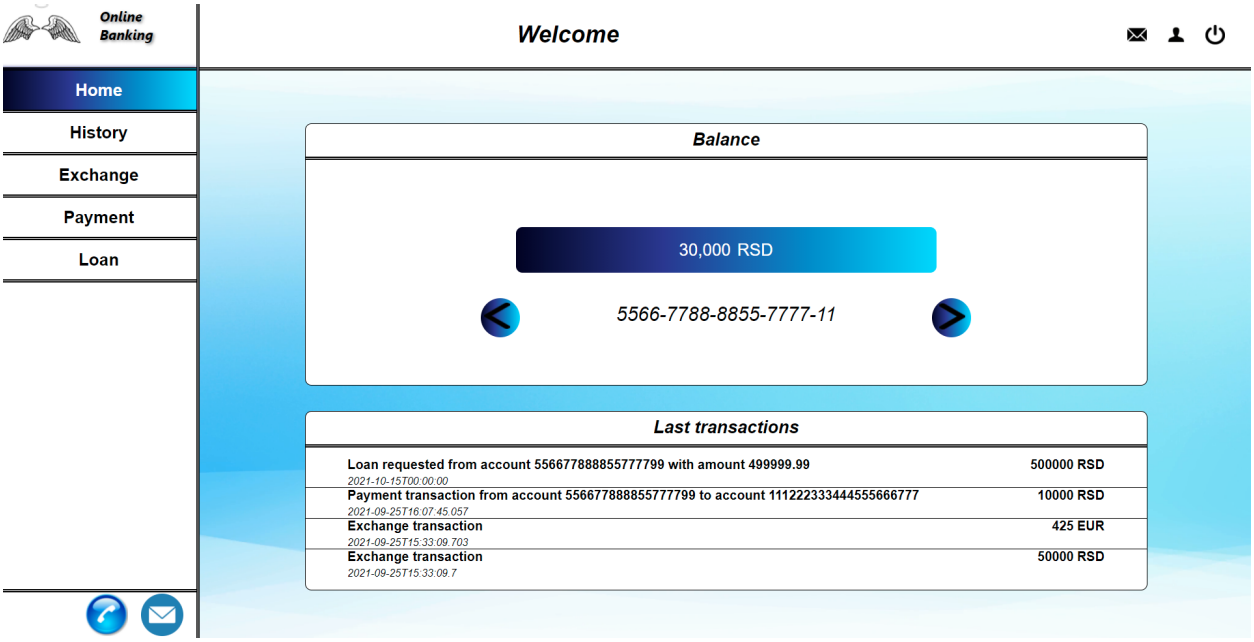



*Слика 5.3 Страница за регистрацију*

## 5.2.3 Почетна страна

Када се корисник успешно пријави на систем онлајн банкарства *BANKX22* прво му је приказана почетна страница (Слика 5.4 Почетна страна). Са леве стране екрана постоји навигација помоћу које корисник може одабрати сервис који жели да изврши на систему. Навигација је глобална компонента која је видљива у сваком сервису и корисник може да промени сервис у било ком тренутку. У доњем левом углу су дате опције којима корисник има могућност да види информације на који број може да добије помоћ оператера или на коју адресу може да пошаље имејл како би добио помоћ.

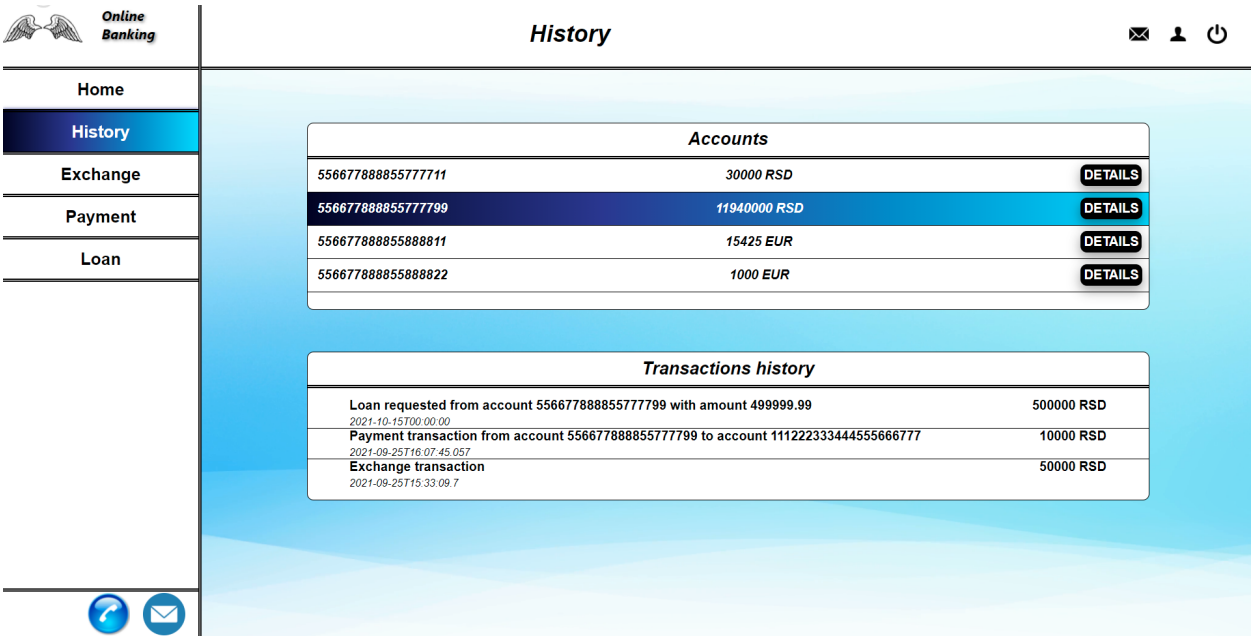
На почетној страни корисник има информације о стању на сваком од рачуна које поседује и може видети списак последњих трансакција које је извршио на систему.




*Слика 5.4 Почетна страна*

### 5.2.4 Страна историје

Уколико корисник изабере опцију „Историја“ из навигације биће му приказана листа рачуна које поседује (Слика 5.5 Историја трансакција). Ако притисне опцију „Детаљи“ која се налази на сваком од рачуна кориснику се отвара модал са детаљима рачуна. Ту се могу видети информације о броју рачуна, типу рачуна, датуму отварања рачуна, детаљима рачуна као и тренутним расположивим средствима на том рачуну. Притиском на било који рачун из листе добија се листа свих извршених трансакција за тај рачун у табели испод. За сваку трансакцију се могу видети њени детаљи у које спадају име трансакције, детаљи трансакције, тип трансакције, време обраде трансакције, статус у коме је трансакција тренутно као и износ који је обрађен трансакцијом.

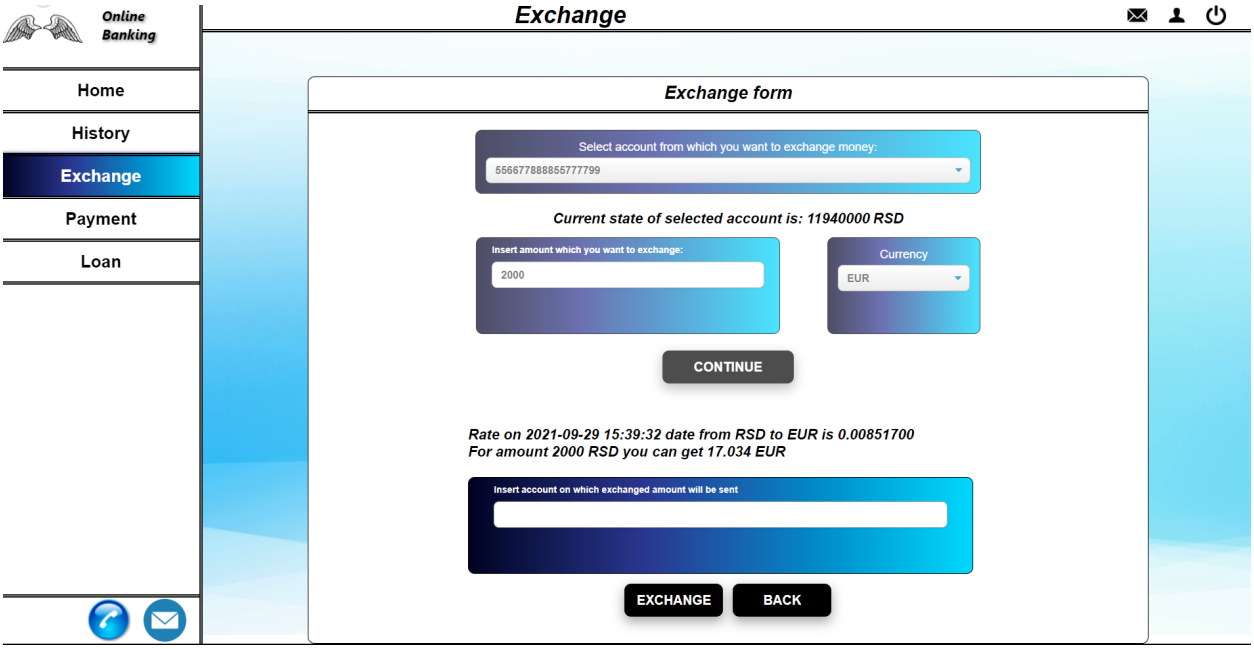



*Слика 5.5 Историја трансакција*

### 5.2.5 Страна за размену валута

Притиском на опцију „Мењачница“ у навигацији корисник има могућност да изврши пребацивање средстава са рачуна који је у одређеној валути на рачун са неком другом валутом. На самом почетку у сервису „Мењачница“ се види приказ формулара на коме се може изабрати рачун са кога ће бити скинута средства за размену, поље у коме се може унети количина средстава која ће бити замењена као и селектор где се може изабрати циљна валута размене. Добијена листа валута у коју корисник може да изврши замену ће бити променљива за сваког корисника у зависности од нивоа привилегија као и од валута самих рачуна које корисник поседује.

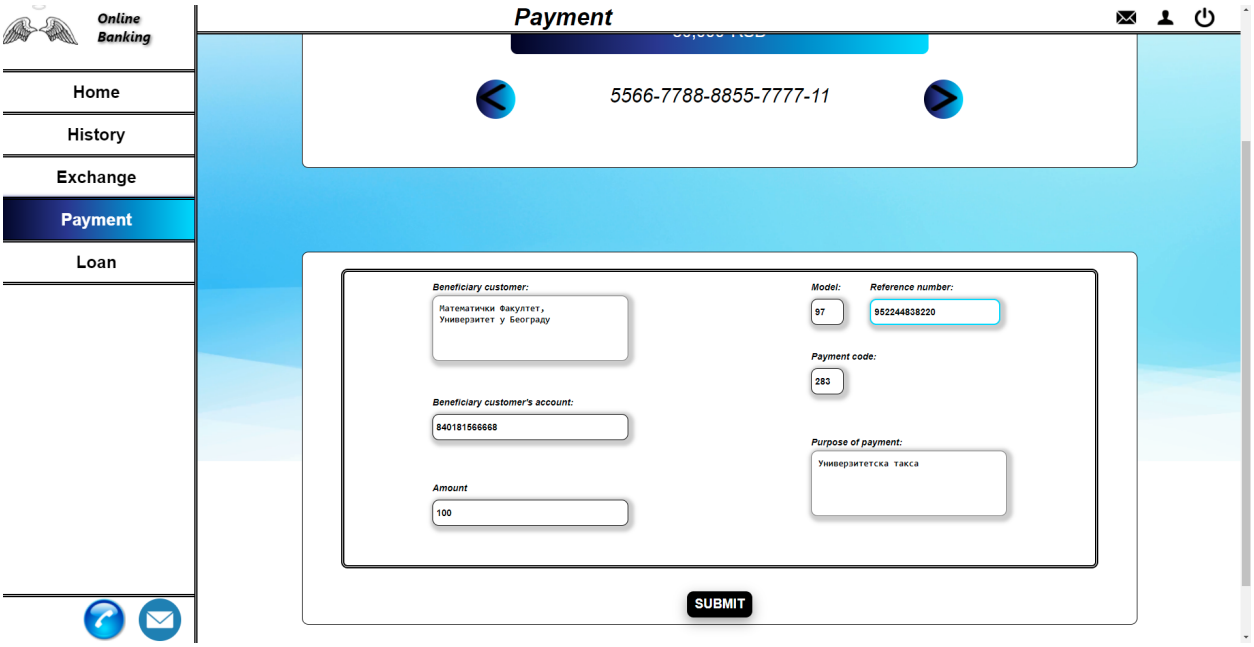
Након уноса свих неопходних информација и притиском на дугме „Настави“ добија се информација о курсу за тадашњи датум из валуте рачуна у циљну валуту као и поље где корисник може унети рачун на коме ће бити пребачена размењена средства. Уписани рачун мора имати валуту коју је корисник изабрао, иначе размена неће бити успешна (Слика 5.6 Страна "Мењачница"). Уколико је размена извршена успешно биће приказан модал детаља о размени као што су рачун са кога су скинута средства, колчина скинутих средстава, рачун на коме су уплаћена средства, количина средстава у циљној валути као и курс који је коришћен приликом конверзије.




*Слика 5.6 Страна "Мењачница"*

## 5.2.6 Страна за плаћање

Корисник може искористити погодности сервиса „Плаћање“ уколико одабере истоимену опцију из навигације. Прво што корисник треба да уради је да изабере рачун са кога жели да изврши плаћање. Слајдер са листом рачуна је исти као на почетној страни. Потом је потребно да попуну форму која представља уплатницу и потврди плаћање (Слика 5.7 Страна "Плаћање").



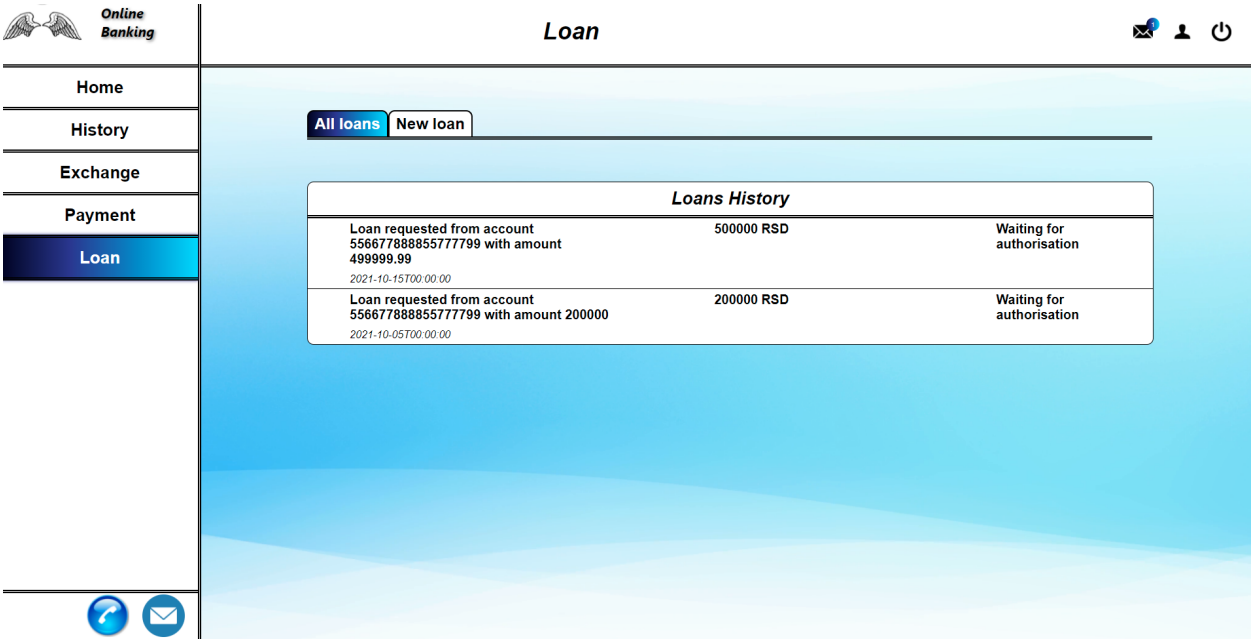

*Слика 5.7 Страна "Плаћање"*

Притиском дугмета „Потврди“ појављује се модал који показује детаље плаћања и кориснику се пружа могућност да прекине трансакцију или да је изврши. Уколико је изврши на екрану се приказује потврда о успешном плаћању а детаљи плаћања се могу видети у секцији „Историја“.

### 5.2.7 Страна за узмање кредита

Секција „Позајмица“ којој се приступа притиском на опцију из навигације има два приказа.

Први приказ је листа свих текућих позајмица које је корисник склопио са банком. Кликом на неку од ставки из листе могу се добити детаљи те позајмице (Слика 5.8 Страница са приказом свих текућих позајмица).

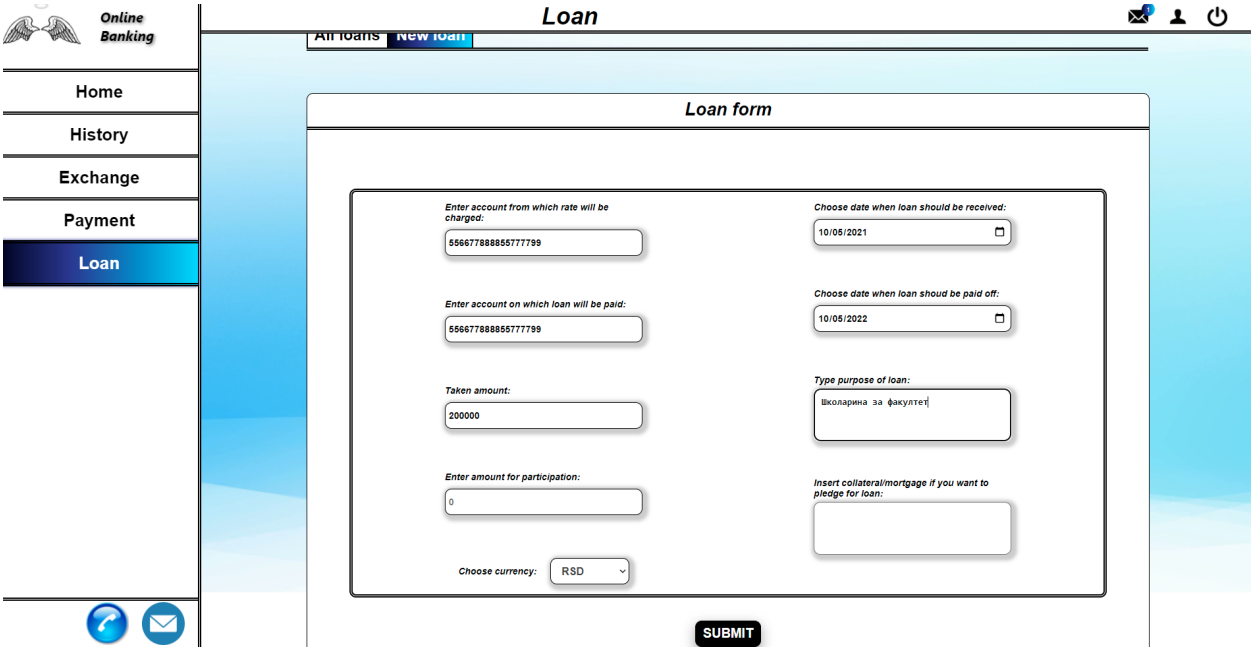



*Слика 5.8 Страница са приказом свих текућих позајмица*

Други приказ је формулар помоћу којег корисник може да поднесе захтев за позајмицу. Потребне информације су: рачун са кога би се скинула средства за отплату рате позајмице, рачун на коме би био уплаћен износ позајмице, износ позајмице, валута у којој ће позајмица бити исплаћена, опционо партиципација помоћу које ће бити смањен износ камате и брзина отплате, датум када би позајмица била уплаћена, датум до када би корисник желео да отплати позајмицу, сврха узимања позајмице и опционо хипотека коју корисник може да приложи.

Након уноса података и притиском на дугме „Потврди“ на екрану се приказује модал са детаљима информација позајмице. Корисник има могућност да прекине са процесом или да потврди захтев. Уколико потврди захтев корисник ће бити пребачен на приказ са листом позајмица на коме ће моћи да види да је захтев поднет и да се чека на ауторизацију. Такође

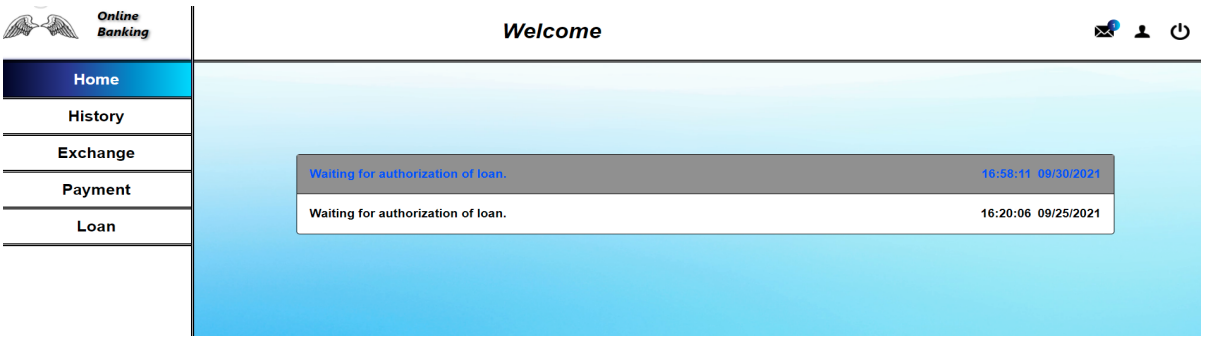
кориснику ће бити послата и нотификација о подношењу захтева и секција са нотификацијама у горњем десном углу екрана ће означити да постоји нотификација која није прочитана.




*Слика 5.9 Страница са формом за позајмицу*

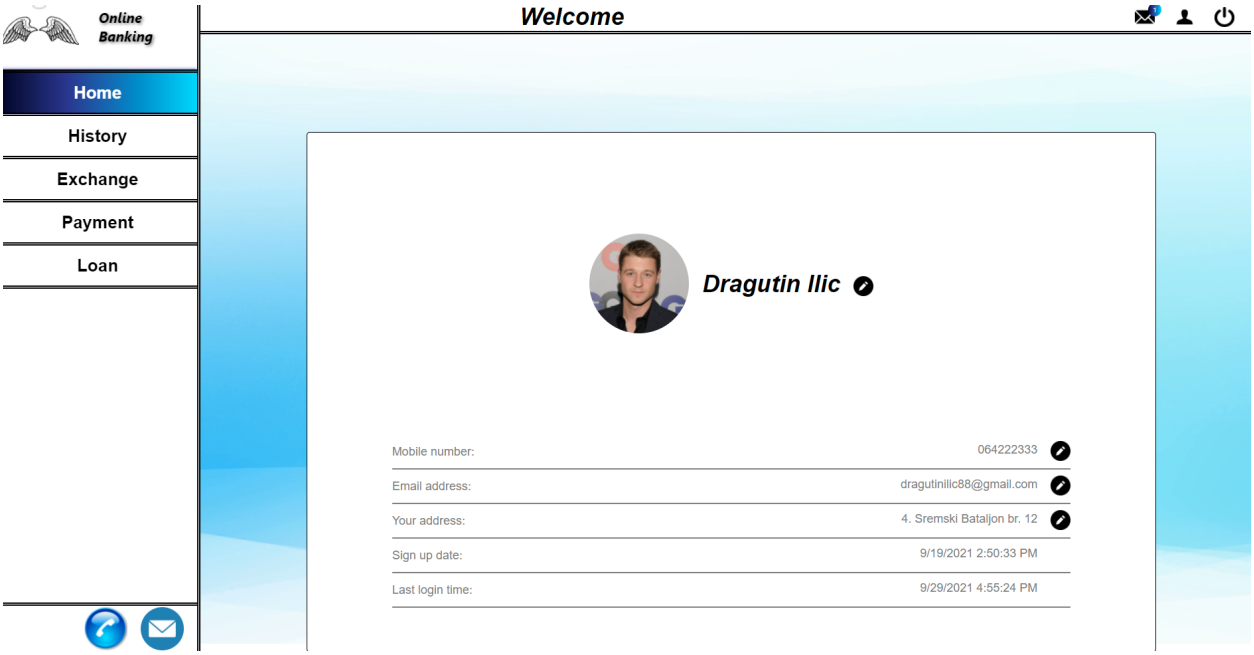
## 5.2.8 Опције у заглављу

У заглављу екрана са десне стране постоје три опције које корисник може да искористи и то су приказ нотификација, приказ профила корисника и одјава корисника. Страница са нотификацијама приказује листу свих нотификација и приступа јој се ако корисник притисне икону писмо. На икони постији маркер који показује број непрочитаних нотификација уколико их има. Све непрочитане нотификације ће бити затамљене. Притиском на неку од нотификација приказују се детаљи нотификације и опција за њено брисање (Слика 5.10 Страница са приказом листе нотификација).




*Слика 5.10 Страница са приказом листе нотификација*

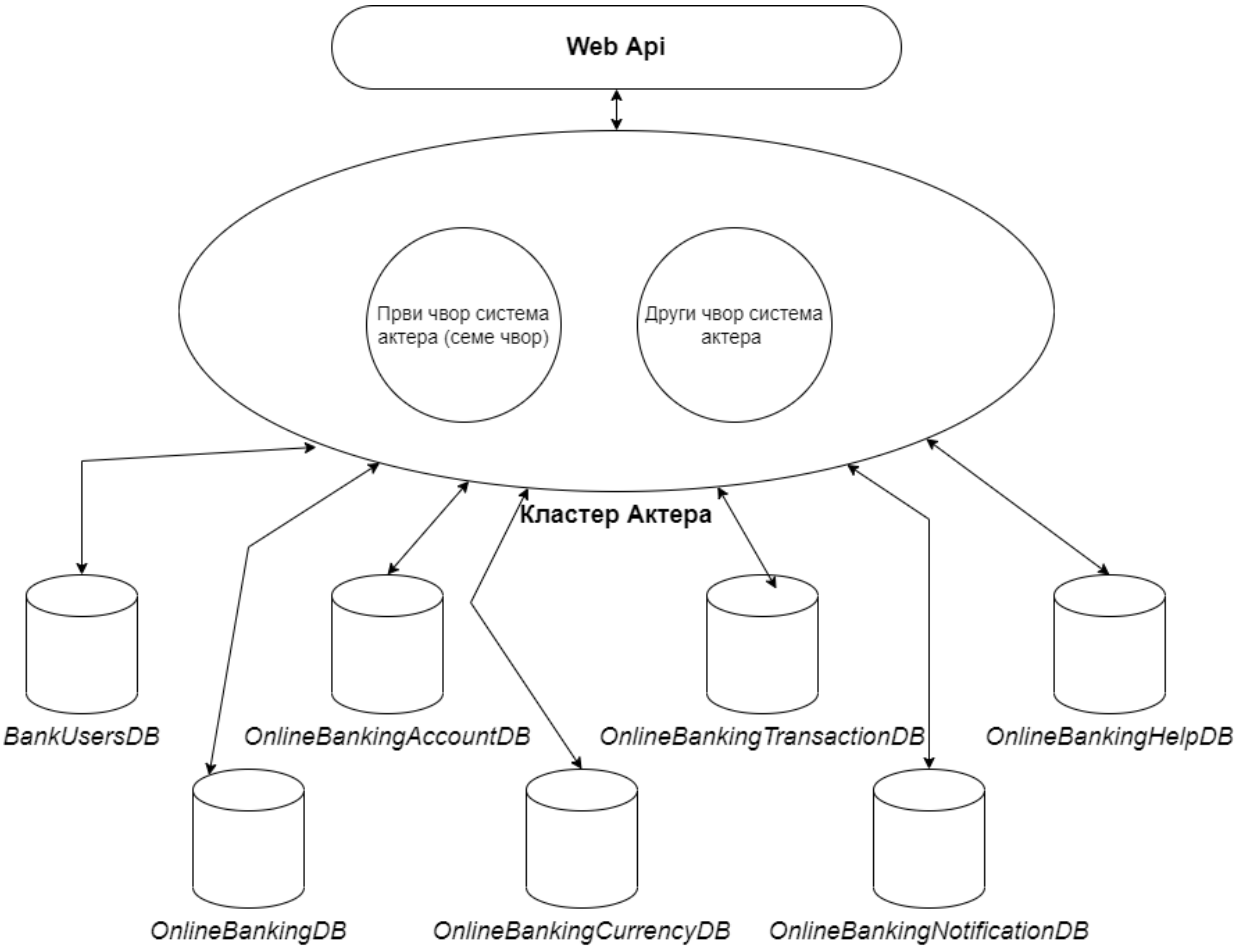
Уколико корисник изабере икону „чикица“ биће пребачен на профилну страницу. На њој се могу наћи подаци о корисничком имену, профилној слици, имејл адреси, мобилним бројем, адресом становања, временом регистровања и задњим временом пријављивања. Све податке осим времена регистровања и пријављивања корисник може ажурирати у било ком тренутку избором иконе „пенкало“ (Слика 5.11 Профилна страна корисника).




*Слика 5.11 Профилна страна корисника*

Последња опција која је понуђена кориснику је опција за одјаву са система. Она се извршава притиском иконе „гашење“. Одјавом се корисник пребацује поново на страну за пријављивање.

## 5.3 Серверска страна




*Слика 5.12 Архитектура серверске стране, висок ниво апстракције*

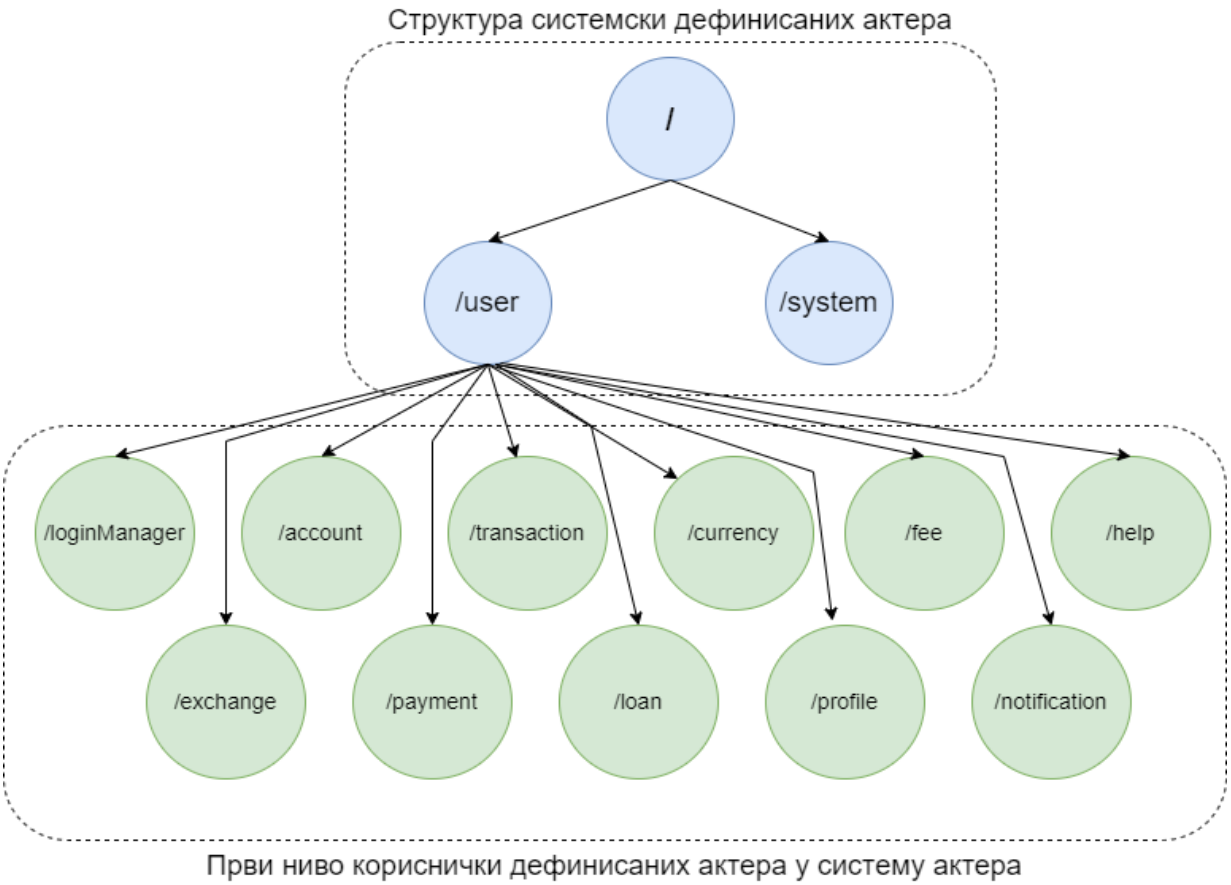
Архитектура серверске стране приказана је на Слици 5.12. За директну комуникацију са клијентским делом апликације се користи *.NET Web API* слој архитектуре. Овај слој се састоји од контролера који представљају крајње тачке (*енг. Endpoint*) којима пристижу HTTP захтеви са клијента. Контролери врше проверавање исправности захтева и података који су послати као део тог захтева и уколико захтев задовољава све критеријуме провере он се конвертује у поруку и као такав шаље одговарајућем актеру који припада систему актера. Друга улога слоја *.NET Web API* је да изврши конфигурацију свих сервиса који се користе у систему (више о детаљима конфигурације се може прочитати у поглављу 4.5.Net Web API интерфејс за програмирање апликација).

*.NET Web API* комуницира са кластерима система актера. Кластер система актера представља мрежу која је отпорна на кварове, еластична, децентрализована и коју чине системи актера који немају ниједну тачку неуспеха нити уско грло. Онлајн систем

банкарства *BANKX22* поседује кластер коју чине две реплике система који се у контексту кластера означавају као чворови.

Први чвор који ће припадати кластеру се назива семе чвор (*енг. Seed Node*) и кластер не може бити оформљен ако приликом његове конфигурације није наведен бар један семе чвор. Једино у комуникацији са овим чвором је могуће придруживање и уклањање других чворова [1]. Комуникација између чворова се врши протоколом који се назива оговарање (*енг. Gossip*). Помоћу овог протокола се врши придруживање и уклањање чланова као и ажурирање стања сваког члана о доступности осталих чланова.

Систем онлајн банкарства *BANKX22* поред чвора семена садржи још један чвор који приликом покретања система у својој конфигурацији садржи адресу на којој се налази семе чвор и помоћу те адресе га контактира и тражи да се придружи групи. Семе чвор који мора бити покренут пре слања захтева од другог чвора након примања захтева означава тај чвор као део кластера и уколико би се у кластеру налазили још неки чворови послао би информације о њима. Пошто то овде није случај формиран је кластер од два члана која комуницира протоколом оговарања. На тај начин се постиже расподела захтева у случају преоптерећености неког од чворова, као и боља отпорност у случају да дође до престанка рада неког од чворова.



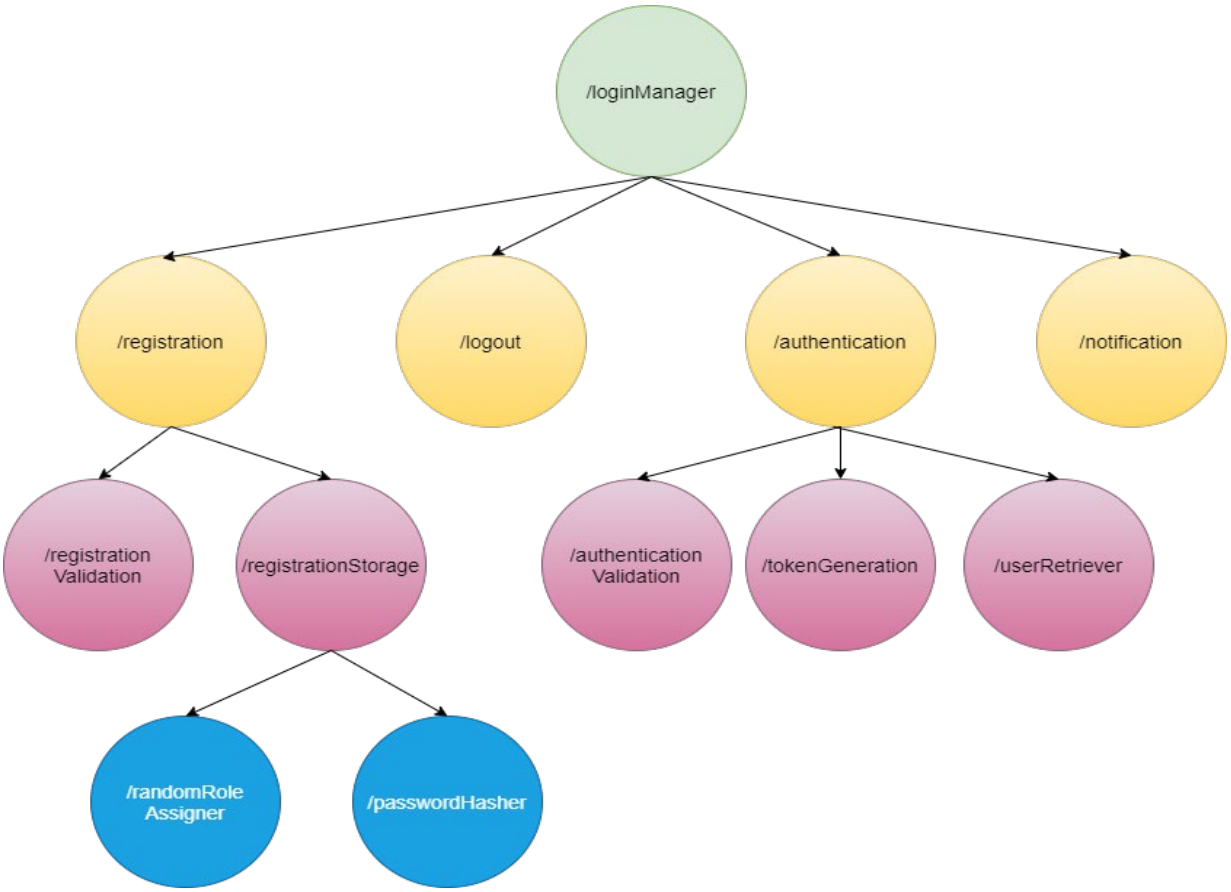

*Слика 5.13 Приказ једне реплике система актера са првим нивоом актера дефинисаних од стране корисника*

На Слици 5.13 дат је хијерархијски приказ једне од реплика система актера. Приликом креирања система актера, развојно окружење *Akka* креира почетну структуру актера коју чине:

- „актер чувар корена“ (*енг. root guardian actor*) који на слици има ознаку косе црте; То је једини актер који нема родитеља и представља родитеља свим другим актерима што значи да ће последњи бити прекинут,
- „актер чувар корисника“ (*енг. user guardian actor*) је родитељ свих актера који су креирани од стране корисника развојног окружења *Akka*,
- „актер чувар система“ (*енг. system guardian actor*) осигурава да се систем угаси по одговарајућем поретку и служи за надзор и одржавање других системских актера који су имплементирани од стране самог развојног окружења [4].

Сви актери који су креирани од стране корисника развојног окружења ће бити креирани као деца актера чувара корисника. На Слици 5.13 су приказани кориснички дефинисани актери који представљају врх хијерархије и чија је основна намена да примају поруке од *.NET Web API* слоја, врше одговарајућа процесирања и шаљу поруке деци актерима на даљу обраду.Сваки од ових актера представља улаз у посебан сервис система и сваки је имплеметиран коришћењем рутера који су свесни кластера система актера и који транспаренто праве одговарајући број инстанци тог актера на сваком чвору групе и шаљу поруке инстанци у зависности од њене оптерећености.

### 5.3.1 Хијерархијски приказ архитектуре сервиса за ауторизацију и аутентификацију корисника

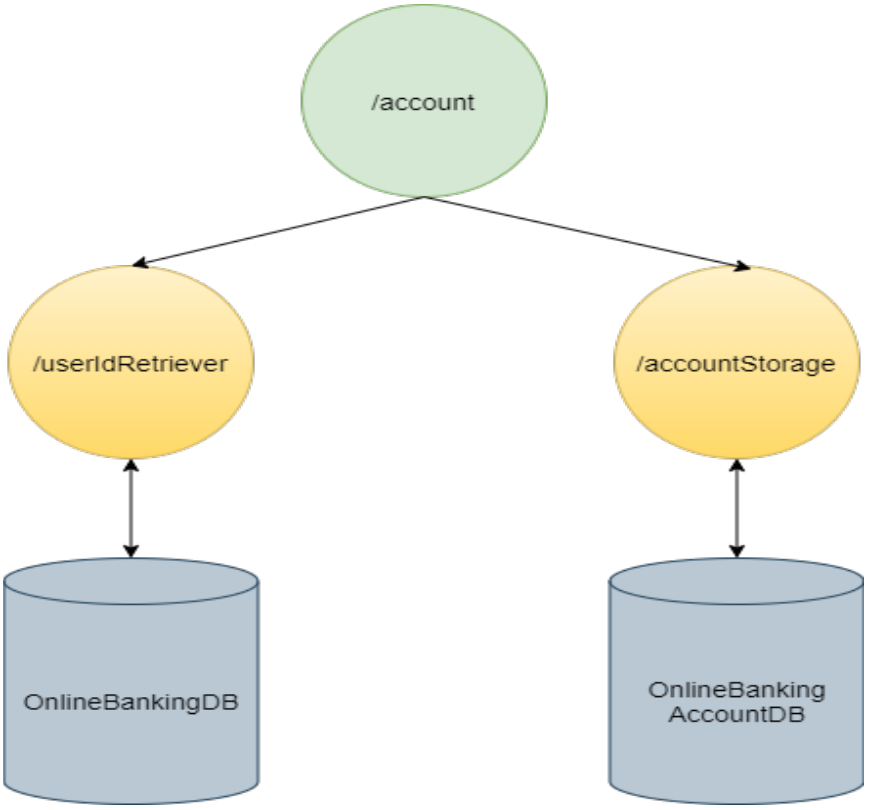



*Слика 5.14 Хијерархија актера који учествују у процесу ауторизације и аутентификације корисника*

На Слици 5.14 дат је хијерахијски приказ актера који учествују у ауторизацији и аутентификацији корисника система онлајн банкарства *BANKX22*. На врху хијерархије се налази актер који је менаџер пријављивања, регистровања и одјављивања корисника. Он прима поруке од различитих акција контролера специјализованог за аутентификацију и ауторизацију и у зависности од типа поруке покреће одговарајући процес тако што шаље поруку неком од деце актера. Видимо да је хијерархија пројектована тако да задовољава Образац једноставне компоненте и Образац језгра грешке. Сложени процес се на сваком нивоу упрошћава, а да се при томе важна стања чувају у актеру ближе корену, тако да сваки актер ради тачно један посао у потпуности. Тако на примеру регистрације постоји посебан актер који се бави одређивањем улога које ће корисници имати у систему, актер који ће хеширати шифре како би се као такве чувале у бази ради сигурности и актер чија је улога да све податке при регистровању успешно сачува у бази. Такође постоји посебан актер који ће се бавити провером исправности података, док његов родитељ врши синхронизацију између потребних операција. Слична хијерахија је успстављена приликом пријаве корисника где се процес пријављивања дели на једноставније операције и свакој операцији

се додељује посебан актер. Тако да постоји посебан актер за валидацију аутентификације, актер за генерисање корисничког токена и актер за читање корисничких информација из базе. Процес одјаве корисника је једноставна операција која захтева једног актера који ће уклонити одређене податке из базе.

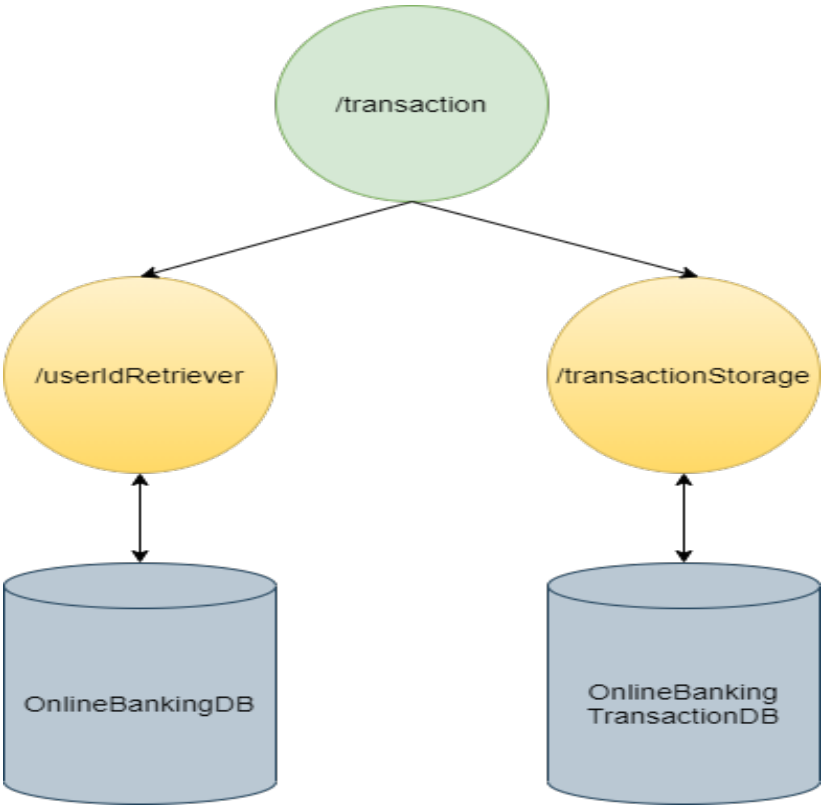
### 5.3.2 Хијерархијски приказ архитектуре сервиса за обраду рачуна




*Слика 5.15 Хијерахијски приказ архитектуре сервиса за добијање рачуна корисника*

Један од основних сервиса који се користи за разне операције система је сервис обраде рачуна (Слици 5.15 Хијерахијски приказ архитектуре сервиса за добијање рачуна корисника). Конторлер слоја *.NET Web API* шаље поруку актеру рачуна како би добио све рачуне који су доступни пријављеном кориснику. Да би рачуни могли да буду прочитани из базе рачуна потребан је идентификациони број корисника. Актер рачуна тако шаље поруку актеру који је задужен да на основу токена пријаве достави идентификацион број корисника. По добијању идентификационог броја, актер рачуна шаље поруку актеру складишту рачуна који дохвата рачуне из базе и шаље поруку са рачунима контролеру. У комуникацији актера коришћени су обрасци самосталне поруке и обрасци тока напред (описани у поглављу 3.6 Обрасци комуникације).

### 5.3.3 Хијерархијски приказ архитектуре сервиса за обраду трансакција



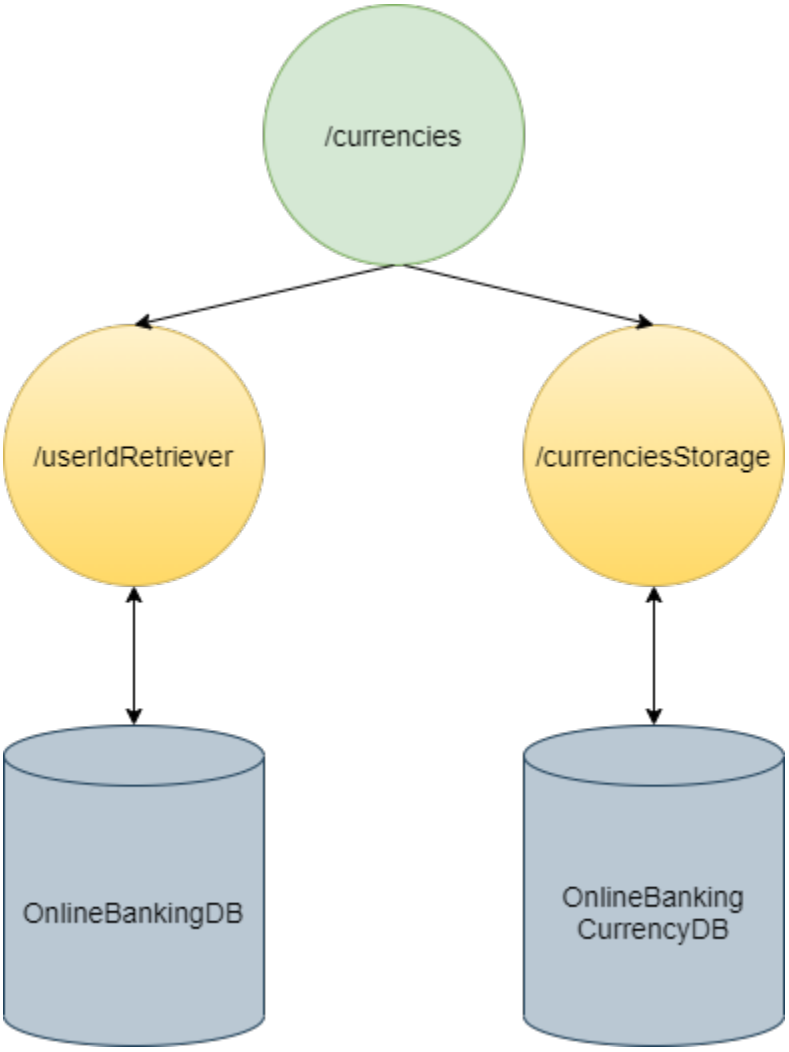

*Слика 5.16 Хијерархијски приказ архитектуре сервиса за обраду трансакција*

Сервис чија архитектура је доста слична сервису обраде рачуна је сервис обраде трансакција (Слика 5.16 Хијерархијски приказ архитектуре сервиса за обраду трансакција). Разлика је та што у сервису обраде трансакција учествује актер складишта транскација који комуницира са базом у којој се налазе све обављене трансакције унутар система. Да би актер прочитао трансакције и послао их конторлеру потребан му је идентификациони број корисника који преузима од актера који их чита из базе корисника онлајн банкарства *BANKX22*.

### 5.3.4 Архитектура сервиса за пребацивање средства из једне валуте у другу

Сервис за добијање валута се користи приликом обављања плаћања, узимања позајмице, као и приликом трансакције размене. Хијерархијски приказ архитектуре сервиса за добијање валута се може видети на Слици 5.17  За сваког корисника постоји одређен

скуп валута који му стоје на располагању у зависности од рачуна које поседује, улоге која му је додељена у систему и привилегија које има.



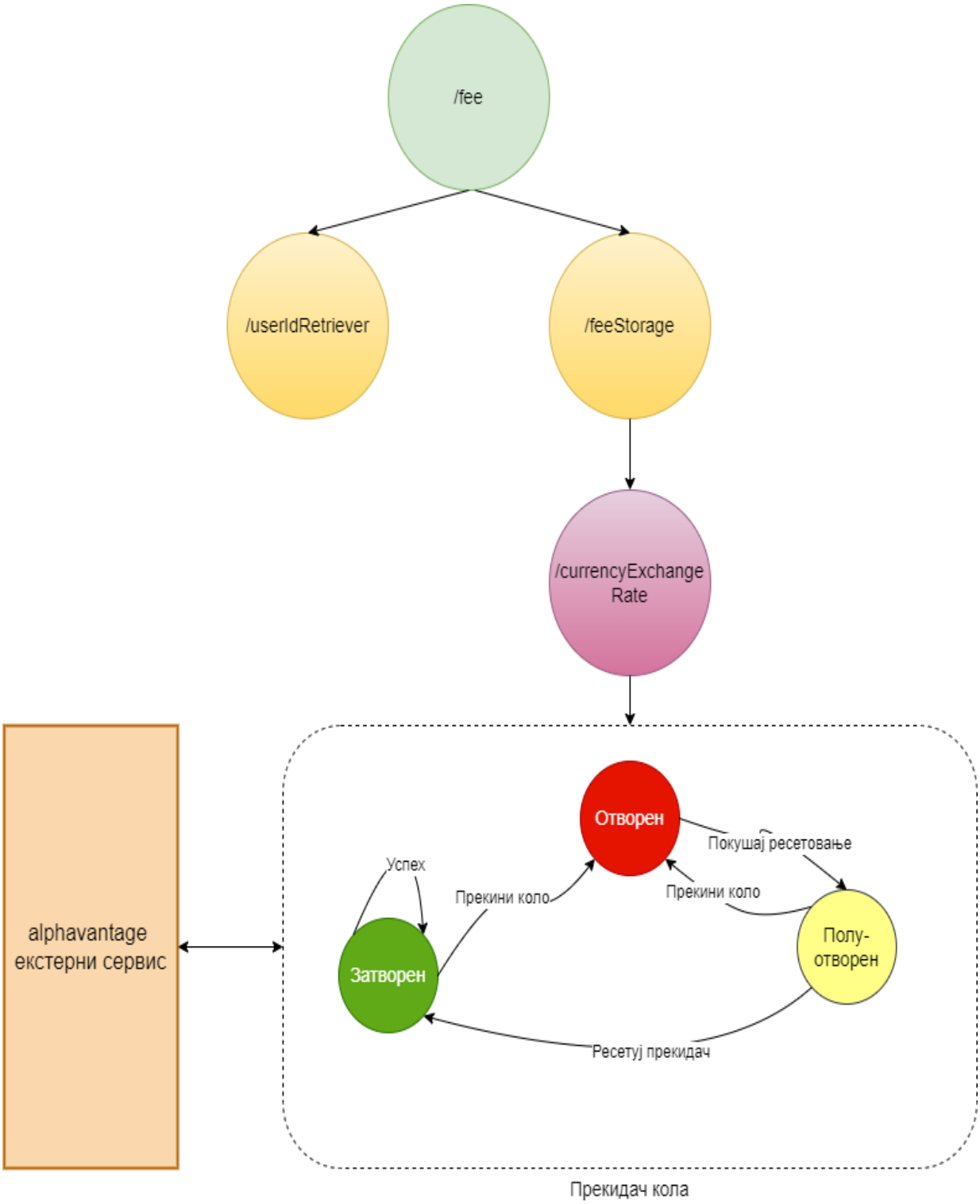

*Слика 5.17 Хијерархијски приказ добијања валута за одговарајућег корисника*

### 5.3.5 Архитектура сервиса за пребацивање средства из једне валуте у другу

Приликом обављања операција које садрже пребацивање средства из једне валуте у другу корисник је у обавези да плати одређену провизију. За одређивање провизије користи се сервис за добијање провизије (Слика 5.18 Хијерархијски приказ архитектуре сервиса за обраду провизија). Један од услова висине провизије зависи од тога које привилегије корисник има. Зато је први корак тог сервиса добијање идентификације корисника. Потом се шаље порука актеру за складиштење провизија да добије провизију конверзије из једне валуте у другу које се шаљу у саставу поруке. Задатак овог актера је да

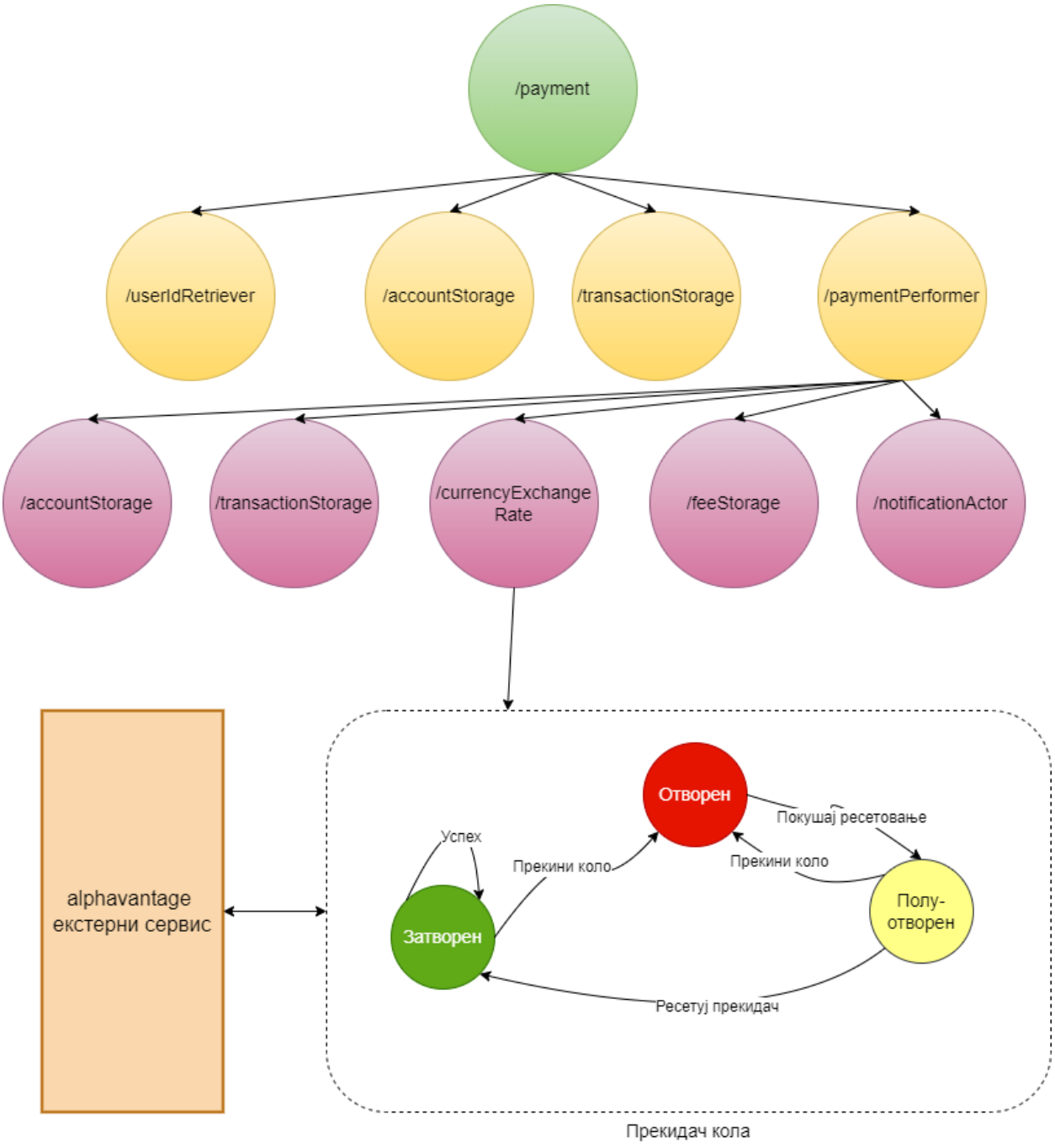
добије провизије од посебног актера који се бави добијањем провизија за размену одговарајућих валута на дан слања захтева, потом да резултат провизије сачува у базу валута и да врати резултат контролеру који је послао захтев.

Актер за добијање провизије комуницира са екстерним системом (*енг. alphavantage*) који поседује информације о односу размене валута на одређени датум [20]. Пошто се ради о екстерном сервису искоришћен је Образац прекидач кола како би се у случају недоступности сервиса спречило беспотребно чекање на одговор тако што коло прелази у отворено стање и одмах шаље поруку да сервис тренутно није доступан. Уколико се деси та ситуација актер за складиштење проверава да ли постоји у бази кеширана информација о односу размена валута и уколико постоји шаље ту информацију као одговор. То је разлог зашто се вршио упис инофрмација о размени у базу након успешног добијања од екстерног сервиса.




*Слика 5.18 Хијерархијски приказ архитектуре сервиса за обраду провизија*

### 5.3.6 Хијерахијски приказ архитектуре сервиса за плаћање




*Слика 5.19 Хијерархиски приказ архитектуре сервиса за плаћање*

Архитектура сервиса за плаћање се може видети на Слици 5.19. Постоје два општа случаја како се одвија ток плаћања. Уколико се плаћање врши са рачуна на рачун који имају

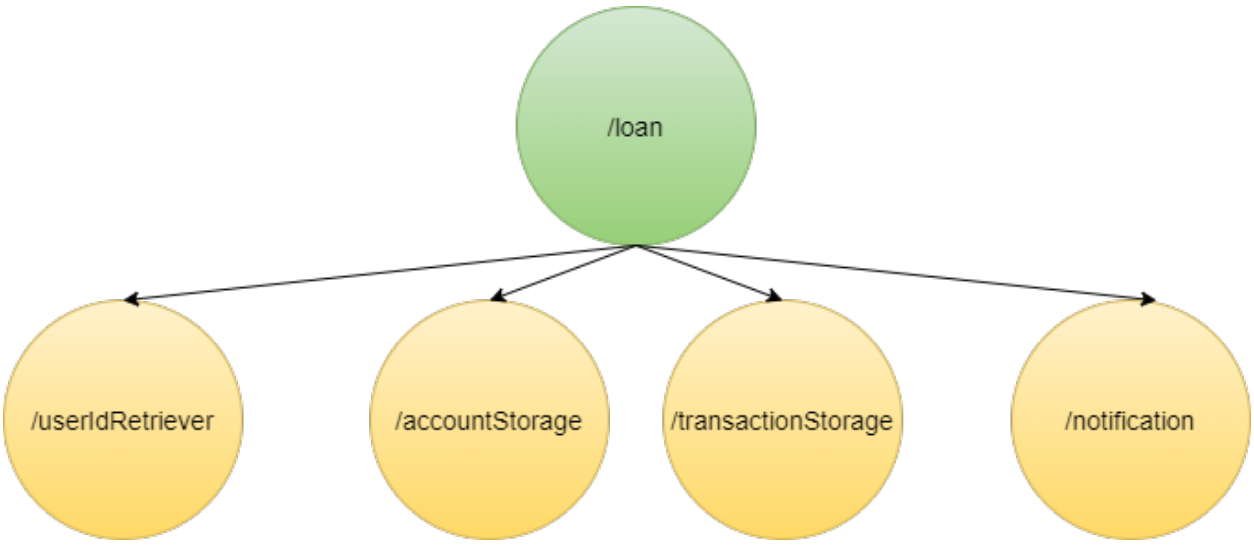
различите валуте онда у процесу се користи и сервис за добијање провизија. Уколико то није случај ток одвијања плаћања је једноставнији.

Као и усваком претходном сервису прво је потребно добити идентификацију корисника. Актер плаћања потом шаље поруку актеру који је складиште рачуна, да добије рачун са кога се врши плаћање, као и рачун на коме се врши уплата уколико припада систему онлајн банкарства *BANKX22*. Средства се скидају са првог рачуна уколико их има довољно на стању и додају се на рачун уплате. Информације о извршеној трансакцији се шаљу актеру складишту трансакција како би могао да ту трансакцију заведе. Порука се шаље и нотификационом сервису како би обавестио корисника путем нотификације о успешном плаћању. Већи број актера учествује у сервису плаћања.

Постоје многи случајеви у којима може доћи до немогућности извршавања плаћања као што су недовољно стање на рачуну са кога се врши уплата или недоступност неког од сервиса које користи сервис плаћања. Због тога је потребна добра координација комуникације актера као и осигуравање атомичности трансакције у сервису који је дистрибуиран. Како бисмо то постигли искоришћен је сага образац (за више информација погледати 3.7 Обрасци контроле тока) приликом пројектовања овог сервиса.

### 5.3.7 Хијерархијски приказ архитектуре сервиса за позајмице

Сервис за позајмице се бави процесирањем нових позајмица од стране корисника као и обрадом историје свих позајмица. Приказ архитектуре актера који су компоненте сервиса позајмице се може видети на Слици 5.20.



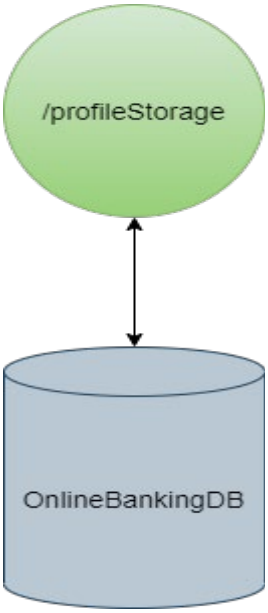

*Слика 5.20 Хијерархијски приказ архитектуре актера која је примењена у сервису позајмице*

У зависности да ли се процесира нова позајмица или се врши ишчитавање постојећих, актер позајмице прелази у одговарајуће стање. Потребно је најпре да се добије идентификациони број корисника па и у овом сервису учесник је актер за добијање идентификације. Актер складишта трансакција се користи да бисмо уписали нову позајмица

у виду трансакције, односно да бисмо прочитали све трансакције које представљају историју позајмице. Актер складишта рачуна се користи како бисмо добили рачуне корисника на коме ће бити уплаћена позајмица, односно са ког рачуна ће бити скидана месечна рата. Када се нова позајмица обради, нотификација ће бити послата корснику па се из тог разлога користи и нотификациони сервис.

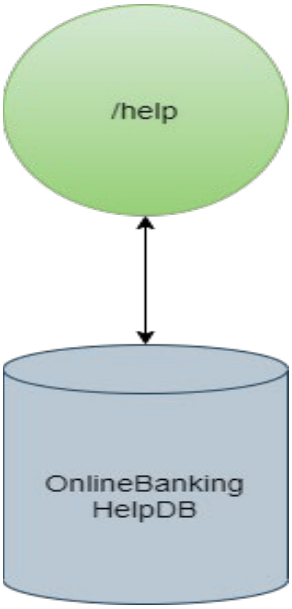
### 5.3.8 Хијерархијски приказ архитектуре профилног сервиса

Профилни сервис се користи за добијање и ажурирање података који се тичу корисника. За пројектовање сервиса профила довољан је актер складишта профила који комуницира са базом онлајн банкарства како би добио или ажурирао информације одређеног корисника *(*Слика 5.21 Архитектура сервиса профила*)*




*Слика 5.21 Архитектура сервиса профила*

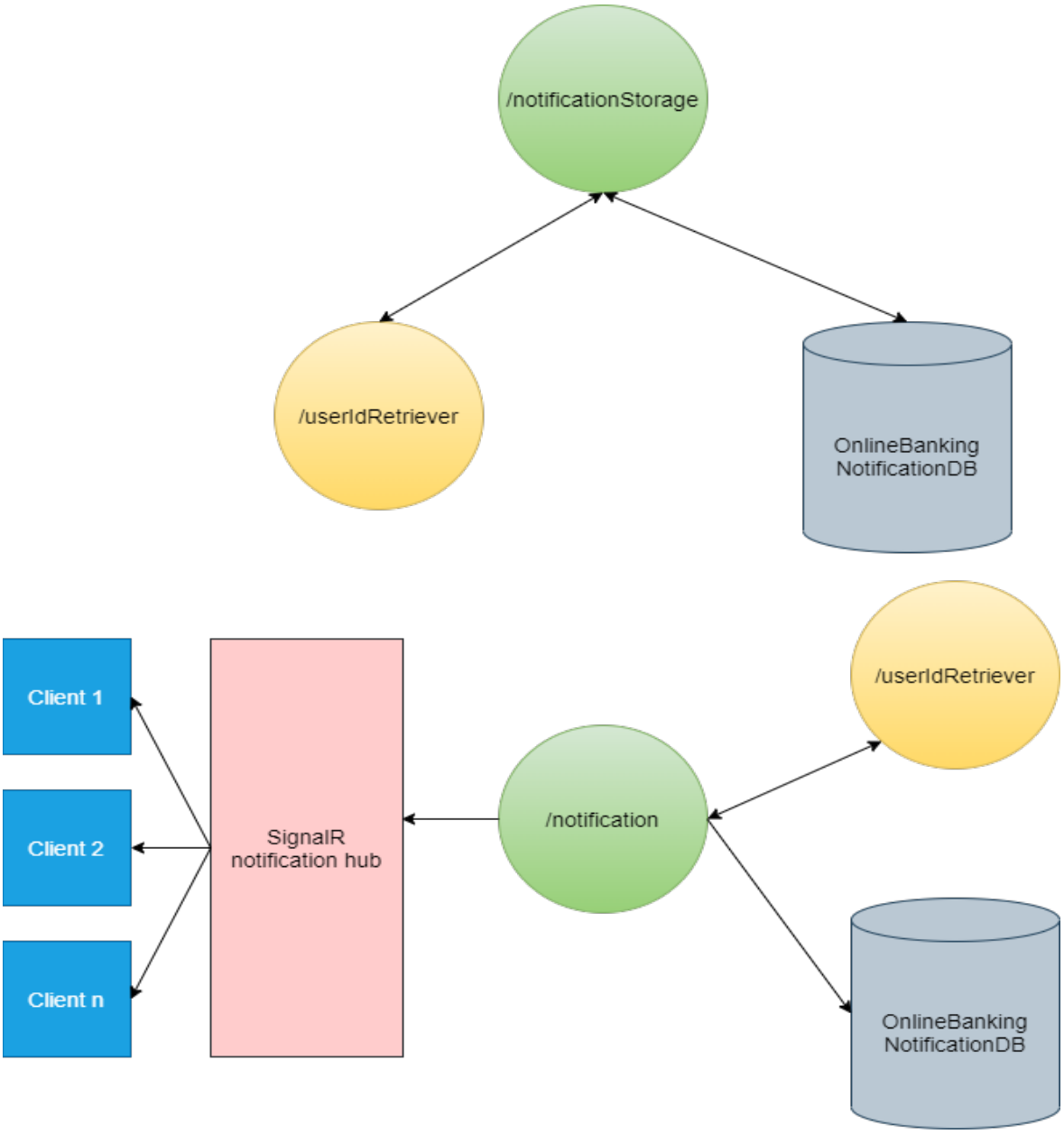
### 5.3.9 Хијерахијски приказ архитектуре сервиса за помоћ




*Слика 5.22 Архитектура сервиса за добијање помоћи*

Сервис за добијање информација о имејлу или броју телефона помоћу којих се може контактирати сервисер за савет или помоћ захтева једног актера који врши комуникацију са посебном базом како би добио захтеване информације (Слика 5.22 Архитектура сервиса за добијање помоћи).

### 5.3.10 Хијерархијски приказ архитектуре сервиса за нотификације




*Слика 5.23 Архитектура сервиса нотификација*

Нотификациони сервис се користи за слање разних нотификација кориснику и то је једини сервис који иницира комуникацију са корисницима, а да му није упућен захтев са клијентске стране. Како бисмо ово омогућили користимо библиотеку *SiganlR* која представља одређену врсту хаба и која има уграђене механизме двосмерне комуникације помоћу којих шаље информације кориснику у реалном времену без претходне потребе за захтевањем тих информација од стране клијента (Слика 5.23 Архитектура сервиса нотификација) [21][22]. Сервис има могућност да шаље нотификације како одређеном

кориснику тако и свим корсницима или некој групи корисника (на пример, корисницима који имају одређене улоге у систему). Нотификације се чувају у бази нотификација.

У реактивним дистрибуираним системима сваки сервис мора имати могућност да буде развијан, испоручен и скалиран независно. Да бисмо то постигли за сваки сервис је коришћена независна база података, односно коришћен је приступ база података по сервису [23]. На тај начин сваки сервис може да користи тип базе који му највише одговара и омогућава да сервиси не буду тесно спрегнути.

# 6 Имплементација апликације

У овом поглављу ће бити приказани и описани неки од занимљивих детаља имплементације сервиса онлајн банкарског система *BANKX22*.

## 6.1 Имплементација ауторизације и аутентификације корисника

Када корисник жели да изврши пријављивање или регистровање на систем, најпре је потребно да унесе све неопходне податке које му приказује *Login* компонента. Притиском на дугме „Потрврди*“* позива се функција *handleSubmitClick* која је регисторвана да се покрене када се деси догађај притиска миша на поменуто дугме. Задатак ове функиције је да у зависности од тога да ли се ради о пријави или регистрацији, обавести библиотеку *Redux*, шаљући јој одговарајуће податке, да треба да контактира *API* сервер и да му саопшти да је покушај пријаве/регистрације у току. За комуникацију између *React* компоненте и библиотеке *Redux* користи се функција *dispatch.* Пошто је у имплементацији коришћен функционални приступ писања компоненти[13] како би функција *dispatch* била доступна компоненти искоришћен је хук[14] *useDispatch. Dispatch* као аргумент прима објекат назван акција који се може добити и као резултат функције која се зове креатор акције (више детаља у поглављу 4.2 Библиотека *Redux*). Тај приступ је и примењен у коду испод. Функција *dispatch* је позвана са акцијом коју ће вратити креатор акције *auth* који као параметар прима објекат са мејлом и шифром уколико се ради о пријави корисника или са акцијом добијеном од креатора акције *signUp* уколико се ради о регистрацији корисника*.*

[13] Приликом коришћења библиотеке *React* компоненте је могуће писати као компоненте базиране на функцијама и компоненте базиране на класама. У време писања овог рада преферира се писање компоненти базираних на функцијама у односу на конвенционални приступ компонената заснованих на класама. Предности су како краћи и концизнији код, тако и све предности које доноси функционални приступ програмирања.

[14] Хук (енг. *Hook*) – буквални превод значи удица или кука али је доста популарнији англицизам па ће у наставку текста бити тако и ословљаван. Хук је специјална врста функције која садржи логику која је потребна многим компонентама.

```
const handleSubmitClick = (infoObject, isSignUpPage) => {
    if (!isSignUpPage) {
      dispatch(
        auth({ email: infoObject.email, password: infoObject.password })
      );
    } else {
      dispatch(signUp(infoObject));
    }
  };
```

Код за креаторе акција *auth* и *signUp* се може видети испод. Функција *createAsyncThunk* је део библиотеке *Redux Toolkit* (поглавље 4.3 *Redux*-ов скуп алата) која прима тип акције у форми стринга и функцију са повратним позивом *payloadCreator* која би требала да врати објекат *Promise*[15]. *PayloadCreator* ће генерисати *Promise* од животног века типова акција које су базиране на префиксу типа акције прослеђеног као параметар и вратиће креатора акције *thunk* који ће извршити функцију са повратним позивом *Promise*-а и обавестиће акције животног века на основу враћеног *Promise-a* [9]. *PayloadCreator* функције *auth* има задатак да пошаље податке за пријаву *Login* контролеру на серверу. То ради тако што ће искористити библиотеку *axios,* погодну за слање асинхроних позива ка неком *API*-ју, и позвати њену методу *post* са прослеђеном *URL* адресом и подацима за пријаву. Уколико сервер врати информације о успешној пријави оне се прослеђују као одговор, а иначе се враћа опис грешке која се јавила при пријави. Функција *createAsyncThunk* враћа креатор акције редукса *thunk*. Ова функција ће имати своје креаторе акција за случај ако се нађе у нерешеном (*енг. pending*), испуњеном (*енг. fulfilled*) или одбаченом (*енг. rejected*) стању који ће бити испаљивани како *thunk* буде прелазио из стања у стање [9]. Они су искоришћени како би *Redux* могао да ажурира одговарајуће стање апликације и на тај начин јави *React-y* да је потребно да прикаже кориснику нови садржај.

```
export const auth = createAsyncThunk(
  "login/auth",
  async (data, { rejectWithValue }) => {
    try {
      const response = await axios.post(
        API_URL + ENDPOINT_PATHS.AUTHENTICATE,
        data
```

[15] Promise – на српски се може превести као обећање, представља неку врсту чувара за вредност која није позната у тренутку када је Promise креиран. Он представља објекат евентуалног завршетка (како успешног тако и неуспешног) асинхроне операције и њене резултујуће вредности. Концепт је врло сличан концепту за објекат *Future* са том разликом да је *Future* контејнер за резултат који не постоји из кога се може само читати, док се у објекат *Promise* може вршити и упис [24].

```
      );
      return response.data;
    } catch (err) {
      const error = extractErrorsFromResponse(
        err.response ? err.response.data : err
      );
      return rejectWithValue(error);
    }
  }
);

export const signUp = createAsyncThunk(
  "login/signUp",
  async (data, { rejectWithValue }) => {
    try {
      const response = await axios.post(
        API_URL + ENDPOINT_PATHS.REGISTER,
        data
      );
      return response.data;
    } catch (err) {
      const error = extractErrorsFromResponse(
        err.response ? err.response.data : err
      );
      return rejectWithValue(error);
    }
  }
);

export const loginSlice = createSlice({
  name: "login",
  initialState,
  extraReducers: {
    [auth.pending]: (state, action) => {
      state.status = FETCH_DATA_STATUS.LOADING;
    },
    [auth.fulfilled]: (state, action) => {
      if (action.payload?.errorMessage) {
        state.status = FETCH_DATA_STATUS.FAILED;
        state.error = action.payload.errorMessage;
      } else {
        state.status = FETCH_DATA_STATUS.SUCEEDED;
        state.token = action.payload.userToken;
      }
    },
```

```
[auth.rejected]: (state, action) => {
  state.status = FETCH_DATA_STATUS.FAILED;
  if (action.payload) {
    if (typeof action.payload === "string") {
      state.error = action.payload;
    } else if (action.payload.errorMessage)
      state.error = action.payload.errorMessage;
    else {
      state.error = "ErrorOccurred";
    }
  } else {
    if (action.error) {
      if (action.error.message) state.error = action.error.message;
      else if (typeof action.error === "string") state.error = action.error;
      else {
        state.error = "ErrorOccurred";
      }
    } else {
      state.error = "ErrorOccurred";
    }
  }
},
[signUp.pending]: (state) => {
  state.registrationStatus = FETCH_DATA_STATUS.LOADING;
},
[signUp.fulfilled]: (state, action) => {
  if (action.payload.isSuccess) {
    state.registrationStatus = FETCH_DATA_STATUS.SUCEEDED;
  } else if (action.payload?.errorMessage) {
    state.registrationStatus = FETCH_DATA_STATUS.FAILED;
    state.error = action.payload.errorMessage;
  } else {
    state.registrationStatus = FETCH_DATA_STATUS.FAILED;
  }
},
[signUp.rejected]: (state, action) => {
  state.registrationStatus = FETCH_DATA_STATUS.FAILED;
  if (action.payload) {
    if (typeof action.payload === "string") {
      state.error = action.payload;
    } else if (action.payload.errorMessage)
      state.error = action.payload.errorMessage;
    else {
      state.error = "ErrorOccurred";
    }
```

```
      } else {
        if (action.error) {
          if (action.error.message) state.error = action.error.message;
          else if (typeof action.error === "string") state.error = action.error;
          else {
            state.error = "ErrorOccurred";
          }
        } else {
          state.error = "ErrorOccurred";
        }
      }
    },
  },
});
```

Код за регистрацију је готово идентичан са том разликом да ће *payloadCreator* функције *signUp* генерисати *Promise* од животног века типова акција које су базиране на префиксу *signUp* и послаће информације за регисторвање *Login* контролеру.

*И*мплементација *Login* конторлера је приказана кодом испод. Конторлер је декорисан са два атрибута. *ApiPrefixRoute* је кориснички дефинисан атрибут који садржи шаблон потписа контролера (шаблон изгледа *“api/[controller]”* где се све у угластим заградама мења са именом контролера, што значи *“api/login”*). *EnableCors* је атрибут којим се дозвољава да се контролеру приступа са различитог домена од домена на коме је лоциран контролер [5]. Сви помоћни сервиси попут *_logger, _loginManagerActor, _loginIncrementor* се добијају применом принципа уметања зависности приликом инстанцирања самог контролера. Тада се у конструктору они могу добити из контејнера инверзне контроле. Да би се нашли у контејнеру потребно је да буду регистровани. Регистрација ће бити приказана касније када се буде описивала конфигурација сервиса *.NET Web API* слоја. Контролер има имплементиране две акције *Authenticate* и *Register* које ће бити позване од стране *Redux-a* на клијентској страни када буде послат одговарајући *HTTP* захтев.

```
[ApiPrefixRoute]
[EnableCors("AllowOrigin")]
public class LoginController : ControllerBase
{

        private readonly ILoggerManager _logger;
        private readonly IActorRef _loginManagerActor;
        private readonly IIncrement _loginIncrementor;
        public LoginController(ILoggerManager logger, LoginManagerActorProvider
                        loginManagerActorProvider, IIncrement incrementor)
        {
```

```
                _logger = logger;
                _loginManagerActor = loginManagerActorProvider();
                _loginIncrementor = incrementor;
        }

        [HttpPost("auth/")]
        public async Task<IActionResult>
                Authenticate([FromBody] AuthenticationData authData)
        {
                var result = await _loginManagerActor.Ask(
                new Authenticate(_loginIncrementor.Increment(nameof(Authenticate)),
                                authData.Email, authData.Password));
                return Ok(result);
        }

        [HttpPost("signUp/")]
        [HttpPost("register/")]
        public async Task<IActionResult>
                 Register([FromBody] RegistrationData registrationData)
        {
                var registerDataMessage =
                        new Register(_loginIncrementor.Increment(nameof(Register)),
                        registrationData.Email, registrationData.Password,
                        registrationData.ConfirmPassword, registrationData.BankId,
                        registrationData.Mobile,registrationData.UserName,
                        registrationData.Address, registrationData.HomePhone);
                var result = await _loginManagerActor.Ask(registerDataMessage);
                return  Ok(result);
        }

}
```

Акције добијају податке као аргументе који су већ прошли одређену проверу исправности од развојног окружења стране. Провера исправности се врши тако што се типови аргумената (у коду изнад *AuthenticationData* и *RegistrationData*) декоришу са атрибутима валидаторима и приликом прихватања захтева, развојно окружење *.NET Web API* проверава да ли податак задовољава све услове постављене од стране валидатора. Ако је то случај креира се порука која ће бити послата *_loginManagerActor* актеру коришћењем упитног обрасца. Порука у случају пријаве је *Authenticate* док је у случају регистровања *Register.* Свака порука има јединствени идентификатор који се одређује позивом методе *Increment* која је део *_loginIncrementator* сервиса. Идентификатор је користан у случају да рецимо дође до рестартовања актера. Тада на основу њега актер може лакше да поврати пређашње стање или ако је потребна да се изврши нека врста координације разних операција у различитим актерима [4]. Због употребе упитног обрасца који враћа објекат будућности, контролер ће резултат када буде спреман уписати у променљиву резултат и њу вратити као одговор на захтев [22].

Порука која је послата актеру *_loginManagerActor* ће бити прихваћена од стране методе *OnReceive* актера, која је наслеђена метода базне класе *BaseUntypedActor* и коју сваки кориснички дефинисани актер мора да наследи како би могао да има сва својства која се очекују од актера. Код *OnReceive* методе је приказан испод. Метод као параметар

прима поруку која има тип *object.* Како је ово тип који је родитељ свим осталим типовима у *.NET* свету то значи да овај метод може примити било који тип под условом да се изврши кастовање у њега приликом коришћења.

У коду можемо видети да порука пролази кроз механизам *switch* конструкције и ако је неког од наведеног типа у *case* услову конвертује се у тај тип и шаље даље неком актеру на обраду или се враћа контролеру од кога је стигао првобитни захтев. На пример уколико стигне порука за пријаву корисника (*енг. Authenticate*) најпре ће контролер који је послао поруку бити сачуван у речник, како би одговор могао да му буде враћен, а потом се порука шаље актеру аутентификације који је инстанциран као дете *_loginManagerActor* актера. Порука ће се наћи у сандучету актера аутентификације који такође има имплементиран метод *OnReceive* и када та порука стигне на ред да буде обрађена метод је прихвата.

Пошто је потребно неколико операција да се одради пре него што актер дозволи пријаву, он инстанцира актере валидације, генерисања корисничког токена и актера који комуницира са базом података корисника како би сваки обавило одговарајућу операцију. Када су подаци прошли валидацију и након што је токен изгенерисан, дохватају се подаци корисника и шаљу се назад актеру *_loginManagerActor* који их враћа контролеру. За комуникацију између актера користи се метод *Tell* који се још и зове „Испали и заборави“ зато што након слања поруке не чека одговор као што је то случај код упитног обрасца (3.6 Обрасци комуникације) већ може да пошаље и референцу пошиљаоца поруке, па кад прималац буде спреман да одговори позива метод *Tell* дефинисан на референци пошиљаоца [1]. Оваквим приступом не долази до застоја унутар актера.

```
protected override void OnReceive(object message)
{
       switch (message)
       {

              case Register reg:
                     logger.Info($"{ActorName},message received:
                     {reg}");
                     controllers.TryAdd("Register" +
                                   reg.RequestId.ToString(),
                            Sender);
                     registrationActor.Tell(reg, Self);
                     break;
              case UserSaved us:
                     logger.Info($"{ActorName},message received:
              {us}");
                     var registerControllerRef =
                            controllers["Register" +
              us.RequestId.ToString()];
                     registerControllerRef.Tell(us, Self);
                     controllers.TryRemove("Register" +
                                   us.RequestId.ToString(), out _);
                     break;
              case CanNotRegister cantReg:
                     logger.Info($"{ActorName},message
                     received:{cantReg}");
```

```
                registerControllerRef = controllers["Register" +
                        cantReg.RequestId.ToString()];
                registerControllerRef.Tell(cantReg, Self);
                controllers.TryRemove("Register" +
                        cantReg.RequestId.ToString(), out _);
                break;
            case Authenticate auth:
                logger.Info($"{ActorName},message received:
                {auth}");
                controllers.TryAdd("Authenticate" +
                        auth.RequestId.ToString(), Sender);
                authenticationActor.Tell(auth, Self);
                break;
            case UserFetched userFetched:
                logger.Info($"{ActorName},message received:
                        {userFetched}");
                registerControllerRef =
            controllers[$"Authenticate{userFetched.RequestId}"];
                registerControllerRef.Tell(new
                        Authenticated(userFetched.RequestId,
                        userFetched.Token), Self);
                controllers.TryRemove($"Authenticate
                        {userFetched.RequestId}", out _);
                notificationActor.Tell(new
                        SendSavedNotifications(
                        userFetched.RequestId,
                        userFetched.UserId), Self);
                break;
            case TokenGenerationFailed genFailed:
                logger.Info($"{ActorName},message received:
                        {genFailed}");
                registerControllerRef =
            controllers[$"Authenticate{genFailed.RequestId}"];
                registerControllerRef.Tell(genFailed, Self);
    controllers.TryRemove($"Authenticate{genFailed.RequestId}",
                    out _);
                break;
            case FetchUserFailed fetchUserFailed:
                logger.Info($"{ActorName},message received:
                        {fetchUserFailed}");
                registerControllerRef =
                controllers[
                $"Authenticate{fetchUserFailed.RequestId}"];
                registerControllerRef.Tell(fetchUserFailed,
            Self);
                controllers.TryRemove(
                $"Authenticate{fetchUserFailed.RequestId}", out
            _);
                break;
            case AuthenticationFailed authFailed:
                logger.Info($"{ActorName},message received:
                {authFailed}");
                registerControllerRef =
            controllers[$"Authenticate{authFailed.RequestId}"];
                registerControllerRef.Tell(authFailed, Self);
                controllers.TryRemove(
                $"Authenticate{authFailed.RequestId}", out _);
                break;
```

```
            }
        }
```

## 6.2 Имплементација конфигурација *.NET Web API* слоја и система актера

По покретању серверске апликације најпре се покреће *.NET Web API* апликација (4.5.*Net Web API* интерфејс за програмирање апликација). У њој се креира *HostBuilder* који представља неку врсту скеле *.NET Web API* апликације. *HostBuilder* се при покретању конфигруише тако што му се шаље специјална класа која се зове *Startup* и имплементира целокупну конфигурацију *.NET Web API* апликације. Њена имплементација је приказана испод. Класа има две методе које ће бити позване од стране развојног окружења приликом покретања *.NET Web API* апликације.

Прва метода се зове *ConfigureService* и као параметар поседује колекцију сервиса који је контејнер инверзне контроле. У контејнер се методама *Add** додају сви сервиси неопходни за рад апликације [5]. У претходном поглављу смо видели пример како се неки од ових сервиса могу користити у *Login* контролеру. Конструктору контролера се као параметар проследи интерфејс који имплементира тај сервис. Приликом извршавања тог конструктора у поље класе које има одговарајући тип интерфејса се изврши додела тог сервиса тако да је доступан свим методама класе. У коду видимо да се у контејнер додају сервис за логовање, сервис за инкрементирање порука, контексти свих база које апликација користи и њихове конфигурације, *SignalR* хаб за нотификације, две реплике система актера као чворови кластери система актера, сервис за додавање актера који комуницирају са контролерима и сервис за логовање уграђене валидације захтева. Поред ових сервиса додати су и сервиси који омогућавају коришћење *CORS* политике, контролере као и библиотеку *SignalR*.

Други метод који имплеменитра класа *Startup* је метод *Configure* који врши неке иницијалне операције апликације као и конфигурсање ланца посредника кроз који мора да прође *HTTP* захтев пре него што дође до акције неког контролера. Сервис *lifetime,* који је један од параметара метода, се користи за покретање и стопирање система актера одмах после подизања, односно стопирања, *.NET Web API* апликације. Параметар *app* представља апстракцију *.NET Web API* апликације и у методи се над њом позивају метода за редирекцију *HTTP* захтева на *HTTPS*, метода за додавње функционалности рутирања, метода за коришћење *CORS-a*, метода за употребу ауторизације и метода за додавање крајњих тачака приступа.

```
public class Startup
{

        readonly string MyAllowSpecificOrigins = "_myAllowSpecificOrigins";
        public Startup(IConfiguration configuration)
        {
                LoggerManager.LoadConfiguration(
                Path.Combine(Directory.GetCurrentDirectory(), "nlog.config"));
                Configuration = configuration;
        }

        public IConfiguration Configuration { get; }

        public void ConfigureServices(IServiceCollection services)
        {

                services.AddSingleton<ILoggerManager, LoggerManager>();

                services.AddSingleton<IIncrement, Incrementor>();

                services.AddDbContext<OnlineBankingContext>(options =>
                        options.UseSqlServer(
                        Configuration.GetConnectionString("OnlineBankingDatabase")
                        , b => b.MigrationsAssembly("OnlineBankingWebApi"))
                        );

                services.AddDbContext<OnlineBankingAccountContext>(options =>
                        options.UseSqlServer(
                        Configuration.GetConnectionString("OnlineBankingAccountDatabase"),
                        b => b.MigrationsAssembly("OnlineBankingWebApi")));

                services.AddDbContext<OnlineBankingTransactionContext>(options =>
                        options.UseSqlServer(
                Configuration.GetConnectionString("OnlineBankingTransactionDatabase"),
                        b => b.MigrationsAssembly("OnlineBankingWebApi")));

                services.AddDbContext<OnlineBankingCurrencyContext>(options =>
                        options.UseSqlServer(
                        Configuration.GetConnectionString("OnlineBankingCurrencyDatabase")
                        , b => b.MigrationsAssembly("OnlineBankingWebApi")));

                services.AddDbContext<OnlineBankingHelpContext>(options =>
                        options.UseSqlServer(
                        Configuration.GetConnectionString("OnlineBankingHelpDatabase")
                        , b => b.MigrationsAssembly("OnlineBankingWebApi")));

                services.AddDbContext<OnlineBankingNotificationContext>(options =>
                        options.UseSqlServer(
                Configuration.GetConnectionString("OnlineBankingNotificationContext")
                        , b => b.MigrationsAssembly("OnlineBankingWebApi")));

                services.AddDbContext<BankUsersContext>(options =>
                        options.EnableSensitiveDataLogging()
                        .UseSqlServer(
                        Configuration.GetConnectionString("BankUsersDatabase")
                        , b => b.MigrationsAssembly("OnlineBankingWebApi")));
```

```
        services.AddSingleton<INotificationHubHelper, NotificationHubHelper>();

        services.AddActorSystem(AkkaSystemConfiguration.SeedNode);

        services.AddActorSystem(AkkaSystemConfiguration.Node);

        services.AddActorProviders();

        services.AddCors(options =>
        {
                options.AddPolicy(name: MyAllowSpecificOrigins,
                        builder =>
                        {
                                builder.WithOrigins("http://localhost:3000",
                                 "http://localhost:9000/api")
                                        .AllowAnyHeader()
                                        .AllowAnyMethod()
                                        .SetIsOriginAllowed((x) => true)
                                        .AllowCredentials();
                        });
                });

        services.AddControllers();

        services.AddSignalR();

        services.PostConfigure<ApiBehaviorOptions>(options =>
        {
                var builtInFactory = options.InvalidModelStateResponseFactory;

                options.InvalidModelStateResponseFactory = context =>
                {
                        var loggerManager =
                                context.HttpContext
                                .RequestServices
                                .GetRequiredService<ILoggerManager>();
                        loggerManager.LogInfo($"Request with URL
                                {context.HttpContext.Request.Host}
                                {context.HttpContext.Request.Path.Value}" +
                                $" reached endpoint
                                {context.ActionDescriptor.DisplayName}" +
                                $" end produce error message:
                                \"{context.ModelState.Values
                                        .First()
                                        .Errors.First()
                                        .ErrorMessage}\"");
                                return builtInFactory(context);
                        };
        });
}


public void Configure(IApplicationBuilder app, IWebHostEnvironment env,
                        IHostApplicationLifetime lifetime)
{
        if (env.IsDevelopment())
        {
```

```
                app.UseDeveloperExceptionPage();
            }

            lifetime.ApplicationStarted.Register(() =>
            {
                app.ApplicationServices.GetService<ActorSystem>();
            });


            lifetime.ApplicationStopping.Register(() =>
            {
                app.ApplicationServices
                    .GetService<ActorSystem>().Terminate().Wait();
            });

            app.UseStaticFiles(
                new StaticFileOptions {
                    FileProvider = new PhysicalFileProvider(
                        Path.Combine(
                            env.ContentRootPath, "ProfileImages")),
                    RequestPath = "/ProfileImages"
                }
            );

            app.UseHttpsRedirection();

            app.UseRouting();

            app.UseCors(MyAllowSpecificOrigins);

            app.UseAuthorization();

            app.UseEndpoints(endpoints =>
            {
                endpoints.MapControllers()
                      .RequireCors(MyAllowSpecificOrigins);
                endpoints.MapHub<NotificationHub>("/hubs/notificationHub");
            });
        }
}
```

У наставку је приказан код за покретање и конфигурисање система актера као и за додавање актера на врху хијерархије у контејнер инверзне зависности како би им се могло приступити у *.NET Web API* апликацији.

```
public static class ServiceCollectionExtensions
{
        public static void AddActorSystem(this IServiceCollection services,
                AkkaSystemConfiguration configurationOption)
        {
            var configPath =configurationOption == AkkaSystemConfiguration.SeedNode ?
                    Path.Combine(ConfigurationConstants.ProjectDirectory,
                        ConfigurationConstants.SeedNodeConfigurationFile) :
```

```
                        Path.Combine(ConfigurationConstants.ProjectDirectory,
                                ConfigurationConstants.NodeConfigurationFile);
            var configString = File.ReadAllText(configPath);
            var config = ConfigurationFactory.ParseString(configString);
            var bootstrap = BootstrapSetup.Create().WithConfig(config);

            services.AddSingleton(provider =>
            {
                var serviceScopeFactory =
                            provider.GetService<IServiceScopeFactory>();
                var di = DependencyResolverSetup.Create(provider);
                var actorSystemSetup = bootstrap.And(di);
                var actorSystem = ActorSystem.Create("OnlineBanking",
                                                        actorSystemSetup);
                actorSystem.AddServiceScopeFactory(serviceScopeFactory);
                return actorSystem;
            });
        }

        public static void AddActorProviders(this IServiceCollection services)
        {
            services.AddSingleton<LoginManagerActorProvider>(provider =>
            {
                var actorSystem = provider.GetService<ActorSystem>();
                var loginActor = actorSystem.ActorOf(
                                    LoginManagerActor.Props()
                                    .WithRouter(FromConfig.Instance),
                                    "loginManager");
                return () => loginActor;
            });

            services.AddSingleton<AccountGetterActorProivder>(provider =>
            {
                var actorSystem = provider.GetService<ActorSystem>();
                var accountGetterActor = actorSystem.ActorOf(
                                    AccountGetterActor.Props()
                                    .WithRouter(FromConfig.Instance),
                                    "accountGetterActor");
                return () => accountGetterActor;
            });

            ...


        }
}
```

Класа *ServiceCollectionExtension*, описана у коду изнад представља проширење контејнера инверзне контроле. Она садржи две методе проширења[16] које *services* контејнер позива у *ConfigureServices* методи *.NET Web API* апликације. Метода *AddActorSystem*

[16] Метода проширења (*енг. extension method*) представља додавање функционалности над већ постојећим типом података и карактерише је кључна реч ***this***, употребљена као префикс првог аргумента. Први параметар представља инстанцу типа којем је придружен овај метод [5].

најпре учитава конфигурациони фајл система актера у зависности да ли се ради о семену или обичном чвору, потом омогућава механизам уметања зависности у систем актера, креира систем актера са називом *OnlineBanking* и прослеђује му конфигурацију. Систем актера региструје у контејнер инверзне контроле као једину инстанцу у контејнеру, тако да кад год систем буде тражен из контејнера биће враћена иста инстанца система.

Метода *AddActorProviders* додаје актере који су дефинисани као први ниво хијерахије (директни потомци *user* актера) у контејнер. У коду изнад је приказана логика додавања *loginManager* и *account* актера. На исти начин се додају и остали актери. Најпре је потребно добити систем актера из контејнера, који мора бити претходно регирстрован. Потом се позива метод *ActorOf* на инстанци система како би се креирао потребан актер. Овој методи се прослеђује *Props* објекат који представља врсту рецепта (на који начин се актер инстанцира са одређеном конфигурацијом и информацијама испоручивања) помоћу којег систем актера инстанцира одређеног актера. Занимљив детаљ је да се актер инстанцира са механизмом рутирања, чија подешавања се читају из конфигурационог фајла. Рутери су конфигурисани да буду свесни свих чворова у групи, тако да ће добијена порука бити рутирана најмање оптерећеној инстанци актера на мање оптерећеном чвору.

```
akka {
	loglevel = DEBUG
	loggers = ["Akka.Logger.NLog.NLogLogger, Akka.Logger.NLog"]
	actor.provider = cluster
	remote {
		dot-netty.tcp {
			port = 5002
			hostname = localhost
		}
	}

	cluster {
		seed-nodes = ["akka.tcp://OnlineBanking@localhost:5002"]
	}

	actor
	{
		debug
		{
			receive = on
			autoreceive = on
			lifecycle = on
			event-stream = on
			unhandled = on
		}

		deployment
		{
			/loginManager {
				router = smallest-mailbox-pool
				nr-of-instances = 10
				cluster {
					enabled = on
					allow-local-routees = on
```

```
                        max-nr-of-instances-per-node = 5
                }
        }
        /accountGetterActor {
                router = round-robin-pool
                nr-of-instances = 10
                cluster {
                        enabled = on
                         allow-local-routees = on
                         max-nr-of-instances-per-node = 5
                }

        }

...
```

Претходни код представља део конфигурационог фајла, написаног у *HOCON*[17] формату, који се односи на употребу логовања унутар система, конфигурисање чворова у групи као и начина рутирања сваког од актера. Видимо да се за сваки рутер одређује политика рутирања (нпр. *smallest mailbox pool* или *round robin*), број инстанци актера који ће бити креирани од стране рутера, дозвала за употребу свих чворова у кластеру као и максималан број инстанци на једном чвору [4]. Овај сегмент конфигурације представља конфигурацију семена чвора, што можемо видети под секцијом *cluster* где је наведено да ће тај чвор бити покренут на адреси *"akka.tcp://OnlineBanking@localhost:5002"*. Овде такође можемо видети колико је једноставно извршити конфигурисање локације где ће сваки чвор бити покренут, тако да је у случају отказа сервера на некој адреси, могуће, врло једноставно, извршити преконфигурацију тако да се чвор нађе на некој другој адреси.

## 6.3 Имплементација сервиса за извршавање плаћања корисника

Имплементација сервиса за извршавање плаћања на корисничкој страни користи исте концепте и конструкте као и сервис за пријављивање и регистровање корисника. Корисник види на екрану садржај који му рендерује *Payment React*-ова компонента. Она садржи форму плаћања, чији подаци се након попуњавања шаљу *dispatch* методом као акцију *sendPaymentInfo Redux-y*. *Redux* помоћу, већ описане функције, *createAsyncThunk* шаље информације *Payment* контролеру *.NET Web API* апликације и одашиље нову акцију којом обавештава да је захтев тренутно у нерешеном стању. React ће све док траје нерешено стање приказивати кориснику *Loader* и на тај начин обавештавати корисника да је процес обраде у току.

На серверској страни акција контролера *Payment* ће захтев конвертовати у поруку и послати је *Payment* актеру. Битна карактеристика *Payment* актера је та што наслеђује

[17] *HOCON* (*енг. Humen Optimized Configuration Object Notation*) је конвенција писања конфигурационих фајлова у облику који је погодан за човека и представља надоградњу *JSON* конвенције [4].

*BaseUntypedPersistentActor* актера који омогућава да се унутрашње стање актера сачува у бази података. Чување стања омогућава његов опоравак у случајевима када је актер стартован, рестартован након пада апликације или од стране супервизора због неког другог разлога или након мигрирања актера у групи. Идеја је да свака порука која стигне у сандуче актера буде обрађена методом *OnCommand. OnCommand* метода ће извршити чување добијене команде, тако што ће је представити као догађај и тај догађај ће бити сачуван у базу методом *Persist.* Када *Persist* сачува догађај у бази, извршиће функцију са повратним позивом у којој се наставља извршавање актера. Ако дође на пример до рестартовања актера, најпре ће се извршити *OnRecover* метода која добија редом сачуване догађаје и на основу њих може вратити стање да буде идентично стању актера пре рестарта. Чување информација као догађаје се назива „извор догађаја“ (за детаље погледати поглавље 3.8 Обрасци управљања и чувања стања) и има доста предности у односу на конвенционално чување података у релационим базама, као што је на пример могућност праћења историје догађаја и проверавање историје трансакција, што је веома битан фактор у финансијским системима. Код имплементације *Payment* актера приказан је у наредном делу.

```
public class PaymentActor : BaseUntypedPersistentActor, ILogReceive
{
	public override string PersistenceId => Self.Path.ToString();
	private readonly PaymentState state = new()
		{ Controllers = new(), Messages = new() };

	//Initialization of children actors
	...

	private int _msgsSinceLastSnapshot = 0;

	public PaymentActor() :base(nameof(PaymentActor)){}

	protected override void OnCommand(object message)
	{
		switch (message)
		{
			case Pay pay:
				logger.Info($"{ActorName}, message received : {pay}");
				state.Controllers.TryAdd(pay.RequestId, Sender);
				state.Messages.TryAdd(pay.RequestId, pay);
				Persist(new PayArrivedEvent(pay.RequestId, pay, Sender),
				(payArrivedEvent) => {
					if (++_msgsSinceLastSnapshot % 100 == 0)
					{
						SaveSnapshot(state);
						_msgsSinceLastSnapshot = 0;
					}
					logger.Info($"{ActorName},
					 message persisted: {payArrivedEvent}");
					userIdRetrieverActor.Tell(
						new RetrieveUserId(pay.RequestId,
							pay.UserToken), Self);
				});
				break;
			case UserIdRetrieved userIdRetrieved:
```

```
                    logger.Info($"{ActorName} ,
                            message received : {userIdRetrieved}");
                    var payMessage =
                            state.Messages[userIdRetrieved.RequestId];
                    accountStorageActor.Tell(
                            new GetAccountsForPayment(payMessage.RequestId,
                            userIdRetrieved.UserId,payMessage.AccountNumber,
                            payMessage.BenericieryCustomerAccount));
                    break;
            case RetrievedAccountsForPayment retrievedAccountsForPayment:
                    logger.Info($"{nameof(ActorName)},
                     message recived {retrievedAccountsForPayment}");
                    state.Messages.TryRemove(
                            retrievedAccountsForPayment.RequestId,
                             out payMessage);
                    state.Controllers.TryRemove(
                            retrievedAccountsForPayment.RequestId,
                            out controller);
                    Persist(
                    new RetrievedAccountsForPaymentEventArrived(
                            retrievedAccountsForPayment.RequestId),
                    (retrievedAccountsForPaymentEventArrived) => {
                            logger.Info($"{ActorName},
                            message persisted
                             {retrievedAccountsForPaymentEventArrived}");
                            if(retrievedAccountsForPayment.
                                    BeneficieryAccount == null)
                            {
                                    paymentPerformerActor.Tell(
                                    new PayOnRemoteAccount(
                                    payMessage.RequestId,
                                    retrievedAccountsForPayment.UserAccount,
                                    payMessage.Amount,
                                    payMessage.BeneficieryCustomer,
                                    payMessage.BenericieryCustomerAccount,
                                    payMessage.Model,
                                    payMessage.Reference,
                                    payMessage.PaymentCode,
                                    payMessage.PaymentPurpose,
                                    controller),
                                     Self);
                            }
                            else
                            {
                                    paymentPerformerActor.Tell(
                                    new PayOnLocalAccount(
                                    payMessage.RequestId,
                                    retrievedAccountsForPayment.UserId,
                                    retrievedAccountsForPayment.UserAccount,
                                    retrievedAccountsForPayment.
                                                        BeneficieryAccount,
                                    payMessage.Amount,
                                    payMessage.BeneficieryCustomer,
                                    payMessage.Model,
                                    payMessage.Reference,
                                    payMessage.PaymentCode,
                                    payMessage.PaymentPurpose,
                                    controller),
```

```
                                        Self);
                                }

                        });
                        break;
                //other cases of commands
                ...
                case SaveSnapshotSuccess saveSnapshotSuccess:
                        DeleteMessages(
                                saveSnapshotSuccess.Metadata.SequenceNr);
                        break;
                case SaveSnapshotFailure saveSnapshotFailure:
                        logger.Error($"{ActorName}, failed to save snapshot");
                        break;
        }
}

protected override void OnRecover(object message)
{
        switch (message)
        {
                case PayArrivedEvent payArrivedEvent:
                        logger.Info($"{ActorName},
                         message recovered : {payArrivedEvent}");
                         state.Controllers.TryAdd(payArrivedEvent.RequestId,
                                        payArrivedEvent.Controller);
                         state.Messages.TryAdd(payArrivedEvent.RequestId,
                         payArrivedEvent.Pay);
                         break;
                case RetrievingUserIdFailedArrivedEvent
                                retrievingUserIdFailedArrivedEvent:
                        logger.Info($"{ActorName},
                        message recovered : {retrievingUserIdFailedArrivedEvent}");
                        state.Controllers.TryRemove(
                                retrievingUserIdFailedArrivedEvent.RequestId,
                                out _);
                        state.Messages.TryRemove(
                                retrievingUserIdFailedArrivedEvent.RequestId,
                                out _);
                                break;
                //all other cases for events
                ...

                case SnapshotOffer offer:
                        logger.Info($"{ActorName}, snapshot offer : {offer}");
                        if (offer.Snapshot is PaymentState snapshotType)
                        {
                                state.Controllers =
                                        new ConcurrentDictionary<ulong, IActorRef>
                                        (state.Controllers.
                                        Concat(snapshotType.Controllers));
                                state.Messages =
                                         new ConcurrentDictionary<ulong, Pay>
                                        (state.Messages.
                                        Concat(snapshotType.Messages));
                        }
                        break;
        }
```

```
        }

        public static Props Props() => Akka.Actor.Props.Create(() => new PaymentActor());

}
```

Занимљива карактеристика код актера који чувају своје стање је да се у сваком тренутку може одрадити његов снимак стања позивањем методе *SaveSnapshot. Payment* актер снима своје стање након сваких сто плаћања. На тај начин се може драматично смањити опоравак актера након рестартовања јер се не мора проћи кроз целу историју догађаја, већ се може почети од последњег снимка. Развојно окружење ће послати *SnapshotOffer* тип методи *OnRecover* приликом рестартовања актера који ће га сачувати као тренутно стање и након тога само ажурирати стање на основу догађаја који су се догодили након снимка.

*Payment* актер шаље различите поруке *PaymentPerformer* актеру у зависности од тога да ли рачун уплатиоца припада банковном систему или је у питању рачун неке друге банке или система. У случају да се ради о рачуну банковног система шаље се *PayOnLocalAccount* порука, а у супротном се шаље *PayOnRemoteAccount.* У зависности од поруке коју прими актер *PaymentPerformer* ће прећи у одговарајуће стање коришћењем методе *BecomeStacked.*

*BecomeStacked* метода замењује постојеће понашање актера, понашањем дефинисаним у методи која се прослеђује *BecomeStacked* методи као аргумент. Ово ново понашање се ставља на врх стека и користи се све док се ново понашање не дода на врх стека поновним позивом *BecomeStacked* методе или се не уклони са стека позивом *UnbecomeStacked.* Уколико у сандуче актера стигне порука која не може бити обрађена тренутним понашањем актера, рецимо код *PaymentPerformer* актера ако се налази у *ProcessLocalPayment* стању, а стигне му *PayOnRemoteAccount* порука, она се може ускладиштити позовом методе *Stash* и када је актер спреман да пређе у стање које може да обради ту поруку, може се доћи до ње позивањем методе *UnstashAll* која вади све ускладиштене поруке, а да при томе не губи редослед којим су пристизале у сандуче актера. Код за *OnCommand* методу *PaymentPerformer* актера где се врши прелазак у *ProcessLocalPayment* понашање уколико стигне *PayLocalAccount* порука, односно у *ProcessRemotePayment* понашање у случају да стигне *PayOnRemoteAccount* порука, је приказан у наредном делу.

```
protected override void OnCommand(object message)
{
        switch (message)
        {
                case PayOnLocalAccount payOnLocalAccount:
                        logger.Info($"{ActorName} - {payOnLocalAccount} received.");
                        _state.LocalAccountMessages.TryAdd(
                                payOnLocalAccount.RequestId, payOnLocalAccount);
                        Persist(
                                new PayOnLocalAccountEventArrived(
                                        payOnLocalAccount.RequestId, payOnLocalAccount)
```

```
                        , (eventArrived)=> {
                                if (++_msgsSinceLastSnapshot % 100 == 0)
                                {
                                        SaveSnapshot(_state);
                                        _msgsSinceLastSnapshot = 0;
                                }
                                logger.Info($"{ActorName}
                                        , message persisted {eventArrived}");
                                if (payOnLocalAccount.UserAccount.Amount
                                                >= payOnLocalAccount.Amount)
                                {
                                        currencyExchangeRateActor.Tell(
                                        new PerformCurrencyExchangeRate(
                                        payOnLocalAccount.RequestId,
                                        payOnLocalAccount.UserId,
                                        payOnLocalAccount.UserAccount.Currency,
                                        payOnLocalAccount.BeneficieryAccount.Currency),
                                        Self);
                                        BecomeStacked(ProcessLocalPayment);
                                }
                                else
                                {
                                        logger.Warning($"{ActorName},
                                 account does not have enough funds to proceed with payment");
                                        payOnLocalAccount.Sender.Tell(
                                                new CouldNotPerformPayment(
                                                payOnLocalAccount.RequestId,
                                                PaymentError.NoSufficientFundsError));
                                }
                        });
                        break;
                case PayOnRemoteAccount payOnRemoteAccount:
                        logger.Info($"{ActorName} - {payOnRemoteAccount} received.");
                        _state.RemoteAccountMessages.TryAdd(
                                        payOnRemoteAccount.RequestId,
                                        payOnRemoteAccount);
                        Persist(
                                new PayOnRemoteAccountEventArrived(
                                payOnRemoteAccount.RequestId,payOnRemoteAccount),
                                (eventArrived) =>
                                {
                                        if (++_msgsSinceLastSnapshot % 100 == 0)
                                        {
                                                SaveSnapshot(_state);
                                                _msgsSinceLastSnapshot = 0;
                                        }
                                        logger.Info($"{ActorName},
                                                message persisted {eventArrived}");
                                        var userAccount = payOnRemoteAccount.UserAccount;
                                        if (userAccount.Amount >= payOnRemoteAccount.Amount)
                                        {
                                                userAccount.Amount -= payOnRemoteAccount.Amount;
                                                accountStorageActor.Tell(
                                                        new UpdateAccounts(
                                                        payOnRemoteAccount.RequestId,
                                                        new List<Account>() { userAccount })
                                                        , Self);
                                                        BecomeStacked(ProcessRemotePayment);
```

```
                                }
                                else {
                                        logger.Warning($"{ActorName},
                                account does not have enough funds to proceed with payment");
                                        payOnRemoteAccount.Sender.Tell(
                                        new CouldNotPerformPayment(
                                                payOnRemoteAccount.RequestId,
                                                PaymentError.NoSufficientFundsError));
                                }
                        });
                        break;
        }
}
```

# 7 Закључак

Систем онлајн банкарства *BANKX22* представља савремен начин имплементације једног финансијског система спремног да одговори на изазове попут сигурности, доступности, одзивности и издржљивости. Сврха овог система је да покаже како се коришћењем једног модела као што је систем актера могу обављати финансијске трансакције, а да притом задовољи све услове које један реактиван дистрибуиран систем треба да има. Коришћењем овог модела и употребом образаца описаних у претходним поглављима имплементирани су сервиси система.

Сваки сервис система је имплементиран тако да буде лабаво везан тј. да може да ради као независна целина. Сваки од сервиса приказује неку занимљиву карактеристику коју пружа корисничко окружење *Akka*, као што су: образац извор догађаја имплементиран механизмом за чување догађаја *UntypedPersistentActor* актера, образац прекидач кола, механизми супервизије и надгледање актера деце од стране актера родитеља који омогућавају ефикасно решење за руковање грешкама и отказима, механизам употребе прослеђивања порука који избегава закључавање и блокирање извршавања кода, елегантан начин рада у вишенитном систему без коришћења сложених механизама синхронизације као и транспарентност без обзира да ли се цео систем извршава на једној или више машина.

На клијентској страни система онлајн банкарства *BANKX22* приказани су савремени начин коришћења библиотека *React* и *Redux* употребом хукова као и правилан начин структуирања и организовања кода. Ове библиотеке омогућавају писање кода на функционалан начин коришћењем многих елемената реактивне парадигме као што су писање функционалних *React* компоненти правећи дрволику структуру и вршењем комуникације која је вођена догађајима.

Даља унапређења система би имали у виду унапређивање раздвајања система на више локација коришћењем постојећих механизама библиотеке *Cluster*. Један од начина да се то постигне би била примена библиотеке *Cluster Sharding* која је део библиотеке *Cluster* и која омогућава да се сваки сервис актера система подели и дистрибуира на различите чворове. На тај начин сегменти сервиса (група актера) који захтевају доста сложена израчунавања, па самим тим и доста процесорске снаге, се могу дистрибуирати на машинама прилагођеним за то и на тај начин спречити да машина која обавља те операције постане уско грло. Још једна могућност унапређења апликације је да се сваки сервис покрије тестовима који повећавају квалитет, доступност и одговорност апликације. Уколико постоји договор на нивоу сервиса (*енг. Service level agreement – SLA [25]*) између пружаоца сервиса и клијента, тестови би представљали један од основних метода гаранције да се он испоштује.

# 8 Референце


1. Anthony Brown “Reactive Applications with Akka.NET”, Manning, 2019
2. Roland Kuhn, “Reactive Design Patterns”, Manning, 2017
3. Vaughn Vernon, “Reactive Messaging Patterns with the Actor Model: Applications and Integration in Scala and Akka”, Addison-Wesley, 2015
4. Akka.NET (https://getakka.net/index.html)
5. Mark J. Price, “C# 8.0 .NET Core 3.0 – Modern Cross-Platform Development”, Packt, 2019
6. Microsoft Documentation (https://docs.microsoft.com/en-us/documentation/)
7. React Documentation (https://reactjs.org/docs/getting-started.html)
8. Redux Documentation (https://redux.js.org/)
9. Redux Toolkit Documentation (https://redux-toolkit.js.org/)
10. ACID versus BASE Date Stores (https://www.dummies.com/category/articles/big-data-33578)
11. Amdahl’s law (https://en.wikipedia.org/wiki/Amdahl%27s_law#:~:text=In%20computer%20architecture%2C%20Amdahl's%20law,system%20whose%20resources%20are%20improved.)
12. Little's law (https://en.wikipedia.org/wiki/Little%27s_law#:~:text=Little's%20Law%20tells%20us%20that,an%20average%20of%200.5%20hour.)
13. Eric Brewer CAP theorem (https://www.ibm.com/cloud/learn/cap-theorem)
14. CQRS in ASP .NET Core 3.1 (https://www.programmingwithwolfgang.com/cqrs-in-asp-net-core-3-1/)
15. Loose coupling (https://en.wikipedia.org/wiki/Loose_coupling#:~:text=In%20computing%20and%20systems%20design,the%20opposite%20of%20tight%20coupling.)
16. Object-relational impedance mismatch (https://en.wikipedia.org/wiki/Object%E2%80%93relational_impedance_mismatch)
17. Why Event Sourcing? (https://eventuate.io/whyeventsourcing.html)
18. Representational state transfer (https://en.wikipedia.org/wiki/Representational_state_transfer)
19. Martin Flower, Inversion of Control Containers and the Dependency Injection pattern (https://martinfowler.com/articles/injection.html)
20. Alpha Vantage API Documentation (https://www.alphavantage.co/documentation/)
21. SignalR Documentation (https://dotnet.microsoft.com/en-us/apps/aspnet/signalr)
22. Best Practices for Integrating Akka.NET with ASP.NET Core and SignalR (https://petabridge.com/blog/akkadotnet-aspnetcore/)
23. Database per service principle (https://microservices.io/patterns/data/database-per-service.html)
24. Difference between a future and a promise (https://stackoverflow.com/questions/14541975/whats-the-difference-between-a-future-and-a-promise)
25. Service-level agreement (https://en.wikipedia.org/wiki/Service-level_agreement)

26. Single-responsibility principle (https://en.wikipedia.org/wiki/Single-responsibility_principle)
27. Saga pattern (https://microservices.io/patterns/data/saga.html)