|  |  |  |
| --- | --- | --- |
|  | УНИВЕРЗИТЕТ У НОВОМ САДУ  **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**  **У НОВОМ САДУ** |  |

Марко Драгићевић

**Развој интернет апликација у облаку: Приступи и могућности традиционалне и Serverless архитектуре са AWS услугама**

Дипломски рад

- Основне академске студије -

Нови Сад, 2023

|  |  |  |
| --- | --- | --- |
|  | УНИВЕРЗИТЕТ У НОВОМ САДУ  **ФАКУЛТЕТ ТЕХНИЧКИХ НАУКА**  21000 НОВИ САД, Трг Доситеја Обрадовића 6 | Датум: |
|  |
| **ЗАДАТАК ЗА ИЗРАДУ ДИПЛОМСКОГ (BACHELOR) РАДА** | Лист: |
| 1/1 |

*(Податке уноси предметни наставник - ментор)*

| Врста студија: | **Основне академске студије** |
| --- | --- |
| Студијски програм: | **Софтверско инжењерство и информационе технологије** |
| Руководилац студијског програма: | **проф. др Мирослав Зарић** |

|  |  |  |  |
| --- | --- | --- | --- |
| Студент: | **Марко Драгићевић** | Број индекса: | **SW 74/2019** |
| Област: | **Електротехничко и рачунарско инжењерство** | | |
| Ментор: |  | | |
|  | | | |

**НАСЛОВ ДИПЛОМСКОГ (BACHELOR) РАДА:**

|  |
| --- |
|  |

**ТЕКСТ ЗАДАТКА:**

|  |
| --- |
|  |

|  |  |
| --- | --- |
| Руководилац студијског програма: | Ментор рада: |
|  |  |

|  |
| --- |
| Primerak za:  - Studenta;  - Mentora |

# КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА

|  |  |
| --- | --- |
| Редни број, **РБР**: |  |
| Идентификациони број, **ИБР**: |  |
| Тип документације, **ТД**: | монографска публикација |
| Тип записа, **ТЗ**: | текстуални штампани документ |
| Врста рада, **ВР**: | дипломски рад |
| Аутор, **АУ**: | Петар Марковић |
| Ментор, **МН**: | др Горан Сладић, редовни професор |
| Наслов рада, **НР**: | Развој апликације студентске службе употребом блокчејн технологије |
| Језик публикације, **ЈП**: | српски |
| Језик извода, **ЈИ**: | српски / енглески |
| Земља публиковања, **ЗП**: | Србија |
| Уже географско подручје, **УГП**: | Војводина |
| Година, **ГО**: | 2022 |
| Издавач, **ИЗ**: | ауторски репринт |
| Место и адреса, **МА**: | Нови Сад, Факултет техничких наука, Трг Доситеја Обрадовића 6 |
| Физички опис рада, **ФО**: | 9 / 51 / 16 / 3 / 17 / 0 / 0 |
| Научна област, **НО**: | Електротехничко и рачунарско инжењерство |
| Научна дисциплина, **НД**: | Безбедност у системима електронског пословања |
| Предметна одредница /  кључне речи, **ПО**: | Блокчејн, Ethereum, паметни уговори |
| **УДК** |  |
| Чува се, **ЧУ**: | Библиотека Факултета техничких наука, Трг Доситеја Обрадовића 6, Нови Сад |
| Важна напомена, **ВН**: |  |
| Извод, **ИЗ**: | У раду је приказана имплементација децентрализоване апликације за студентску службу. За ову сврху, на бекенду је употребљена Hardhat развојна платформа за рад са паметним уговорима у Solidity језику.За фронтенд је коришћена *React* библиотека у комбинацији са *Moralis* платформом за комуникацију са паметним уговором. |
| Датум прихватања теме, **ДП**: |  |
| Датум одбране, **ДО**: |  |
| Чланови комисије, **КО**: |  |
| председник | др Гордана Милосављевић, ред. проф, ФТН Нови Сад |
| члан | др Мирослав Зарић, доцент, ФТН Нови Сад |
| ментор | др Горан Сладић, ванр. проф, ФТН Нови Сад |
| Потпис ментора | |

# KEY WORDS DOCUMENTATION

|  |  |
| --- | --- |
| Accession number, **ANO**: |  |
| Identification number, **INO**: |  |
| Document type, **DT**: | monographic publication |
| Type of record, **TR**: | textual material |
| Contents code, **CC**: | bachelor thesis |
| Author, **AU**: | Petar Marković |
| Mentor, **MN**: | Goran Sladić, regular professor, PhD |
| Title, **TI**: | Development of students service application using blockchain technology |
| Language of text, **LT**: | Serbian |
| Language of abstract, **LA**: | Serbian / English |
| Country of publication, **CP**: | Serbia |
| Locality of publication, **LP**: | Vojvodina |
| Publication year, **PY**: | 2022 |
| Publisher, **PB**: | author’s reprint |
| Publication place, **PP**: | Novi Sad, Faculty of Technical Sciences, Trg Dositeja Obradovića 6 |
| Physical description, **PD**: | 9 / 51 / 16 / 3 / 17 / 0 / 0 |
| Scientific field, **SF**: | Software Engineering and Information Technologies |
| Scientific discipline, **SD**: | E-Business Systems Security |
| Subject / Keywords, **S/KW**: | Blockchain, Etherium, smart contracts |
| **UDC** |  |
| Holding data, **HD**: | Library of the Faculty of Technical Sciences, Trg Dositeja Obradovića 6, Novi Sad |
| Note, **N**: |  |
| Abstract, **AB**: | The paper presents the implementation of a decentralized application for student services. For this purpose, the Hardhat development platform for working with smart contracts in the Solidity language was used on the backend. The React library was used for the frontend in combination with the Moralis platform for communication with the smart contract. |
| Accepted by sci. Board on, **ASB**: |  |
| Defended on, **DE**: |  |
| Defense board, **DB**: |  |
| president | Gordana Milosavljević, PhD, full prof, FTN Novi Sad |
| member | Miroslav Zarić, PhD, assist. prof, FTN Novi Sad |
| mentor | Goran Sladić, PhD, assoc. prof, FTN Novi Sad |
| Mentor's signature | |

**САДРЖАЈ**

[КЉУЧНА ДОКУМЕНТАЦИЈСКА ИНФОРМАЦИЈА 2](#_Toc113415530)

[KEY WORDS DOCUMENTATION 2](#_Toc113415531)

[1 Увод 2](#_Toc113415532)

[2 Централизовани и децентрализовани систем **Error! Bookmark not defined.**](#_Toc113415533)

[3 Теоријске основе **Error! Bookmark not defined.**](#_Toc113415534)

[3.1 Дистрибуирани системи 2](#_Toc113415535)

[3.2 Блокчејн 2](#_Toc113415536)

[3.3 Консензус алгоритми 2](#_Toc113415537)

[3.3.1 Доказ о раду 2](#_Toc113415538)

[3.3.2 Доказ о улогу 2](#_Toc113415539)

[3.4 Ethereum 2](#_Toc113415540)

[3.5 Светска мрежа и њене итерације 2](#_Toc113415541)

[4 Модел система 2](#_Toc113415542)

[4.1 Класни дијаграм 2](#_Toc113415543)

[4.2 Дијаграм случајева коришћења 2](#_Toc113415544)

[4.3 Дијаграм активности 2](#_Toc113415545)

[5 Имплементација система 2](#_Toc113415546)

[5.1 Иницијална поставка 2](#_Toc113415547)

[5.2 Бекенд део апликације 2](#_Toc113415548)

[5.3 Фронтенд део апликације 2](#_Toc113415549)

[5.4 Финална апликација 2](#_Toc113415550)

[6 Закључак 2](#_Toc113415551)

[7 Литература 2](#_Toc113415552)

[8 Биографија 2](#_Toc113415553)

# Увод

Савремени свет софтверског инжењерства непрестано се развија и пружа нове приступе и технологије за развој робусних, скалабилних и ефикасних веб апликација. Један од револуционарних приступа који је изазвао значајну промену у начину на који се развијају и имплементирају апликације јесте *Serverless* архитектура. Овај приступ се темељи на томе да се инфраструктурни детаљи преносе на *cloud* провајдера, омогућавајући развојним тимовима да се фокусирају на кодирање функционалности, уместо на одржавање сервера и инфраструктуре.

## Мотивација и контекст

Развој веб апликација у данашњем пословном окружењу захтева не само ефикасан развој, већ и високу скалабилност, безбедност и брзу доступност. Традиционални приступ развоју, који се често ослања на монолитне архитектуре, може донети изазове у постизању ових захтева. С друге стране, *serverless* приступ, који се заснива на извршавању функција у ОБЛАКУ\_ према потреби, обећава високу скалабилност и оптимизацију ресурса. AWS, као лидер у *cloud* технологијама, пружа широк спектар услуга које подржавају оба приступа, отварајући могућности за креативно инжењерско решење.

## Циљеви рада

Циљ овог рада је истражити AWS Serverless технологије, анализирати њихове кључне карактеристике, предности и изазове. Кроз овај рад, анализираћемо основне компоненте AWS Serverless екосистема, истражити начине на које ове технологије подржавају развој апликација високих перформанси, скалабилности и поузданости, те истражити сценарије у којима се овај приступ најбоље примењује. Фокус ће бити стављен на разумевање како AWS Lambda, API Gateway, Amazon DynamoDB и други сервиси функционишу заједно како би омогућили креирање Serverless апликација. Кроз детаљне анализе случајева и упоређивања са традиционалним приступима, настојимо осветлити кључне предности Serverless модела и дати читаоцима дубље разумевање његовог потенцијала за револуцију начина на који се софтвер дизајнира и развија.

## Структура рада

Овај рад је организован у X поглавља како би се постигли наведени циљеви. У другом поглављу ће бити представљен теоријски оквир и основни концепти *Serverless* приступа. Треће поглавље ће се фокусирати на спецификацију захтева представљену кроз UML дијаграме случајева коришћења. Четврто поглавље ће показати прототипску имплементацију, детаљно описати архитектуру кроз развој веб апликације користећи одговарајуће технологије и AWS сервисе. У закључку, пето поглавље ће сумирати резултате, истећи закључке и понудити смернице за будућа истраживања.

Кроз овај рад, постићи ћемо дубље разумевање предности и изазова *serverless* приступа у контексту развоја *web* апликација, чиме ће се пружити вредан увид за доносиоце одлука приликом одабира архитектуре за своје пројекте.

# Теоријске основе

Serverless архитектура је парадигма која трансформише начин на који размишљамо о развоју и испоруци софтверских апликација. У овом поглављу, истражићемо шире теоријске основе Serverless модела, истичући кључне принципе и предности овог приступа, пре него што детаљније размотримо како AWS сервиси доприносе овој архитектури.

## Cloud рачунарство

Да би разумели Serverless парадигму, морамо проћи кроз основне концепте *cloud* рачунарства.

*Cloud* рачунарство је концепт у ком се ресурси, као што су рачунари, складишта и сервиси, пружају као услуге преко интернета. То омогућава корисницима да приступају и користе ове ресурсе према потреби, без потребе за инвестирањем у властиту физичку структуру. *Cloud* рачунарство игра кључну улогу у оснаживању Serverless технологија. Његова способност за аутоматско скалирање, ефикасно коришћење ресурса и брзи развој чини га идеалним окружењем за имплементацију Serverless архитектуре. Комбинација ова два концепта ствара темеље за иновацију, агилност и економичност у развоју модерних апликација.

## Разумевање Serverless парадигме

Serverless парадигма се заснива на суштинској промени у начину на који размишљамо о инфраструктури и одржавању софтверских апликација. У традиционалном моделу, развој и испорука софтвера захтевају планирање, конфигурисање и управљање серверским ресурсима. У Serverless моделу, инфраструктурни детаљи постају одговорност *cloud* провајдера, омогућујући развојним тимовима да се фокусирају на кодирање пословне логике. Функције се извршавају само кад су потребне, а ресурси се аутоматски скалирају према захтевима.

У Serverless моделу, функције су основни градивни блокови. Свака функција представља изоловану јединицу која се извршава у одговору на одређене догађаје или захтеве. Ове функције су независне и могу се изводити у одвојеним окружењима, омогућавајући лакше одржавање и скалирање.

## Дефиниција Serverless архитектуре

Serverless архитектура, позната и као *Function-as-a-Service (FaaS)*, представља фундаменталну промену у начину на који размишљамо о инфраструктурним захтевима апликација. Уместо традиционалног модела са стално присутним серверима, Serverless приступ пребацује фокус на извршавање кода у одговору на догађаје (*event-driven*). То значи да се функције извршавају само када су позване, без потребе за сталним одржавањем ресурса. Овај приступ омогућава брже испоручивање функционалности и ефикасније искоришћавање ресурса, јер се ресурси алоцирају динамички према потреби.

## Предности Serverless парадигме

Serverless приступ доноси низ предности које унапређују развој, испоруку и управљање апликацијама.

### Аутоматско скалирање

Serverless апликације се аутоматски скалирају према стварним потребама, омогућавајући ефикасно управљање варијабилним оптерећењима и смањујући потребу за предвиђањем ресурса (ручним скалирањем).

### Агилност и брзина развоја

Одсуство потребе за конфигурацијом и управљањем инфраструктуром омогућава развојном тиму да се фокусира на сам код и функционалности што доноси бржи и агилнији процес израде софтвера.

### Прецизни трошкови

Serverless модел омогућава плаћање само за стварно извршено извршавање функција, чиме се смањују непотребни трошкови инфраструктуре, тј. елиминише се потреба за преплаћивањем непотребних ресурса.

## Архитектурални обрасци у Serverless парадигми

Serverless модел подржава различите архитектуралне обрасце у зависности од потреба апликације:

* **Микросервиси** – Serverless апликације се често организују као колекција микросервиса, где свака функција представља одвојени сервис са јасно дефинисаном функционалношћу. Ово омогућава скалирање и одржавање независно за сваки сервис.
* **Обрада догађаја** (*event-driven*) – Serverless приступ је посебно ефикасан за обраду догађаја из спољног света, као што су промене у базама података или пристизање порука.
* **Обрада података** (*data-driven*) – Serverless технологије омогућавају ефикасну обраду, анализу и трансформацију велике количине података.

## Изазови Serverless парадигме

Видели смо да Serverless модел доноси многе предности, али исто тако треба узети у обзир и то да се суочава са неким изазовима који захтевају пажљиво планирање и управљање:

* **Време извршавања функција**: Време одзива функција може варирати у зависности од оптерећења и конфигурације, што може утицати на перформансе апликације
* **Зависности између сервиса**: Са независним извршавањем функција, комуникација и зависности међу сервисима захтева додатно разматрање у дизајну апликације.
* **Сигурност**: Обезбеђивање сигурности у Serverless окружењу захтева пажљиво управљање правима приступа, енкрипцијом и контролом идентитета.
* **Трошкови**: Прецизни трошкови су поменути као предност Serverless парадигме, али уколико код није оптимизован или су лоше испланирани ресурси – то може довести до неочекивано високих трошкова.

Serverless архитектура мења начин на који се софтвер развија, испоручује и управља. Разумевање основних принципа, предности и изазова Serverless парадигме је суштинско за успешну примену овог приступа. У наставку, истражићемо како AWS сервиси доприносе овом концепту кроз кључне технолошке компоненте.

## Кључни AWS Serverless сервиси

У AWS Serverless екосистему, неколико кључних сервиса игра кључну улогу у креирању Serverless апликација.

### AWS Lambda

*AWS Lambda* је сервис за извршавање кода у облаку\_. *Lambda* представља функцију која се извршава у одговору на догађаје, као што је HTTP захтев. Када се функција позове, *AWS Lambda* аутоматски алоцира ресурсе и извршава код. Овај сервис омогућава развојном тиму да се фокусирају на пословну логику, док је за инфраструктурне аспекте задужен клауд провајдер.

### API Gateway

API Gateway је сервис који омогућава креирање, управљање и заштиту API-ја без потребе за управљањем инфраструктуром. Он функционише као улазна тачка за захтеве спољних клијената и повезује се са AWS Lambda функцијама или другим сервисима. API Gateway омогућава дефинисање рута, аутентикацију корисника и управљање приступом ресурсима.

### Amazon DynamoDB

Amazon DynamoDB је скалабилна и брза *NoSQL* база података. Добар је избор за Serverless апликације због своје способности скалирања према потреби. DynamoDB се интегрише са AWS Lambda функцијама како би омогућио ефикасно управљање подацима и брзу манипулацију информација.

### AWS Step Functions

AWS Step Functions омогућава дефинисање и управљање корацима и пословним процесима. Овај сервис је посебно користан за оркестрацију између различитих функција и сервиса унутар Serverless апликација. Дефинисањем корака и услова, могуће је створити сложене токове података и догађаја.

### AWS Cognito

AWS Cognito омогућава управљање идентитетима и приступом корисника апликацијама. Пружа функционалности као што су регистрација корисника, управљање аутентикацијом и ауторизацијом, омогућујући безбедан приступ Serverless апликацијама.

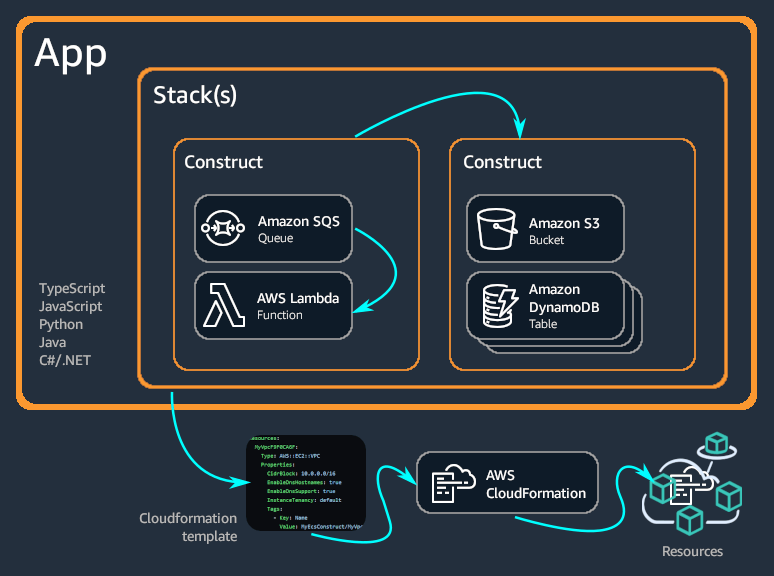
## AWS CDK

*Cloud Development Kit* (*CDK*) компаније Amazon Web Services је моћан алат који омогућава развојном тиму да дефинише и управља инфраструктуром кроз код (енгл. *Infrastructure as Code - IaC*). У овом раду је управо CDK коришћен за дефиницију инфраструктуре и видећемо како он пружа још већу агилност и ефикасност у развоју и управљању Serverless технологијама.

CDK је развојни оквир који омогућава програмирање инфраструктуре помоћу програмских језика као што су *TypeScript, Python, Java* и *C#.* Уместо писања традиционалних конфигурационих фајлова, развојни тимови користе код да дефинишу ресурсе, као што су функције, API руте, базе података и слично.

### Структура и функционалности AWS CDK

AWS CDK омогућава развојним тимовима да користе концепте као што су *Stacks* и *Constructs* за организацију и управљање инфраструктурним ресурсима, што се може видети на Слика 1:



Слика 1 - AWS CDK архитектура

* ***Stacks:*** Изолована окружења инфраструктурних ресурса. Сваки *stack* може садржати више ресурса и компонената.
* ***Constructs:*** Предефинисане компоненте које олакшавају креирање и конфигурисање ресурса. Ове компоненте омогућују поновно коришћење најбољих пракси и конвенција. *Construct* може представљати тачно један AWS ресурс, нпр. *Amazon S3*, али исто тако може бити на вишем нивоу апстракције и садржати више повезаних AWS ресурса.

Неке од предности коришћења AWS CDK-а су: смањење грешака, бржа итерација, боља контрола и прилагодљивост ресурса.

# Спецификација захтева

У овом поглављу, фокусираћемо се на спецификацију захтева и демонстрацију могућности Serverless веб апликације коју смо развили за управљање запосленима, тимовима и пројектима у оквиру компаније. Прво ћемо детаљно описати сврху апликације и функционалности које пружа, а затим ћемо користити опис случајева коришћења како бисмо приказали интеракцију корисника са апликацијом.

Циљ ове веб апликације јесте пружити ефикасан начин управљања запосленима, тимовима и пројектима унутар компаније. Главне функционалности укључују додавање, преглед, ажурирање и брисање података о запосленима, тимовима и пројектима. Апликација омогућава менаџерима да лако организују своје тимове, прате напредак пројеката и ефикасно управљају ресурсима.

## Опис случајева коришћења

Опис случајева коришћења ће нам помоћи да боље разумемо интеракције између корисника и система.

### Пријава на систем

|  |  |
| --- | --- |
| Use Case Name: | Пријава на систем |
| Description: | Овај случај коришћења омогућава пријаву на систем како би менаџер имао приступ свим својим функцијама у систему. Корисник се пријављује на систем тако што уноси одговарајући *email* и лозинку. |
| Primary Actor: | Менаџер |
| Preconditions: | 1. Менаџер треба да има валидан налог |
| Postconditions: | 1. Систем приказује почетну страницу |
| Маin Flow: | 1. Менаџер уноси *email* и лозинку 2. Систем врши валидацију поља 3. Систем проверава унете креденцијале 4. Систем приказује менаџерову почетну страницу |
| Alternative Flows: | 2a – Email или лозинка нису унети   1. Систем приказује поруку о недостатку *email-a* и/или лозинке 2. Случај коришћења наставља извршавање на кораку 1) главног тока   3а – Нетачни креденцијали   1. Систем приказује поруку о неисправности унетих креденцијала 2. Случај коришћења наставља извршавање на кораку 1) главног тока |

Табела 1 - Use Case - пријава на систем

### Регистрација

|  |  |
| --- | --- |
| Use Case Name: | Регистрација на систем |
| Description: | Овај случај коришћења омогућава регистрацију на систем како би менаџер имао могућност пријаве и обављања жељених функција. Корисник се пријављује на систем тако што уноси име, презиме, *email*, и лозинку. Након уноса података, корисник добија код на унету *email* адресу са кодом којим верификује исту. |
| Primary Actor: | Менаџер |
| Preconditions: | Менаџер је запослен у компанији |
| Postconditions: | Систем приказује страницу за пријаву на систем |
| Main Flow: | 1. Менаџер уноси име, презиме, *email* и лозинку 2. Потврђује форму и шаље податке систему 3. Систем врши валидацију података 4. Систем шаље верификациони код на унети *email* 5. Менаџер уноси код у одговарајуће поље на страници 6. Менаџеру се приказује страница за пријаву на систем |
| Alternative Flow: | 3а – Нису унети сви подаци за регистрацију   1. Систем приказује поруку о недостајућим подацима 2. Случај коришћења наставља извршавање на кораку 1) главног тока   *3б –* У систему већ постоји унети *email*   1. Систем приказује поруку о већ постојећем кориснику са унетим *email-*ом 2. Случај коришћења наставља извршавање на кораку 1) главног тока   *5а –* Корисник уноси погрешан код за верификацију *email-а*   1. Систем приказује поруку о погрешном коду за верификацију 2. Случај коришћења наставља извршавање на кораку 5) |

Табела 2 - Use Case – регистрација

### Креирање новог тима

|  |  |
| --- | --- |
| Use Case Name: | Креирање новог тима |
| Description: | Овај случај коришћења омогућава менаџеру да креира нове тимове, како би поставио одређену групу запослених на пројекте. |
| Primary Actor: | Менаџер |
| Preconditions: | Менаџер је пријављен на систем |
| Postconditions: | Нови тим је доступан за пројекте и запослене |
| Main Flow: | 1. Менаџер уноси податке о новом тиму: назив тима 2. Менаџер бира вођу тима (engl. *Team Lead*) 3. Менаџер додаје чланове тима из табеле запослених 4. Менаџер потврђује форму и шаље податке о новом тиму систему 5. Систем врши валидацију 6. Систем показује поруку о успешности 7. Менаџеру се приказује страница за преглед свих тимова |

Табела 3 - Use Case - креирање новог тима

### Измена тима

|  |  |
| --- | --- |
| Use Case Name: | Измена тима |
| Description: | Овај случај коришћења омогућава менаџеру да мења постојеће тимове, односно да промени назив тима, вођу, као и чланове тима. |
| Primary Actor: | Менаџер |
| Preconditions: | Менаџер је пријављен на систем |
| Postconditions: | Измењен тим је доступан за пројекте и запослене |
| Main Flow: | 1. Менаџер уноси нови назив тима уколико жели да га измени 2. Менаџер бира вођу тима 3. Менаџер може да склони вођу тима са тренутне позиције 4. Менаџер додаје чланове тима из табеле запослених 5. Менаџер може да избаци запослене који су већ чланови тима 6. Менаџер потврђује форму и шаље податке о новом тиму систему 7. Систем врши валидацију 8. Систем приказује поруку о успешности 9. Менаџеру се приказује страница за преглед свих тимова |

Табела 4 - Use Case - измена тима

### Брисање тима

|  |  |
| --- | --- |
| Use Case Name: | Брисање тима |
| Description: | Овај случај коришћења омогућава менаџеру да брише постојеће тимове, како не би били доступни за нове запослене и пројекте у будућности. |
| Primary Actor: | Менаџер |
| Preconditions: | Менаџер је пријављен на систем |
| Postconditions: | Избрисан тим није доступан за нове запослене и пројекте |
| Main Flow: | 1. Менаџер из листе свих тимова селектује тим који жели 2. Систем му нуди опцију за брисање 3. Менаџер потврђује опцију за брисање 4. Систем приказује поруку о успешности 5. Менаџеру се приказује страница за преглед свих тимова |

Табела 5 - Use Case - брисање тима

### Додавање новог запосленог

|  |  |
| --- | --- |
| Use Case Name: | Додавање новог запосленог |
| Description: | Овај случај коришћења омогућава менаџеру да креира новог запосленог како би постао доступан за нове пројекте и тимове. Сваки запослени треба да има: име, презиме, адресу, позицију и сениоритет. |
| Primary Actor: | Менаџер |
| Preconditions: | Менаџер је пријављен на систем |
| Postconditions: | Нови запослени је доступан за пројекте и тимове |
| Main Flow: | 1. Менаџер уноси податке о новом запосленом: име, презиме, адреса становања, позиција, сениоритет, *email* 2. Менаџер може да изабере тим ком ће запослени припадати 3. Систем врши валидацију 4. Систем приказује поруку о успешности 5. Менаџеру се приказује страница за преглед свих запослених |

Табела 6 - Use Case - додавање новог запосленог

### Измена запосленог

|  |  |
| --- | --- |
| Use Case Name: | Измена запосленог |
| Description: | Овај случај коришћења омогућава менаџеру измени податке изабраног радника. Менаџер има опцију да избаци члана из тренутног, или га додели неком другом тиму. |
| Primary Actor: | Менаџер |
| Preconditions: | Менаџер је пријављен на систем |
| Postconditions: | Радник је ажуриран унетим подацима и тимом |
| Main Flow: | 1. Менаџер уноси податке које жели да измени: име, презиме, адреса становања, позиција и сениоритет 2. Менаџер може да избаци радника из тренутног тима 3. Менаџер може да додели запосленом други тим 4. Систем врши валидацију 5. Систем приказује поруку о успешности 6. Менаџеру се приказује страница за преглед свих запослених |

Табела 7 - Use Case - измена запосленог

### Преглед запосленог

|  |  |
| --- | --- |
| Use Case Name: | Преглед запосленог |
| Description: | Овај случај коришћења омогућава менаџеру да на посебној страници има детаљан приказ информација о запосленом. Информације које су доступне су: име, презиме, адреса, *email*, позиција, тренутни тим, тренутни пројекат, сениоритет, као и сви пројекти на којима је дати запослени радио у прошлости. |
| Primary Actor: | Менаџер |
| Preconditions: | Менаџер је пријављен на систем |
| Postconditions: | / |
| Main Flow: | 1. Менаџер са странице за преглед свих запослених селектује одређеног запосленог 2. Систем добавља податке о запосленом 3. Менаџеру се приказује страница са детаљним приказом изабраног радника (име, презиме, адреса, *email*, сениоритет, позиција, тренутни тим, тренутни пројекат, табела са свим пројектима на којима је радио у прошлости |

Табела 8 - Use Case - преглед запосленог

### Брисање запосленог

|  |  |
| --- | --- |
| Use Case Name: | Брисање тима |
| Description: | Овај случај коришћења омогућава менаџеру да брише постојеће запослене, како не би били доступни за тимове и пројекте у будућности. |
| Primary Actor: | Менаџер |
| Preconditions: | Менаџер је пријављен на систем |
| Postconditions: | Избрисан радник није доступан за тимове и пројекте |
| Main Flow: | 1. Менаџер из листе свих запослених селектује радника 2. Систем му нуди опцију за брисање 3. Менаџер потврђује опцију за брисање 4. Систем приказује поруку о успешности 5. Менаџеру се приказује страница за преглед свих запослених |

Табела 9 - Use Case - брисање запосленог

### Креирање новог пројекта

|  |  |
| --- | --- |
| Use Case Name: | Креирање новог пројекта |
| Description: | Овај случај коришћења омогућава менаџеру да креира нове пројекте. |
| Primary Actor: | Менаџер |
| Preconditions: | Менаџер је пријављен на систем |
| Postconditions: | Нови пројекат је сачуван и омогућено ја праћење напретка |
| Main Flow: | 1. Менаџер уноси податке о новом пројекту: назив пројекта, податке о клијенту, датум почетка рада на пројекту, предвиђен датум завршетка рада на пројекту, опис и статус пројекта 2. Менаџер може изабрати тим који ће бити задужен за пројекат 3. Систем врши валидацију 4. Систем приказује поруку о успешности 5. Менаџеру се приказује страница за преглед свих пројеката |

Табела 10 - Use Case - креирање новог пројекта

### Измена пројекта

|  |  |
| --- | --- |
| Use Case Name: | Измена пројекта |
| Description: | Овај случај коришћења омогућава менаџеру измени изабрани пројекат. Менаџер има опцију да измени тим који је задужен за пројекат. |
| Primary Actor: | Менаџер |
| Preconditions: | Менаџер је пријављен на систем |
| Postconditions: | Пројекат је ажуриран унетим подацима и тимом |
| Main Flow: | 1. Менаџер уноси податке које жели да измени: назив, подаци о клијенту, датум почетка и завршетка рада, статус пројекта 2. Менаџер може да додели пројекат другом тиму 3. Систем врши валидацију 4. Систем приказује поруку о успешности 5. Менаџеру се приказује страница за преглед свих пројеката |

Табела 11 - Use Case - измена пројекта

### Преглед пројекта

|  |  |
| --- | --- |
| Use Case Name: | Преглед пројекта |
| Description: | Овај случај коришћења омогућава менаџеру да на посебној страници има детаљан приказ информација о пројекту. Информације које су доступне су: назив пројекта, подаци о клијенту, статус пројекта, тим, вођа тима, датум почетка и завршетка рада, опис пројекта као и сви запослени који тренутно раде на пројекту. |
| Primary Actor: | Менаџер |
| Preconditions: | Менаџер је пријављен на систем |
| Postconditions: | / |
| Main Flow: | 1. Менаџер са странице за преглед свих пројеката селектује одређени пројекат 2. Систем добавља податке о пројекту 3. Менаџеру се приказује страница са детаљним приказом изабраног пројекта (назив, опис, статус пројекта, тим, вођа тима, подаци о клијенту, датум почетка и завршетка рада и табела са запосленима који тренутно раде на пројекту |

Табела 12 - Use Case - преглед пројекта

### Брисање пројекта

|  |  |
| --- | --- |
| Use Case Name: | Брисање пројекта |
| Description: | Овај случај коришћења омогућава менаџеру да брише постојеће пројекте, како не би били доступни тимовима у будућности. |
| Primary Actor: | Менаџер |
| Preconditions: | Менаџер је пријављен на систем |
| Postconditions: | Избрисан радник није доступан за тимове и пројекте |
| Main Flow: | 1. Менаџер из листе свих пројеката селектује пројекат 2. Систем му нуди опцију за брисање 3. Менаџер потврђује опцију за брисање 4. Систем приказује поруку о успешности 5. Менаџеру се приказује страница за преглед свих пројеката |

Табела 13 - Use Case - брисање пројекта

### Преглед тима

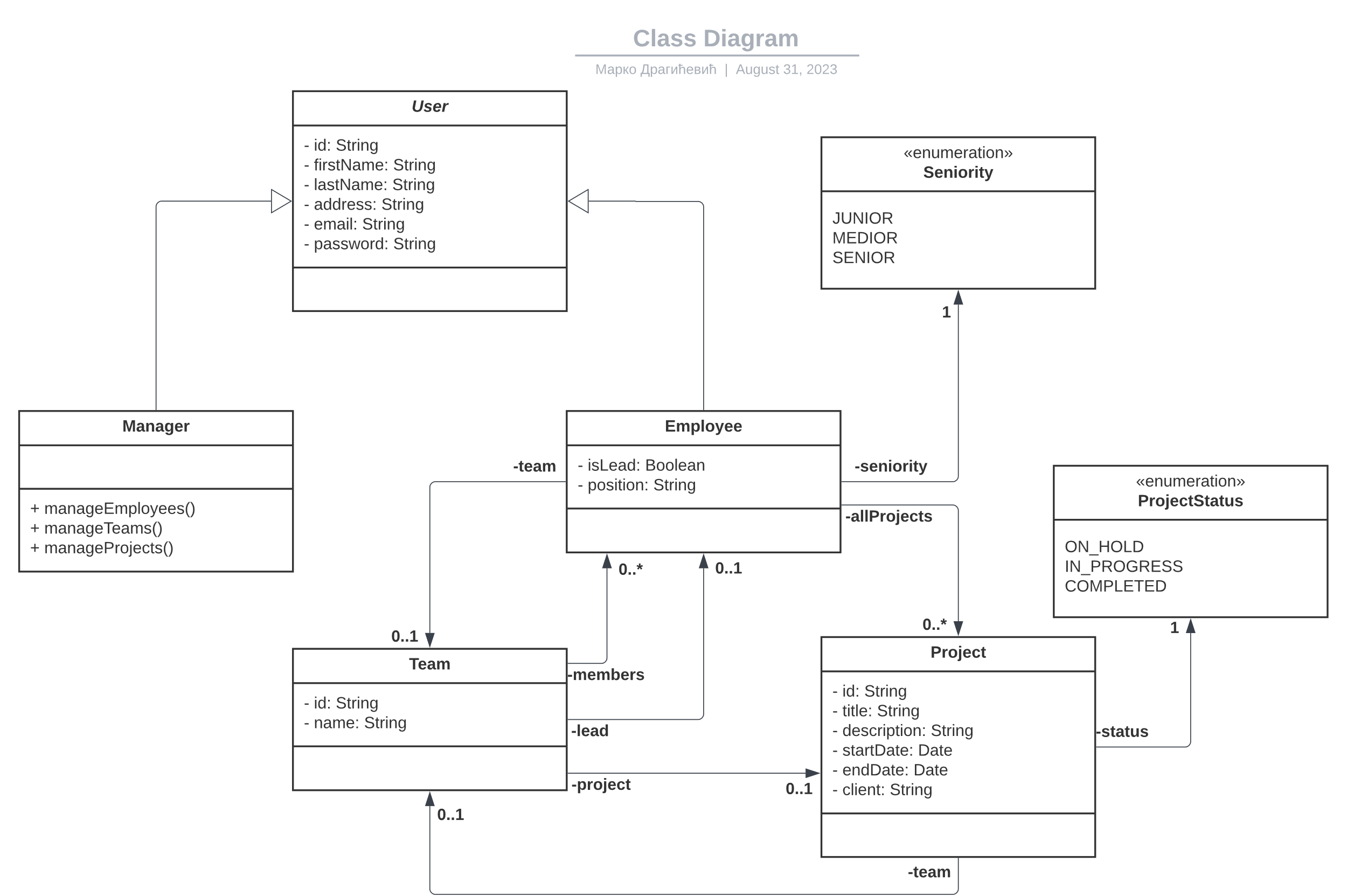
|  |  |
| --- | --- |
| Use Case Name: | Преглед тима |
| Description: | Овај случај коришћења омогућава менаџеру да на посебној страници има детаљан приказ информација о тиму. Информације које су доступне су: назив тима, вођа тима, пројекат на коме тим тренутно ради, број чланова тима, као и табеларни приказ свих чланова. |
| Primary Actor: | Менаџер |
| Preconditions: | Менаџер је пријављен на систем |
| Postconditions: | / |
| Main Flow: | 1. Менаџер са странице за преглед свих тимова селектује одређени тим 2. Систем добавља податке о тиму 3. Менаџеру се приказује страница са детаљним приказом изабраног тима (назив, вођа тима, пројекат на ком тим ради, број чланова тима и табела са запосленима који тренутно раде у тиму |

Табела 14 - Use Case - преглед тима

# Прототипска имплементација

У овом поглављу, детаљно је описан начин на који је имплементирана Serverless веб апликација за управљање запосленима, тимовима и пројектима. Приказано је како су коришћени различити AWS сервиси да би се омогућило ефикасно функционисање апликације. Такође, пажња је посвећена архитектури апликације, моделу података и размотрени су неки кључни аспекти имплементације, укључујући фотографије корисничког интерфејса.

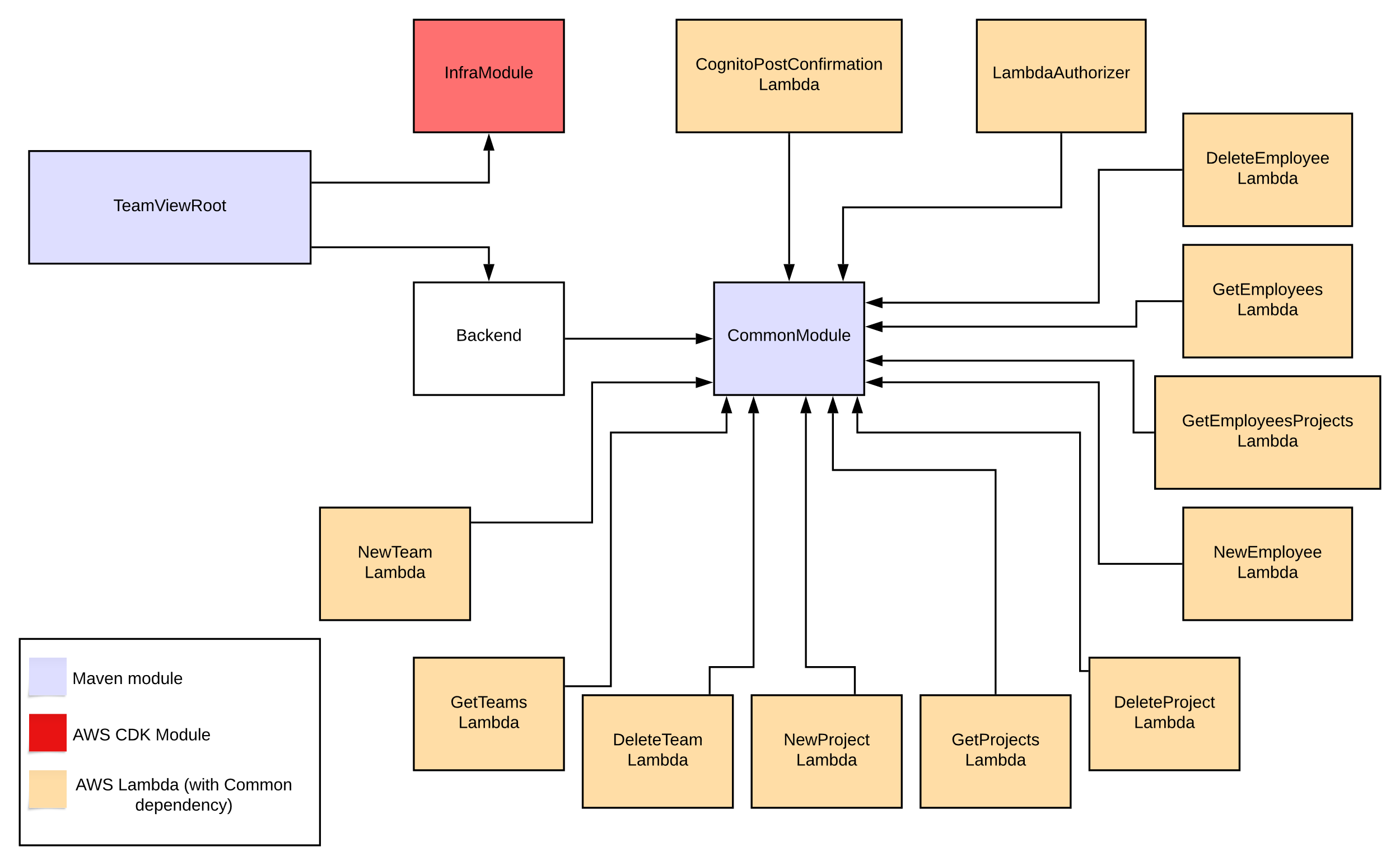
## Модел података



Класни дијаграм је тип дијаграма статичке структуре коришћен зарад описа система преко класа, њених атрибута и метода, али и односа између објеката. На xta приказан је класни дијаграм. Класе у овом дијаграму су:

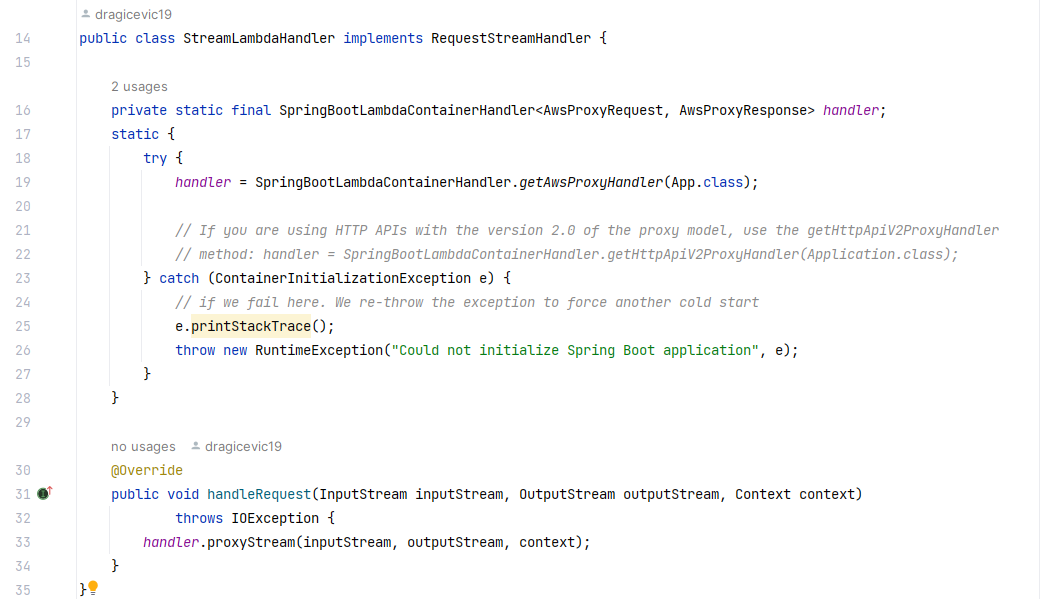
* ***User*** *–* први атрибут класе *User* представља идентификациони низ карактера јединствен за сваког члана компаније. Поред овог атрибута, класа *User* садржи и атрибуте за: име, презиме, адресу становања и *email* и лозинку. *User* је надкласа двема класама:
  + ***Manager*** *–* нема додатних атрибута, али за разлику од запосленог (енгл. *Employee)* има могућност додавања, прегледа, брисања и измене запослених, тимова и пројеката
  + ***Employee*** *–* поред атрибута из класе *User,* ова класа садржи и два додатна: *isLead* атрибут нам говори да ли је одређени запослени вођа тима, док атрибут *position* описује позицију на радном месту (нпр. *Java Backend Engineer).* Поред позиције, сваки запослени поседује одређени сениоритет (енгл. *Seniority).* Атрибут *team* нам указује на то у ком се тиму тренутно налази запослени, док *allProject* садржи све пројекте на којима је запослени радио у прошлости.
* ***Team*** *–* класа представља један тим који има свој идентификатор (*id*), као и назив тима (*name*). Веза ка класи *Employee* указује на чланове тима (*members*), као и на вођу тима (lead). Сваки тим може радити на највише једном пројекту у истом тренутку (*project).*
* ***Project*** *–* сваки пројекат има свој идентификатор (*id*), наслов пројекта (*title*), опис (*description*), почетни и завршни датум рада (*startDate* и *endDate*), клијента (*client*) за кога се пројекат ради, статус (*status)*, као и тим који тренутно ради на пројекту (*team*).

## Структура пројекта

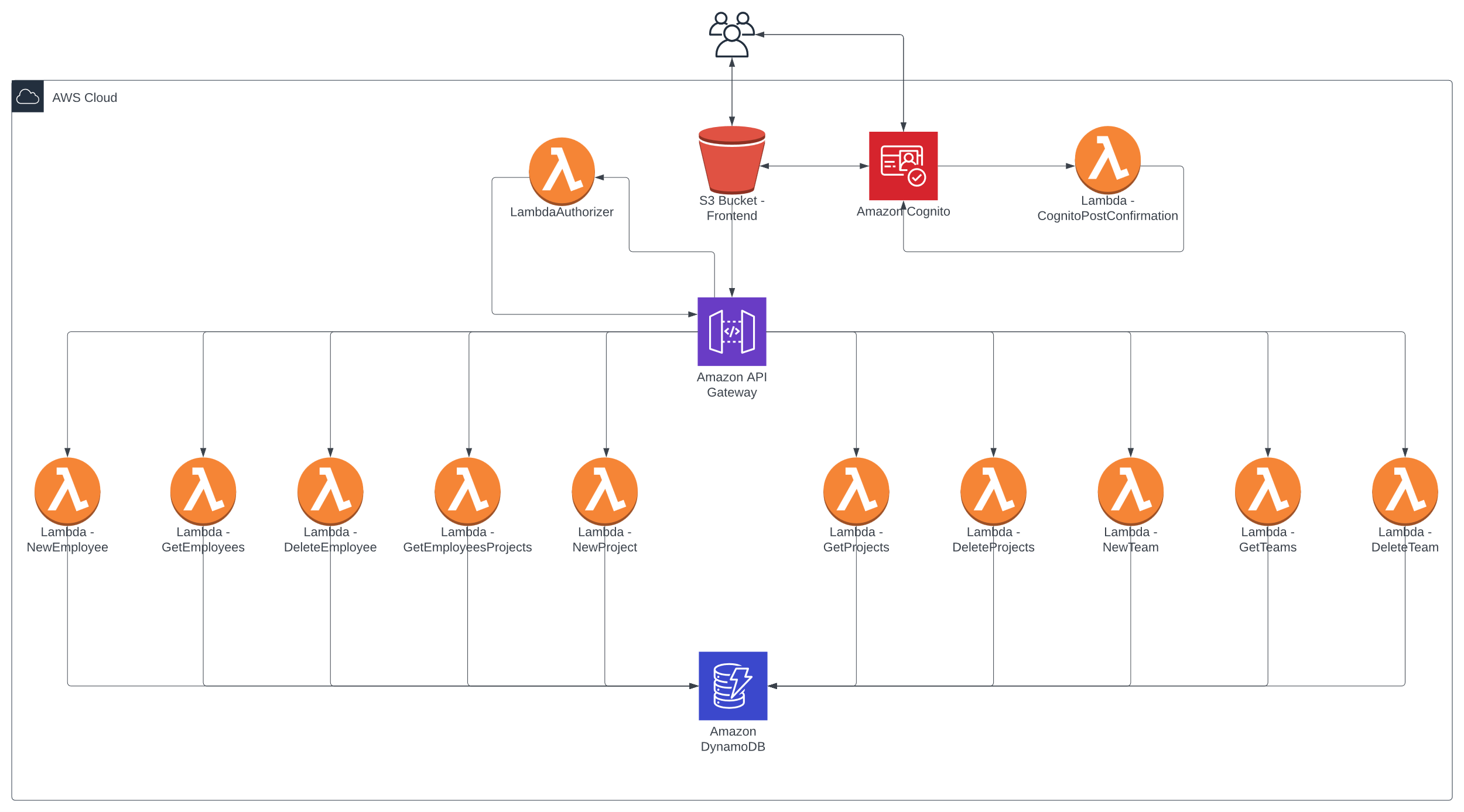


*Backend* део апликације је организован као *multi-module* пројекат. У наредним пасусима биће објашњена слика џта, како је код распоређен да би се постигло најефикасније и најчистије решење, без понављања кода.

* ***TeamViewRoot*** – модул представља улазну тачку, односно корен у стаблу модула. Састоји се из два дела: *infra* модул и *backend.*
* ***Infra*** – овај модул је резултат команде *cdk init*. Служи за дефинисање и конфигурацију читаве AWS инфраструктуре. Састоји се из две класе: *InfraApp* која садржи *main* функцију и улазна је тачка програма, као и *InfraStack* која служи за дефиницију и конфигурацију AWS сервиса, односно наше инфраструктуре. Резултат овог модула јесте *InfraStack* који ће бити доступан на AWS сервису под називом *AWS Cloud Formation,* а то се постиже командом *cdk deploy.*
* ***Backend*** фолдер садржи сву бизнис логику, модел података и све што је потребно за функционисање ове веб апликације.
* ***Common*** модул служи за смештање свих заједничких делова кода које *Lambda* функције користе. Родитељски модул му је *TeamViewRoot. Common* модул је увучен као зависност (енгл. *dependency)* у сваки од модула за *Lambda* функције. Садржи модел података, *DTO* класе, класе са рад са изузецима (енгл. exceptions), као и класу за комуникацију са *DynamoDB* базом података.
* ***Lambda*** функције – свака *Lambda* функција представља једну *Spring Boot* апликацију. Свака од функција одговара на неки од захтева апликације, врши ауторизацију и комуницира са базом података. Да би се смањило понављање кода у свакој од *Lambda* функција, заједничке ствари као што су класе модела, *DTO* класе и класе за рад са базом података су увезене као зависност из *Common* модула. Свака *Lambda* садржи *Handler* функцију, која представља *proxy* између *API Gateway-*а и *Spring Boot* контролера. Пример једне *Handler* функције која иницијализује *Spring Boot* апликацију је приказан на слици џцв.



## AWS архитектура



У овом поглављу истражени су детаљи AWS архитектуре која је коришћена за имплементацију Serverless веб апликације. Уводно је размотрена важност правилног избора AWS сервиса и њихова међусобна интеграција како би се постигла оптимална функционалност и перформансе апликације. Након тога, анализиран је сваки од кључних AWS сервиса које смо користили: *Amazon S3, AWS Cognito, API Gateway, AWS Lambda* и *Amazon DynamoDB.* Сваки од ових сервиса доприноси јединственим аспектима Serverless архитектуре и омогућава апликацији да функционише беспрекорно и ефикасно.

### Увод у AWS архитекту

AWS (*Amazon Web Services*)пружа широк спектар услуга које омогућавају креирање и управљањеразличитим компонентама апликације у облаку. Правилно планирање и избор одговарајућих AWS сервиса кључно је за постизање скалабилности, перформанси и поузданости Serverless апликације. Апликација за управљање запосленима, тимовима и пројектима користи низ AWS сервиса који се комбинују како би омогућили брз развој, ефикасну комуникацију и сигурну употребу.

### Amazon S3



*Amazon S3 (Simple Storage Service)* је сервис који је у овом раду коришћен за хостовање статичких фајлова корисничког интерфејса. Ова компонента омогућава брз и поуздан приступ овим фајловима корисницима широм света. За потребе апликације, *frontend* део је рађен помоћу *Angular-*а. У наставку је приказан начин на који је она постављена на *Amazon S3*.

Први корак у хостовању апликације на *S3* је креирање *S3 bucket-*а. Ово је једноставан процес и може се обавити путем AWS конзоле или кроз AWS CDK. У овом раду, коришћен је CDK за динамичко креирање и конфигурисање S3 bucket-а како би се аутоматизовали процеси развоја и *deploy-*а.

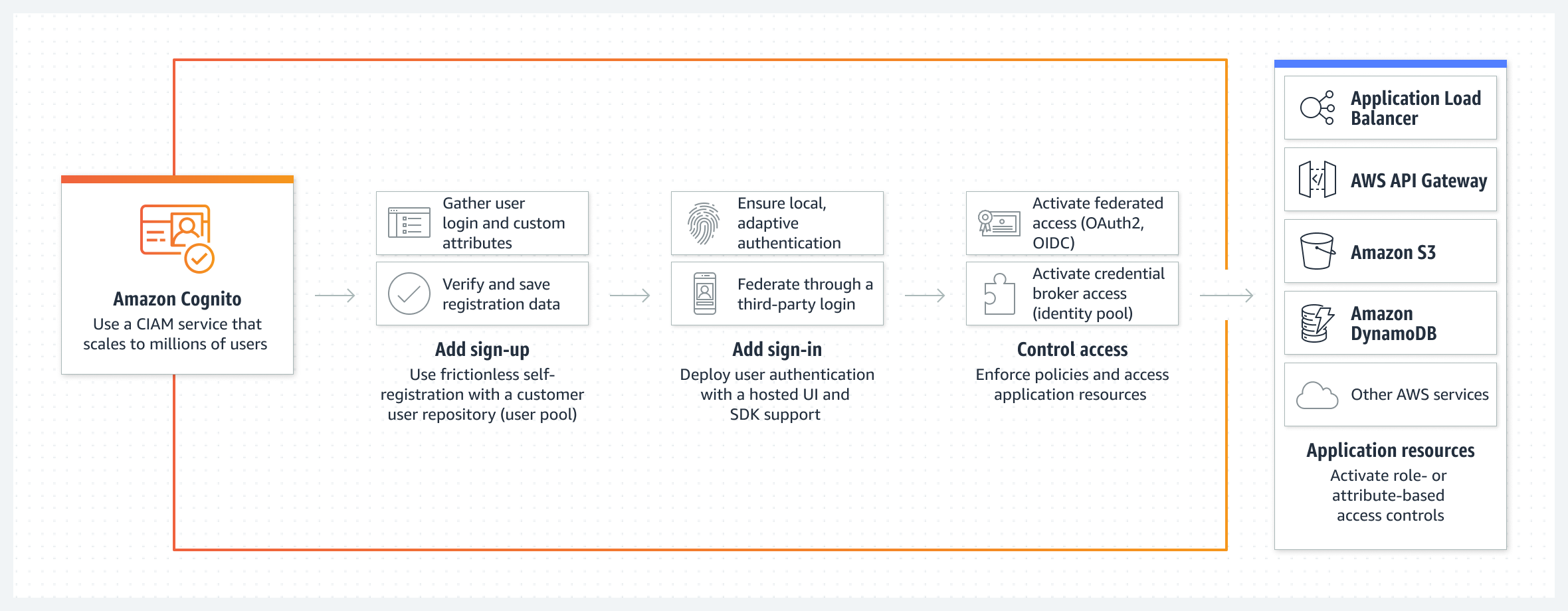
Организација фајлова унутар *S3 bucket-*а је важна за лако управљање и одржавање апликације. *Dist* фолдер, који је генерисан након извршавања ***npm build*** команде, садржи све потребне фајлове за *Angular* апликацију. Ови фајлови су постављени унутар *S3 bucket-*а тако да им се лако може приступити путем одговарајућих URL-ова.



Подешавање путање (енгл. *routing*) на *S3* је важно како би се омогућило прослеђивање свих захтева корисника ка индекс фајлу – *index.html* Angular апликације. Путање се прослеђују *Angular* апликацији, што омогућава да се рута обрађује на страни клијента.

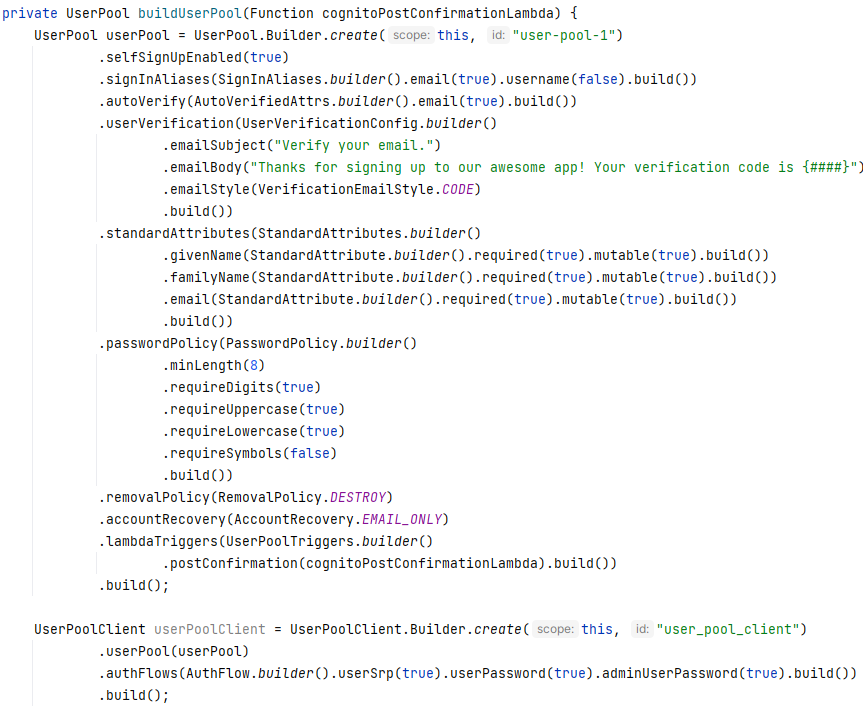
Хостовање *Angular* апликације на *Amazon S3* представља ефикасан и скалабилан начин за достављање *frontend* апликације корисницима. Овај приступ омогућава брз приступ и поузданост, уз једноставно управљање и скалабилност. Коришћење AWS CDK за креирање и конфигурисање *S3 bucket-*а доприноси аутоматизацији *deploy-*а и одржавања апликације. Ово је кључни део наше *Serverless* архитектуре који омогућава да апликација буде доступна и функционална за кориснике.

### AWS Cognito



*AWS Cognito* је сервис који пружа аутентикацију, ауторизацију и управљање корисничким идентитетима. Овај сервис омогућава корисницима да се сигурно пријављују и региструју на систем, док истовремено пружа механизме за контролу приступа различитим ресурсима. У оквиру овог поглавља, истражена је конфигурација аутентикације корисника и управљање ауторизацијом кроз права приступа.

Први корак у постављању *AWS Cognito* сервиса за апликацију јесте креирање *User Pool-*a. *User Pool* представља централно место за управљање корисничким идентитетима, аутентикацију и атрибутима корисника. Коришћен је AWS CDK за креирање *User Pool*-а, што је омогућило аутоматизацију овог процеса и чување конфигурације кроз код који је приказан на слици џцв.



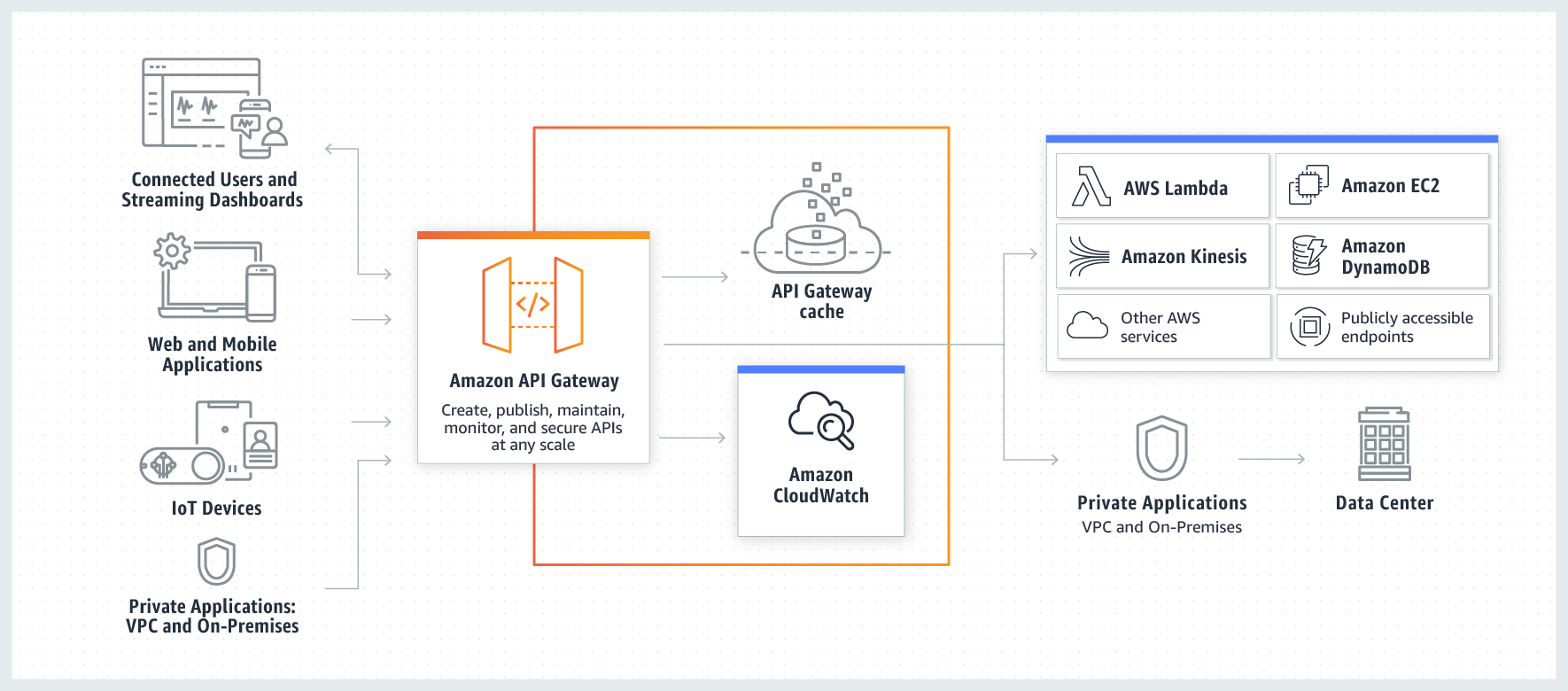
У делу кода са слике цвб је извршена конфигурација *AWS Cognito User Pool-*а и дефинисане су три корисничке групе за аутентикацију и ауторизацију корисника у оквиру апликације.

Креиран је *User Pool,* централна компонента за управљање корисницима и аутентикацију. Постављене су различите опције, укључујући могућност регистрације корисника, дефинисање атрибута који се користе за пријаву (*email* и лозинка), верификацију путем *email-*а, дефинисање стандардних атрибута за сваког корисника (име, презиме, *email*), правила за лозинке и друге поставке за сигуран рад система. *User Pool Client* се користи за комуникацију са *User Pool-*ом и подржава различите аутентификационе токове, укључујући SRP (енгл. *Secure Remote Password*) и *User/Password* аутентикацију.

Након тога, креиране су три корисничке групе – *EmployeeGroup (*група за запослене), *TeamLeadGroup (*група којој припадају вође тимова) и *AdminGroup (*група којој припада менаџер компаније). Корисничке групе представљају различите улоге у систему. Помоћу група је имплементирана ауторизација, о чему ће бити речи у наставку.

Ова конфигурација омогућава апликацији да ефикасно управља идентитетима, аутентикацијом и ауторизацијом корисника, омогућавајући приступ ресурсима у складу са улогом корисника у систему. Доступне су следеће функционалности: **пријава** на систем, **регистрација** на систем, **верификација** *email* адресе након регистрације, као и **промена** заборављене **лозинке**.

### API Gateway



*AWS API Gateway* омогућава креирање, публикацију и управљање API-има који омогућавају комуникацију између корисничког интерфејса и *Lambda* функција. Овај сервис омогућава дефинисање рута, валидацију захтева и контролу ауторизације. У делу посвећеном *API Gateway-*у, описана је конфигурација рута за различите функције и како је обезбеђена безбедна комуникација између корисничког интерфејса и *Lambda* функција.

Креирање *API Gateway-*а представља кључни корак у постављању *Serverless* архитектуре. Кроз *AWS CDK*, динамички смо креирали *RESTful API* који ће омогућити комуникацију између клијента и *backend* функционалности. На слици џцв је приказано креирање ресурса за API.



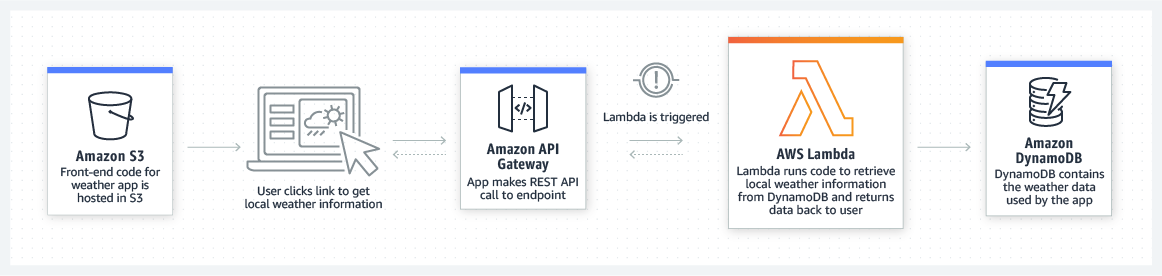
Након креирања API-ја, додати су ресурси који представљају путање за приступ различитим деловима апликације. Кроз додавање ресурса, дефинисана је структура за приступ различитим функционалностима. На пример, */employees* ресурс може водити до функционалности које се односе на запослене у систему, док */projects* ресурс води до операција везаних за пројекте.

Затим су дефинисане методе за сваки ресурс, као што су GET, POST, PUT и DELETE, како бисмо омогућили различите акције над тим ресурсима. Свака метода је повезана са одређеном *Lambda* функцијом која извршава одговарајуће операције. Ово омогућава прецизно управљање функционалностима које су доступне путем API-ја. За сваку методу је постављен *Lambda Custom Authorizer*, *Lambda* функција помоћу које се врши ауторизација, односно контрола приступа заснована на улози у систему.

|  |  |  |
| --- | --- | --- |
| *Ресурс* | *HTTP метода* | *функционалност* |
| /employees | GET | Добављање свих запослених |
| /employees | POST | Креирање новог запосленог |
| /employees | PUT | Измена запосленог |
| /employees | DELETE | Брисање запосленог |
| /teams | GET | Добављање свих тимова |
| /teams | POST | Креирање новог тима |
| *Ресурс* | ***HTTP метода*** | ***функционалност*** |
| */teams* | PUT | Измена тима |
| /teams | DELETE | Брисање тимова |
| /projects | GET | Добављање свих пројеката |
| /projects | POST | Креирање новог пројекта |
| /projects | PUT | Измена пројекта |
| /projects | DELETE | Брисање пројекта |
| /user-projects | GET | Добављање свих пројеката за одређеног запосленог |

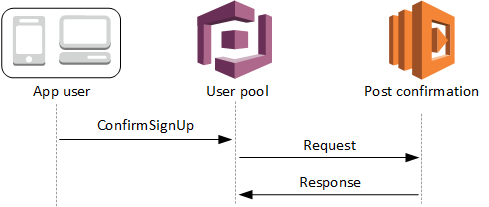
Табела 15 - Приказ свих endpoint-а

### AWS Lambda



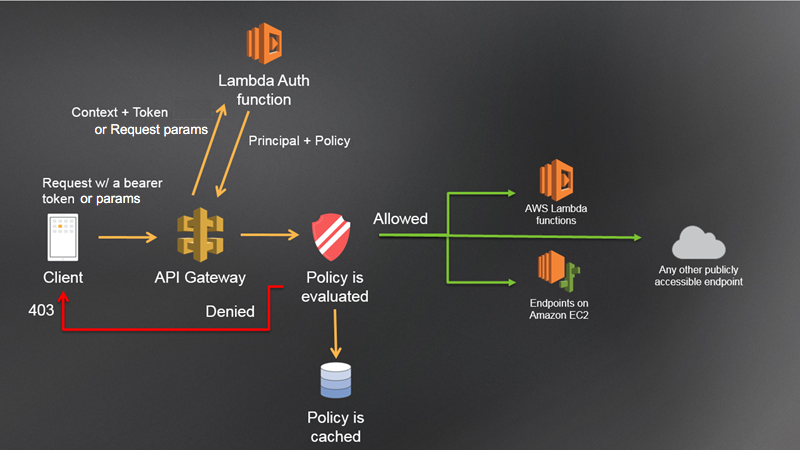
*AWS Lambda* је кључна компонента *Serverless* архитектуре која омогућава извршавање функционалности апликације без потребе за управљањем серверском инфраструктуром. *Lambda* функције се извршавају само када су потребне, аутоматски скалирају и наплаћују се само за време извршавања. Ова флексибилност чини *Lambda* функције идеалним избором за *Serverless* апликације. У оквиру овог рада, имплементиран је низ *Lambda* функција које обављају различите функционалности апликације.

1. **CognitoPostConfirmation Lambda**

****

Једна од кључних *Lambda* функција у апликацији јесте *CognitoPostConfirmation Lambda.* Ова функција се аутоматски извршава након што се нови менаџер региструје, односно унесе код који добија на *email* након регистрације. Њена сврха је да аутоматски убаци новог менаџера у одговарајућу *Cognito* корисничку групу, у овом случају *AdminGroup.* Ово осигурава да нови менаџер одмах добије одговарајуће привилегије за приступ одређеним ресурсима апликације.

1. **LambdaAuthorizer**

****

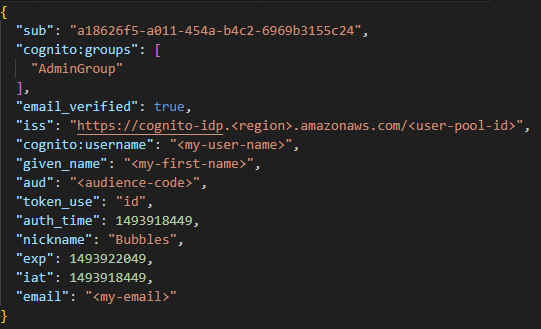
Како бисмо осигурали сигуран приступ нашим *endpoint-*има, имплементирали смо *Custom Lambda Authorizer.* Ова функција извлачи информације о корисничким групама из JWT (енгл. *JSON Web Token*) који се доставља приликом сваког захтева. На основу ових информација, *authorizer* генерише *policy* документе који прецизно дефинишу права приступа корисника за сваки ресурс. Ова контрола омогућава нам да прецизно управљамо ауторизацијом, чиме се осигурава да само корисници са одговарајућим привилегијама могу да приступе ресурсима.

У овом раду, ауторизација је заснована на улогама у систему. Свака улога је представљена једном *Cognito* корисничком групом. Сваки корисник система може припадати само једној групи.

Прво се проверава да ли је корисник пријављен, односно да ли је прослеђен JWT. Уколико није, *Authorizer* враћа грешку *Unauthorized 401,* што је приказано на слици џцв:

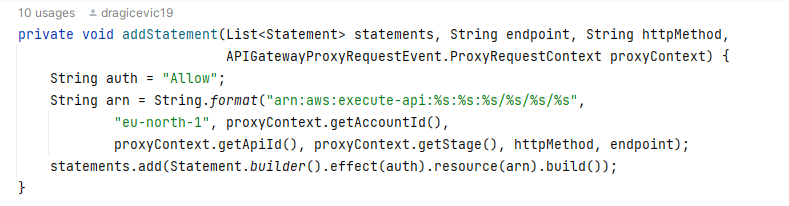


Структура JWT-а је приказана на слици џцв. Када је токен присутан, обрада се наставља. Прво се проверава којој групи корисник припада – атрибут *cognito:groups* у токену. Корисник може припадати једној од три групе које представљају улоге у систему: *EmployeeGroup, LeadGroup* или *AdminGroup.* Затим се у зависности од групе креирају полисе, односно правила, која кориснику омогућавају приступ сервисима доступним за његову улогу, тј. групу. Овај процес је приказан на слици џцв.





За сваки од *endpoint-*а креира се *Statement* у полиси (енгл. *Policy*) која обезбеђује кориснику приступ. Сваки *Statement* може да дозволи (енгл. *Allow*), или забрани (енгл. *Deny)* приступ датом *endpoint-*у, што је приказано на слици џцв. У овом раду, кориснику је дозвољен приступ само оним *endpoint-*има за које му полиса експлицитно то омогућава, у осталим случајевима клијент ће добити грешку *403 Forbidden.*



**Кеширање** (енгл. *caching*) код *Lambda Authorizer-*а је моћна техника која може побољшати перформансе и смањити оптерећење на *Serverless* систему. *Lambda Authorizer* се може оптимизовати коришћењем кеширања како би се смањило поновно извршавање ауторизације за исте захтеве од истих корисника у кратком временском периоду. Када корисник покуша да приступи одређеном *endpoint-*у, *Lambda Authorizer* се покреће како би ауторизовао захтев. Након што је ауторизација успешно извршена, резултати ауторизације се могу кеширати са одређеним временским ограничењем. Када следећи захтев стигне до истог корисника или за исти ресурс унутар временског оквира које је конфигурисано за кеширање, *Lambda Authorizer* може брзо вратити претходно кеширани резултат уместо поновног извршавања комплетне ауторизације. Ово смањује време одговора за корисника и смањује се оптерећење на самој *Lambda* функцији.

1. ***Lambda* функције за *Endpoint*-е**

За сваки од *endpoint-*а у *API*-ју, имплементиране су посебне *Lambda* функције. Ове функције обављају специфичне операције, као што је додавање новог пројекта, креирање новог запосленог и други задаци који су кључни за функционалност наше апликације. Ове функционалности су позиване путем *API Gateway-*а, омогућавајући корисницима интеракцију са апликацијом.



На слици џцв приказано је креирање *Lambda* функције користећи *AWS CDK.* Свака *Lambda* представља један модул односно једну *Spring Boot* апликацију. Поред класичних делова *Spring Boot* апликације као што су контролер, сервис, репозиторијум, свака *Lambda* има свој *Handler.* Овај *LambdaHandler* представља *proxy* који иницијализује *Spring Boot* апликацију и прослеђује захтев контролеру, где се обавља пословна логика. Свакој *Lambda* функцији која уписује или чита из *DynamoDB* базе података, мора бити експлицитно дата дозвола за то.

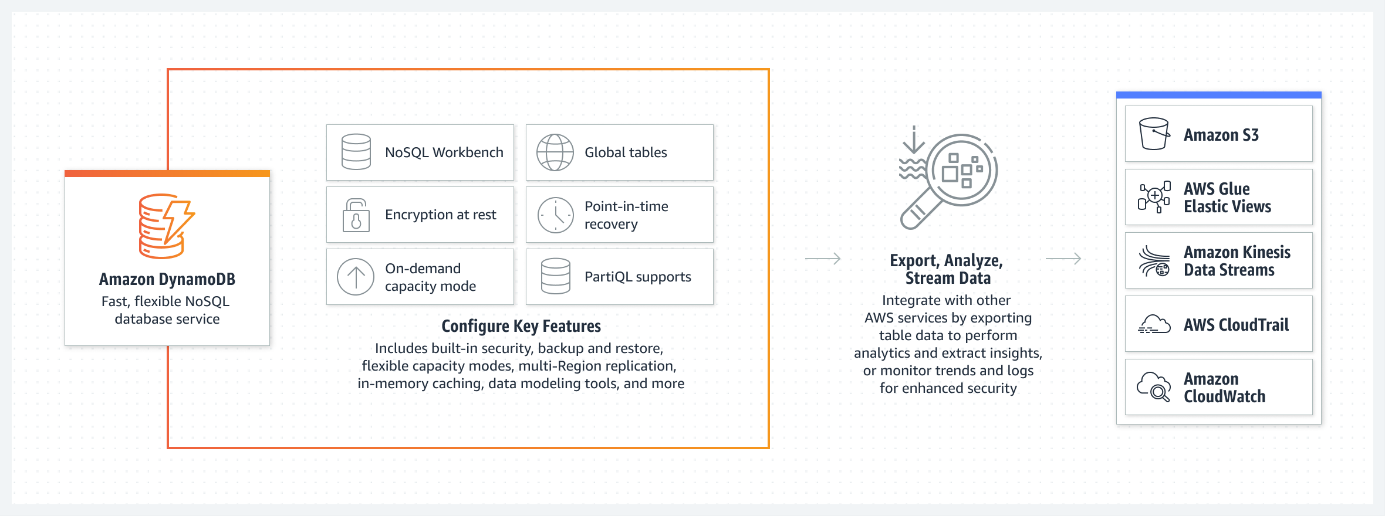
Командом *mvn clean install,* сваки од модула, односно *Lambda* функција, генерише извршну .*jar* датотеку која се складишти у *assets* фолдер. Путања до генерисане .*jar* датотеке се прослеђује при креирању *Lambda* функције.

#### AWS Lambda SnapStart

Један од већих изазова код *Lambda* функција јесте иницијално покретање (енгл. *cold start*). Када се *Lambda* први пут позове, или када није била активна у неком временском периоду, *Lambda* сервис мора извршити неколико операција како би поставио окружење за извршавање функције. Иницијално покретање обично траје дуже неко каснија извршавања исте функције.

*AWS Lambda SnapStart* је нова могућност коју је AWS представио на конференцији *re:Invent 2022.* Ова функционалност омогућава побољшање перформанси покретања функција за апликације које су осетљиве на латенцију, до десет пута без додатних трошкова и обично без промена у коду функције. Највећи допринос латенцији покретања је време које *Lambda* проводи иницијализујући функцију, што укључује учитавање кода функције, покретање *runtime-*а и иницијализацију кода функције. Са *SnapStart-*ом, *Lambda* иницијализује функцију када се објави верзија функције. *Lambda* узима *Firecracker microVM snapshot* меморије и стања диска иницијализованог извршног окружења, шифрује *snapshot* и кешира га за приступ са ниском латенцијом. Када се позове верзија функције први пут и како се позиви повећавају, *Lambda* прави извршно окружење помоћу кешираног *snapshot-*а, уместо да врши иницијализацију из почетка, и тако побољшава време иницијалног покретања. *SnapStart* подржава управљање *Java 11* и *Java 17* извршним окружењима.

### Amazon DynamoDB



*Amazon DynamoDB* је *NoSQL* база података која пружа високу доступност, скалабилност и перформансе. За разлику од традиционалних, *SQL* база података, не користе се *JOIN* операције и друге конструкције релационих база. У овом раду, коришћен је *DynamoDB* са *Single-Table* приступом како би ефикасно моделовали податке и омогућили брже и скалабилне упите. *Single-Table* је приступ моделирању података где се сви подаци складиште у једној табели уместо у више различитих табела. Овај приступ омогућава да сви подаци буду у истом простору, чиме се смањује комплексност и омогућава ефикасан приступ подацима.

Развој са *Amazon DynamoDB* базом и *SQL* базама података се значајно разликује због различите природе ових система. У наставку ћемо проћи кроз неке кључне разлике и аспекте на које треба обратити пажњу при развоју.

*SQL* базе користе табеларни модел података са дефинисаним шемама. Подаци су организовани у табелама са редовима и колонама, често пратећи нормализацију како би се смањила редуданција података. Са друге стране, *DynamoDB* је *NoSQL* база са флексибилнијим моделом података. Нема строго дефинисане шеме и омогућава складиштење разноврсних података у истој табели. Модел података у *DynamoDB* се најчешће базира на приступима (енгл. *access patterns*) које апликација захтева.

У *SQL* базама, приступи подацима се често базирају на *SQL* упитима који користе *JOIN* операције између различитих табела. Такође, подржавају богате *SQL* упите за филтрирање, сортирање и груписање података. Приступи подацима у *DynamoDB* се базирају на одређеним *access pattern*-има које треба унапред планирати. Сваки образац приступа захтева разматрање и пројектовање како би се осигурало ефикасно дохватање података. То може укључивати и креирање глобалних секундарних индекса – *GSI* и пажљив избор кључева за партиционисање и сортирање.

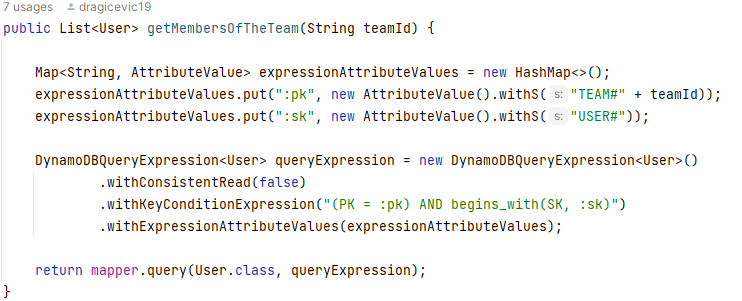
*SQL* базе захтевају пажљиво скалирање инфраструктуре и оптимизацију упита како би се постигле високе перформансе. Вертикално и хоризонтално скалирање су обично потребни за постизање перформанси. Управо то је значајна предност *DynamoDB.* Не захтева управљање инфраструктуром, омогућава аутоматско скалирање складишта и перформанси. Капацитети се аутоматски прилагођавају потребама апликације. То омогућава високе перформансе без потребе за комплексним конфигурацијама.

Када је реч о трошковима, *SQL* базе обично захтевају плаћање лиценце за базу, одржавање хардверске инфраструктуре и административне трошкове. Трошкови су обично фиксни или захтевају руковање капацитетима. *DynamoDB* се наплаћује на основу количине ресурса (читања и уписи) и складиштења које се користи.

*DynamoDB* може бити веома ефикасан, али је од кључног значаја разумети приступе подацима које апликација захтева и моделовати податке према одговарајућим *access pattern*-има. Табела џцв приказује начин на који је моделована апликација за управљање запосленима, пројектима и тимовима.

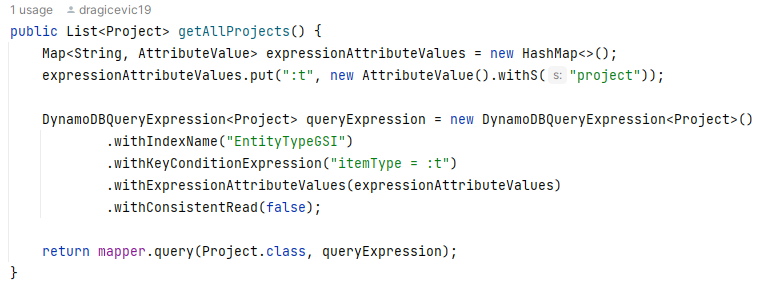
|  |  |  |
| --- | --- | --- |
| *Entity* | *Partition Key* | *Sort Key* |
| User | TEAM#<team\_ID> || NOTEAM | USER#<user\_ID> |
| Project | TEAM#<team\_ID> || NOTEAM | PROJECT#<project\_ID> |
| Team | TEAM#<team\_ID> | TEAM#<team\_ID > |
| UserProject | USER#<user\_ID> | PROJECT#<project\_ID> |

Пример једног *access pattern*-а јесте приказ свих чланова одређеног тима. Код за овакав упит је приказан на слици џцв. *Partition Key* се поставља на: *TEAM#<team\_id>*, где *<team\_id>* представља ID тима чији се чланови траже, а *Sort Key* треба да почиње са *USER#.*



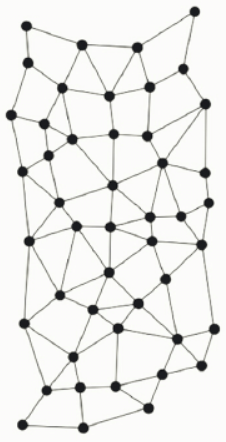
**Глобални секундарни индекси** (енгл. *Global Secondary Index – GSI*) су кључна компонента *DynamoDB* која омогућава ефикасан приступ подацима на различите начине, што је посебно корисно када је потребно подржати различите *access pattern*-е у апликацији. *GSI* је индекс који се може креирати у *DynamoDB* табели и омогућава ефикасан приступ подацима по различитим атрибутима од оних који су коришћени као кључеви табеле. Ово омогућава брзе упите и филтрирање података без потребе за скенирањем целе табеле.

У овом раду, коришћен је *GSI* да би имали могућност ефикасног приступа свим тимовима, пројектима или запосленима. Атрибут ***itemType*** постављен је као *Partition Key* за *GSI,* што значи да можемо врло брзо добити листу свих ентитета са одговарајућом вредношћу атрибута *itemType.* Ако желимо да довучемо све тимове, претражићемо GSI где је *partition* key, односно атрибут *itemType,* једнак *team.* Код за добављање свих пројеката је приказан на слици џцв*.*

****

## Дистрибусирани системи

Дистрибуирани систем се може дефинисати као колекција независних компоненти које међусобно комуницирају ради постизања заједничких циљева. Као такав, овај систем ће наизглед попримити форму једног интерфејса или јединственог рачунара крајњем кориснику. Сама идеја је таква да се заједничким радом може на максимално ефикасан начин искористити сви ресурси и информације. Такође, саме грешке су ретко скупе, јер уколико дође до отказа на једном уређају, ово не би требало да утиче на доступност сервиса. Још једна значајна особина дистрибуираних система јесте то да су изузетно флексибилни, јер се са лакоћом могу додати компоненте или ојачати већ постојеће [5, 6]. На слици 3.1.1 се може видети илустрација дистрибуираног система.



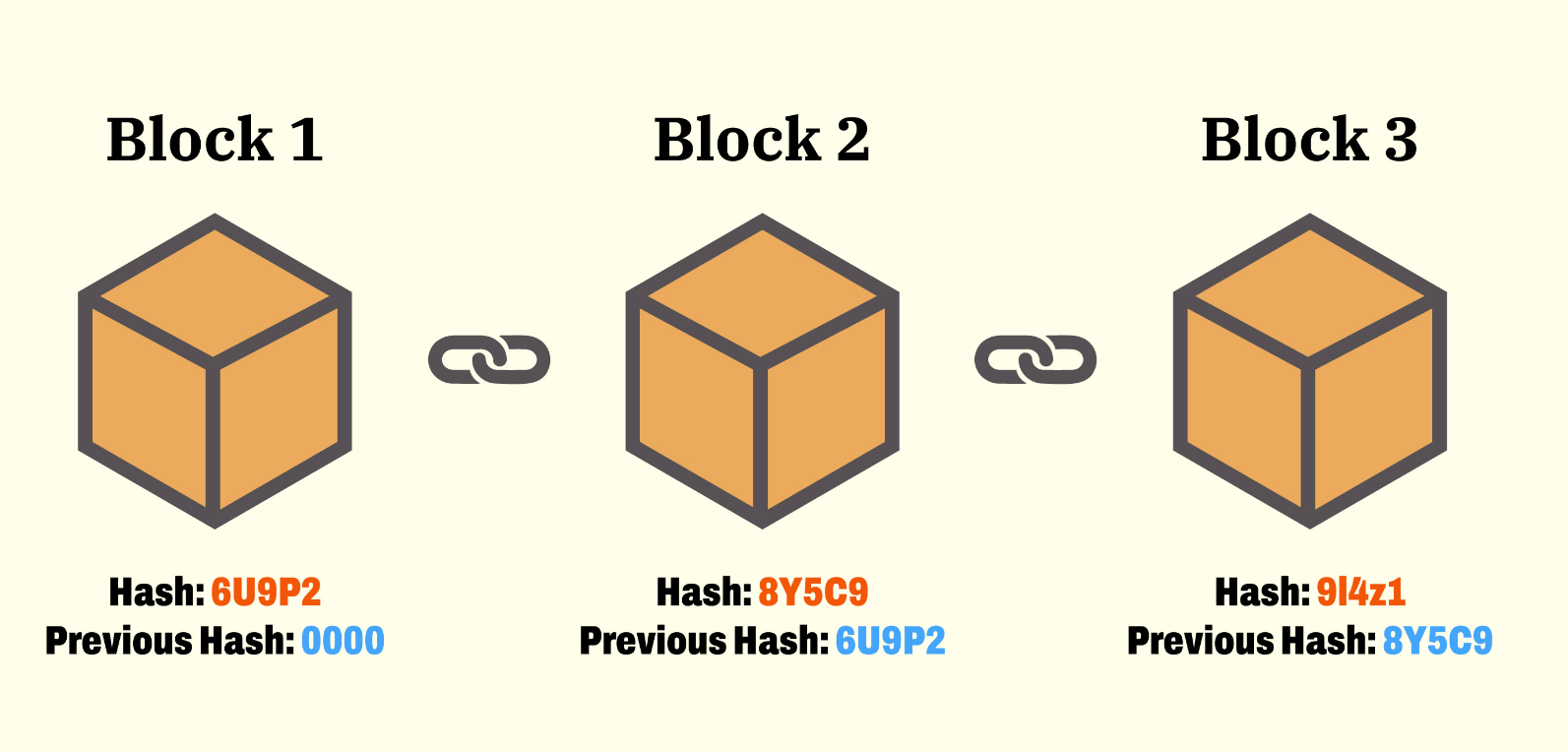
Слика 3.1.1 дистрибуирани систем [40]

Неки од значајних бенефита употребе дистрибуираних система су свакако лакоћа скалирања (како хоризонталног тако и вертикалног), конкурентност, толеранција на грешке и транспарентност. Нажалост, имплементација једног оваквог система долази и са изазовима, као што је повећана комплексност, сложенија поставка безбедности и потешкоће синхронизовања процеса [6].

## Блокчејн

Блокчејн се неретко дефинише као јавно доступна дистрибуирана главна књига, која је подељена између чворова компјутерске мреже [7]. Његова форма је облика ланац блокова који по структури подсећа на једноструко спрегнуту листу. Као технологија је први пут описана 1991. године чија је оригинална идеја била да временски означи дигиталне документе и да самим тим смањи ризик антидатирања или злонамерног манипулисања њиме [8], али је њена прва практична употреба била 2009. у виду платформе познате као *Bitcoin* [9].

Идеја блокчејна јесте да омогући чување и дистрибуцију информација, али не и њену измену. Блок представља градивну јединицу блокчејна и сваки блок садржи одређени скуп података, сопствени хеш и хеш претходног блока. Неки од најбитнијих података који се налазе у сваком блоку су трансакције, временска одредница блока, величина блока и енкриптовани број (*nonce*). Сваки блок може да прими ограничену количину информација, и уколико се превазиђе капацитет блока, формира се нови [10, 11]. Хеш блока представља његов јединствени идентификатор. Сам хеш се рачуна током креирања блока, и он се приликом измене самог блока мења. Хеш претходног блока се складишти како би се формирао ланац блокова и додатно осигурао блокчејн. Наравно, први блок у ланцу може да садржи само свој хеш и скуп података и он се назива иницијални блок (genesis block). На слици 3.2.1 је приказан једноставни пример блокчејна.



слика 3.2.1 блокчејн [41]

## Консензус алгоритми

До популаризације блокчејна је у великој мери дошло због имутабилности, транспарентности, приватности и безбедности коју ова технологија пружа [12]. Иако не постоји један централни ауторитет, свака трансакција се сматра верификованом и обезбеђеном. Ово је могуће због употребе консензус алгоритама.

Консензус алгоритам је процедура уз помоћ које сви учесници блокчејн мреже могу да дођу до заједничког закључка о стању самог блокчејна у реалном времену. Овај механизам омогућава мрежи да постигне поузданост тако што формира ниво поверења између различитих чворова док истовремено осигурава безбедност у окружењу. Суштински, консензус алгоритам обезбеђује да сваки нови блок који се дода на блокчејн буде једина права истинита верзија која је одобрена од стране чворова блокчејна. Сваки алгоритам тежи ка више циљева, као што су постизање заједничког споразума, пружање економског подстицаја, омогућавање поштеног и равноправног учешћа и максимизација отпорности на грешке [13, 14, 15]. Иако тренутно постоји велики број алгоритама, сваки са својим случајем коришћења, рад ће се фокусирати на 2 најпознатија консензус алгоритма: доказ о раду (*Proof of Work – PoW)* и доказ о улогу (*Proof of Stake – PoS)*.

### Доказ о раду

Доказ о раду је најстарији консензус механизам коришћен у блокчејн технологији и тренутно га користе *Bitcoin* и *Ethereum [17]*. Идеја иза овог механизма јесте да се чворови учесници, боље познати као рудари, међусобно такмиче у решавању комплексних математичких проблема употребом моћних рачунара. Рудар који први реши проблем и нађе одговарајући хеш стиче право да формира нови блок и потврди трансакције. Он је такође награђен са унапред дефинисаном количином криптовалута [15, 16].

Иако се генерално сматра најпоузданијим од свих механизама консензуса, свакако постоје потенцијални проблеми о којима се мора водити рачуна. Један од њих је могућност да централни ентитет поседује 51% или више чворова у мрежи и самим тим корумпира блокчејн тако што ће добити већину мреже. Ово је познато и као ризик 51%. Такође, ово захтева велику количину компјутерске моћи што не само што је изузетно скупо, него толика електрична потрошња може имати изузетно лош утицај на животну средину [14].

### Доказ о улогу

Као алтернатива *PoW* се најчешће користи *PoS*. Разлог популаризације овог консензус алгоритма јесте што не захтева скупе хардверске компоненте за решавање комплексних математичких загонетки. Учесници овог механизма се називају валидатори и они стичу право да потврде блокове који би могли да се додају на блокчејн. Међутим, да би се стекао статус валидатора, неопходно је приложити одређену количину криптовалута као улог. У зависности од конкретних блокова који се додају на блокчејн. Валидација блокова се врши тако што се валидатори кладе на откривене блокове који би се потенцијално могли додати у блокчејн. У зависности од додатих блокова, сви валидатори добијају награду у складу са њиховом уложеном опкладом и вредност њиховог улога складно расте. На крају *PoS* за генерисање блока се бира валидатор чија је вредност улога највећа на мрежи [14, 16].

Упркос његовој ефикасности и одрживости, овај приступ је наишао на критику због могућности да доведе до централизације ланца. Разлог томе је што валидатори са већом количином новца могу стећи већу контролу над системом, и на тај начин нарушити децентрализовану природу система [15].

## Ethereum

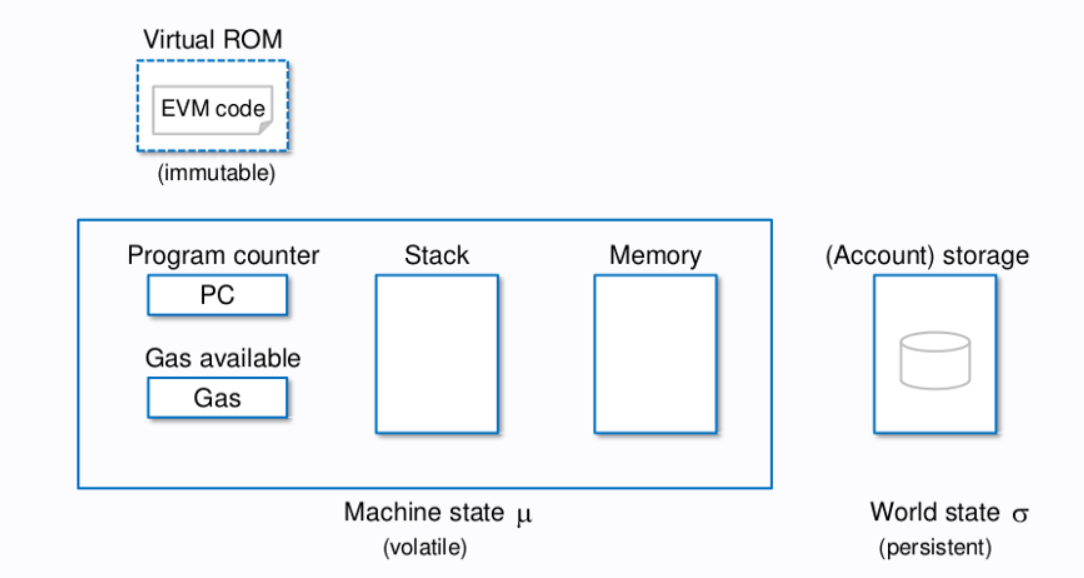
На самом почетку поглавља је споменут *Bitcoin* као прва апликативна примена блокчејна. Иако револуционарна технологија, сам *Bitcoin* је по дизајну туринг непотпун [18]. Другим речима, он не може да покрене комплексне програме или да реши сложене задатке. Међутим, *Bitcoin* је намерно имплементиран на овај начин, како би се избегло коришћење превише ресурса за чворове и на тај начин повећала једноставност, али и безбедност платформе. Иако је овакав дизајн био погодан за потребе *Bitcoin-a*, свакако је идеја туринг потпуне блокчејн платформе била примамљива. Један пример такве платформе јесте *Ethereum*.

*Ethereum* је децентрализована глобална платформа заснована на блокчејн технологији која омогућава свим својим корисницима да креирају безбедну дигиталну технологију. *Ethereum* такође има своју валуту, познату као *Ether* (*ETH*) [35], као и програмски језик *Solidity [34]*.

Велика предност *Ethereum-a* јесте његова туринг потпуна природа. Баш због те карактеристике је постала јако погодна платформа за програмере који желе да развију своју децентрализовану апликацију. Развој ових децентрализованих апликација је омогућен захваљујући алату који је познат као паметни уговор.

Најједноставније дефинисано, паметни уговор је обичан програм који се покреће на *Ethereum-*овом блокчејну. Он садржи код и податке који су ускладиштени на посебној адреси на самом блокчејну. Сви учесници могу да интерагују са паметним уговором, али ниједан од њих нема контролу над њим, чак ни учесник који је поставио паметни уговор на блокчејн. Ове интеракције се постижу подношењем трансакција које затим извршавају функције дефинисане у паметном уговору. Свака трансакција захтева и одређену количину гаса. Гас је вид додатне таксе неопходне за извршавање кода у паметном уговору. Количина гаса у великој мери зависи од тренутне понуде и потражње за рачунарском снагом која је потребна за обраду паметних уговора и трансакција. Паметни уговори дозвољавају обављање трансакција између анонимних страна без потребе за централним органом. Свака трансакција извршена у оквиру паметног уговора је следљива, транспарентна и иреверзибилна [20, 21, 22].

Темељ читаве оперативне структуре *Ethereum-*a је *Ethereum Virtual Machine (EVM)*. *EVM* је задужен за извршавање и постављање паметних уговора уз помоћ њене мреже дистрибуираних јавних чворова. Како би се постигао консензус, сваки *Ethereum*-ов чвор ради на *EVM*. Виртуелна машина је потпуно изолована и самим тим код унутар *EVM* нема приступ мрежи, фајл системима или другим процесима [23, 24, 25]. На слици 3.4.1 се може видети архитектура *EVM*.



Слика 3.4.1 архитектура Етеријумове виртуелне машине [42]

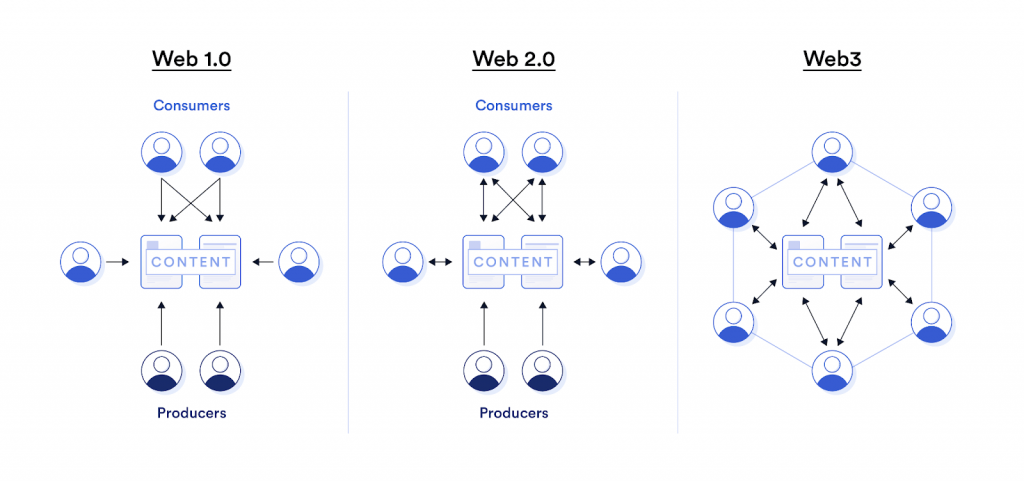
## Светска мрежа и њене итерације

Од свог оснивања, светска мрежа је изузетно напредовала и данас се сматра да постоји више верзија веба. Тренутан став је тај да је веб прошао кроз 2 итерације, а да ће основу наредне итерације чинити блокчејн и паметни уговори.

Прва итерација је позната као *Web 1.0*. То је мрежа отвореног кода без пермисија која је пружала стриктно статички садржај. Ова верзија веба је дизајнирана тако да помогне корисницима да боље пронађу информације. Међутим, ова иницијална фаза мреже се сматра доста примитивном јер му недостају битни динамички елементи као што су форме, визуелни елементи и интерактивност [26].

Наредна етапа је позната као *Web 2.0*. *Web 2.0* описује тренутно стање светске мреже, које пружа динамички садржај и бољу употребљивост за крајње кориснике у поређењу са *Web 1.0*. Ова нова верзија веба је омогућила виши ниво размене информација, али и повећану међусобну повезаност између корисника [26, 27].

Иако се позитиван утицај *Web 2.0* не може занемарити, ова етапа је имала и мане. Порастом учесника веба, Одређен број врхунских компанија је остварило монопол над интернетом и тако су стекли велику моћ у одлучивању шта је дозвољено, а шта није. Узимајући у обзир ову чињеницу, као и нагли развој блокчејна, полако је настала идеја о потенцијално новој итерацији мреже која би се звала *Web 3.0*. Концепт је формирање потпуно децентрализоване мреже без пермисија са динамичким садржајем и економијом која је базирана на криптовалутама и токенима. Велики бенефит који *Web 3.0* пружа је приватност података уз помоћ уграђене псеудонимности, али и поновна дистрибуција власништва над интернетом [26, 27, 28]. На слици 3.5.1 се налази илустрација горепоменутих итерација веба.



Слика 3.5.1 итерације веба [43]

# Модел система

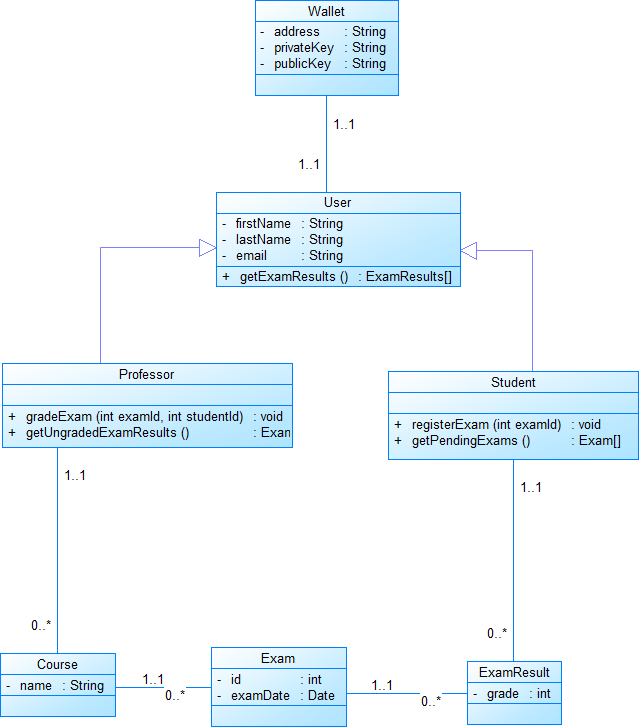
Како би се најбоље разумео концепт и начин пословања децентрализоване апликације студентске службе, неопходно је прво описати њен модел. За ову сврху, апликацију можемо представити путем дијаграма који репрезентују систем на различитим нивоима апстракције. У наставку поглавља биће приказана 3 дијаграма: класни дијаграм, дијаграм случајева коришћена и дијаграм активности.

## Класни дијаграм

Класни дијаграм је тип дијаграма статичке структуре коришћен зарад описа система преко класа, њених атрибута и метода, али и односа између објеката.

На слици 4.1.1 приказан је класни дијаграм. Класе у овом дијаграму су:

* *Wallet* (енгл. новчаник) – новчаник складишти криптовалуте корисника и преко њега се обављају све трансакције. Први атрибут *Wallet*-a је адреса. Адреса новчаника је хексадецимална вредност изведена из јавног кључа новчаника и представља јавни идентификатор сваког новчаника преко којег могу да се уплате средства на налог. Поред овога сваки новчаник садржи и 2 кључа: приватни кључ и јавни кључ. Јавни кључ се користи како би могле да се пошаљу криптовалуте новчанику. Са друге стране, улога приватног кључа је налик шифри и користи се зарад верификације трансакција и доказивања власништва над адресом. Последњи атрибут је салдо и то је децимални број који представља количину *ETH* на новчанику.
* *Course* (енгл. курс) – као атрибуте има назив курса и конкретног предавача.
* *Exam* (енгл. Испит) – као атрибуте има шифру испита, датум одржавања испита и асоцирани предмет.
* *ЕxamResult* (енгл. резултат испита) – као атрибуте има студента који је полагао предмет, оцену и конкретан испит.
* *User* (енгл. корисник) – као атрибуте садржи новчаник, име и презиме и имејл адресу. Такође, корисник има методу преко које може да прегледа резултате испита. наслеђују је класе:
  + *Student* (енгл. студент) – не поседује додатне атрибуте, али има операцију преко које може да пријави испит и добије преглед непријављених испита.
  + *Professor* (енгл. професор) – професор као атрибут још садржи и низ курсева на којима предаје. Такође, професор има методу за добављање неоцењених испита и оцењивање испита.



Слика 4.1.1 дијаграм класе

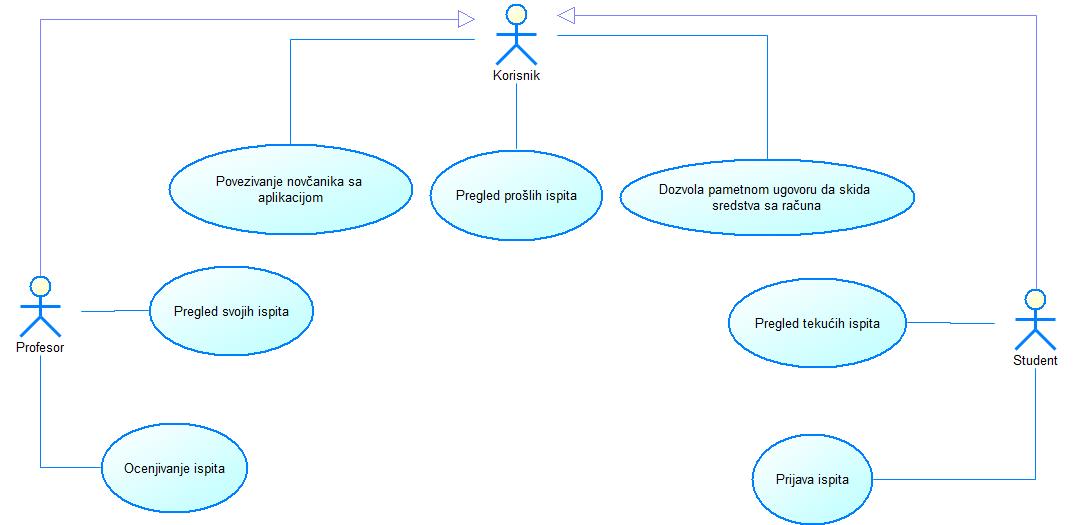
## Дијаграм случајева коришћења

На слици 4.2.1 приказан је дијаграм случајева коришћења. Улога дијаграма случајева коришћења јесте да демонстрира различите начине на које учесници или актери система могу интераговати са системом. Идеја јесте да се направи апстрактна депикција релација измећу корисника, случајева коришћења и система.

У случају описане апликације, постоје 2 врсте актера: студент и професор. Заједничко за обе врсте корисника је то што могу да повежу свој новчаник са апликацијом, као и да дозволе паметном уговору да скида средства са рачуна приликом извршавања трансакција. Осим тога они могу да прегледају резултате прошлих испита: у случају студента он има резултате свих испита на које је изашао, док професор има приказ свих испита које је оценио. Ово су једине заједничке функционалности за обе роле.

Поред прегледа оцењених испита, професор такође има могућност да види списак свих пријављених испита које он држи, а који још увек нису оцењени. Такође, професор може да оцени неки од испита.

Са друге стране, студент има преглед свих текућих непријављених испита за које испуњава услове за пријаву. Oсим тога, уколико студент задовољава услове, он може да пријави испит студентској служби.



Слика 4.2.1 дијаграм случајева коришћења

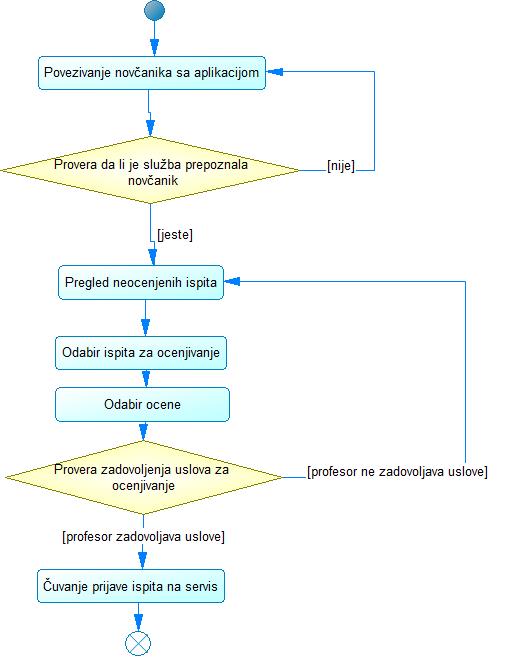
## Дијаграм активности

Дијаграм активности се користи ради описа како су активности у оквиру једног система координисане. Сваки догађај је сачињен од скупа операција где су наведене операције најчешће корелиране или у јакој релацији и самим тим наредна операција може у већој или мањој мери зависити од претходне.

Ток пријаве испита изгледа овако:

1. студент се пријављује на службу повезивањем свог новчаника.
2. уколико је служба препознала новчаник студента, њему се приказује преглед свих испита за које је испунио услове полагања.
3. студент бира један од испита који жели да пријави.
4. Пре него што прође пријава на систем, неопходно је проверити да ли студент задовољава све критеријуме прво (да ли има довољно средства на рачуну, да ли је испит пријављен више од 3 дана пре почетка одржавања итд.).
5. Уколико су ови критеријуми задовољени, пријава ће бити успешно сачувана.

На слици 4.2.1 приказан је дијаграм активности за студента.

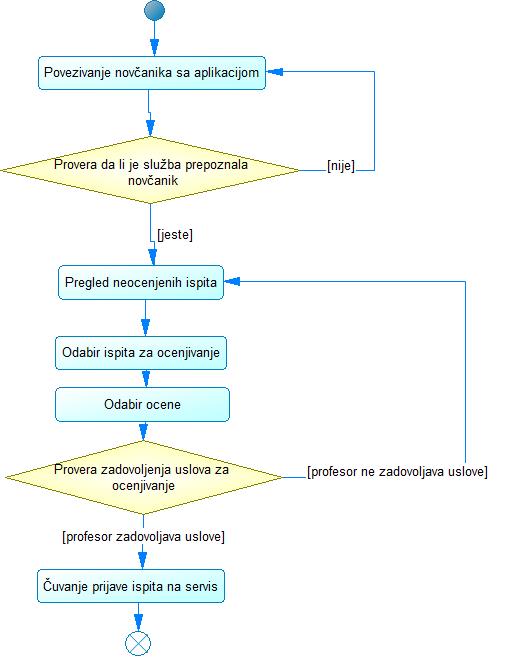


Слика 4.2.1 дијаграм активности за студента

Ток оцењивања испита изгледа овако:

1. професор се пријављује на службу повезивањем свог новчаника.
2. уколико је служба препознала новчаник професора, њему се приказује преглед свих неоцењених испита које он држи.
3. Након одабира испита и оцене, уколико има довољно средства на рачуну да плати гас, испит се архивира на служби.

На слици 4.2.2 приказан је дијаграм активности за професора.



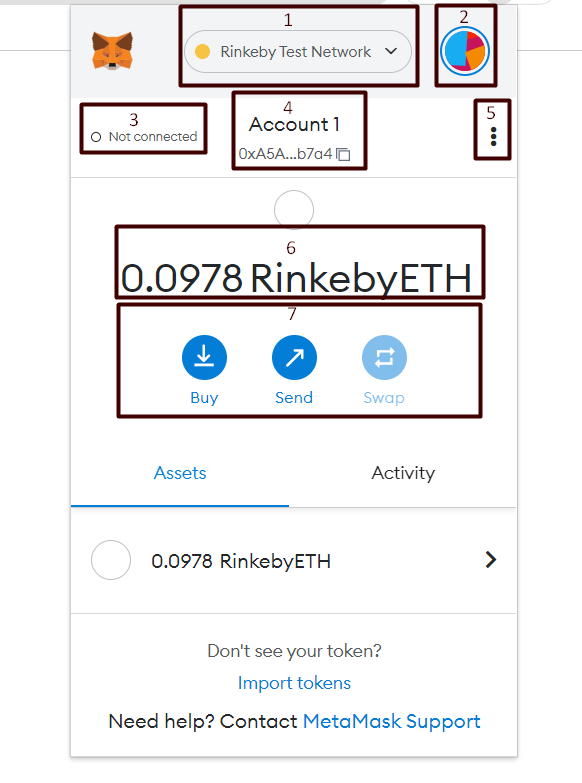
Слика 4.2.2 дијаграм активности за професора

# Имплементација система

Наредно поглавље ће бити фокусирано на саму имплементацију система, процес развоја и тестирања, као и финално решење. Ради погодности, ово поглавље ће бити подељено на 3 сегмента. Први део ће се фокусирати на неке основне концепте и компоненте које је неопходно разумети и подесити пре развоја и употребе програма. Други део ће се фокусирати на бекенд аспект решења, тачније на писање и поставку паметног уговора, док ће последњи део анализирати фронтенд апликације, конкретније самог клијента, али и његову комуникацију са паметним уговором.

## Иницијална поставка

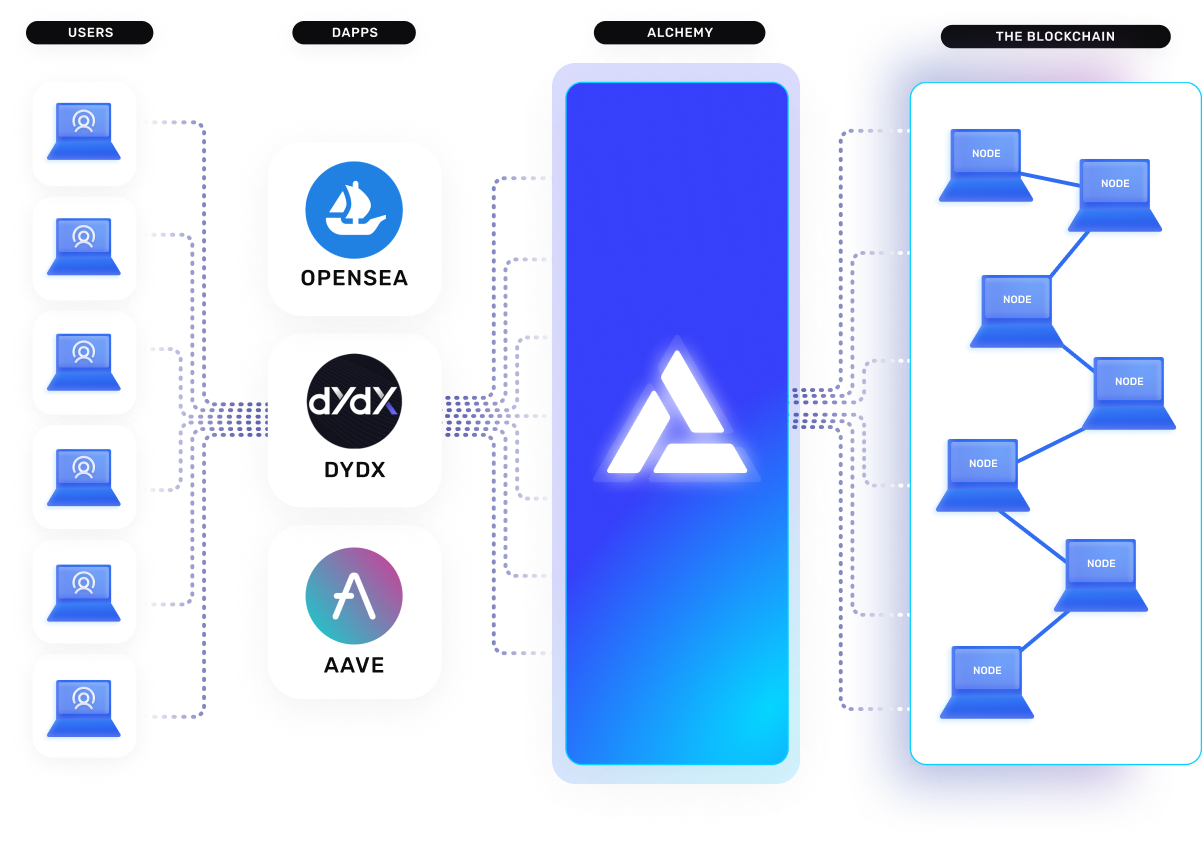
Како би уопште могла да се постигне комуникација са децентрализованим апликацијама, али и могућност развоја истих, прво је неопходно да се подеси лични новчаник криптовалута. Сврха ових новчаника јесте да омогући корисницима да шаљу и примају криптовалуте, али и да извршавају трансакције над паметним уговорима. Постоји велики број ових новчаника и сваки од њих има своје предности и мане, међутим за сврху овог рада коришћен је *MetaMask* [29]. Овај новчаник је изабран из простог разлога што је он *Ethereum-*ов новчаник, а коначно решење овог рада је развијено на *Ethereum* платформи. Сама инсталација *MetaMask*-a је једноставна и интуитивна, која захтева само инсталирање екстензије претраживача и постављање шифре.



Слика 5.1.1 *MetaMask* новчаник

На слици 5.1.1 је приказан интерфејс *MetaMask* новчаника. Под бројем 1 је означен падајући мени уз помоћ којег се може одабрати блокчејн мрежа на коју корисник жели да се повеже. Подразумевано се као опције налазе *Ethereum* главна мрежа, али и тестне мреже *Ropsten*, *Kovan*, *Rinkeby* и *Goerli*, а корисник по потреби може додати више мрежа [30]. Ради олакшаног развоја и како програмери не би користили права средства приликом развоја апликације, неретко се користе неке од претходно наведених тестних мрежа. За сврху овог пројекта је коришћена *Rinkeby* мрежа. Дугме са бројем 2 омогућава кориснику рад са налозима. Након што кликне ово дугме, корисник може да се пребаци на други налог, али такође може да креира или импортује нов налог. Број 3 представља индикатор који показује да ли је новчаник повезан са апликацијом или страницом. Број 4 представља назив и адресу тренутног налога. Број пет је падајући мени који пружа кориснику функционалности као могућност да прошири приказ интерфејса, погледа детаље налога и добије приказ свих повезаних страница. Број 6 представља конкретну валуту за изабрану мрежу као и колику количину те валуте корисник има у свом новчанику.

Након поставке новчаника, наредни корак је креирање налога на *Alchemy-у [31]*. *Alchemy* представља моћну блокчејн платформу за програмере која пружа низ алата за развој. Ова платформа замењује стандардне локалне чворове са чворовима који имају бржу, скалабилнију и децентрализованију архитектуру. Поред овога, он пружа алате за мониторинг, аналитику, евидентирање и дебаговање грешака за блокчејн базиран софтвер. За потребе овог пројекта, креиран је *Alchemy* пројекат за *Ethereum* *Rinkeby* тест мрежу. Након креирањa апликације, неопходно је извући *Application Programming Interface* (*API*) кључ како би се успешно аутентификовао корисников чвор и омогућило слање захтева ка блокчејну. На слици 5.1.2 је приказана илустрација *Alchemy* платформе.



Слика 5.1.2 *Alchemy [31]*

Последњи важни предуслов пре почетка развоја апликације јесте упознавање са *Etherscan* платформом [32]. *Etherscan* се дефинише као блокчејн претраживач за *Ethereum*. Преко ове платформе могуће је пронаћи важне податке сачуване на ланцу, као што су трансакције, блокови, као и адресе новчаника и паметних уговора. Уз помоћ овог алата корисник може боље разумети како се постиже интеракција са компонентама блокчејна. Поред ових функционалности, *Etherscan* се може користити и за верификацију паметних уговора. Ова верификација је нужна, јер преко ње се обезбеђује поверење и континуитет у извршењу пословних процеса паметног уговора. Како би се програмеру омогућило да верификује свој паметни уговор приликом поставке на блокчејн, неопходно је направити налог на *Etherscan* платформи и затим креирати свој *API* кључ.

## Бекенд део апликације

За потребе развоја бекенд стране рада, коришћен је радни оквир *Hardhat [33]*. *Hardhat* је окружење за тестирање, компајлирање, постављање и дебаговање децентрализованих апликација засновано на *Ethereum* блокчејну. Захваљујући *Hardhat-у*, програмери могу да организују задатке неопходне за развој децентрализованих апликација и паметних уговора. Поред тога, *Hardhat* омогућује аутоматизацију одређених задатака и пружа девелоперима мноштво корисних функција. Такође, велика предност овог радног оквира јесте њена уграђена локална *Ethereum* мрежа која је фокусирана на дебаговање *Solidity* кода, тако што пружа поруке грешке приликом компајлирања, извршавања кода, постављања итд.

Како би се инкорпорирао *Hardhat* у решење, прво је неопходно креирати пројекат. За ову потребу се може искористити било који менаџер пројекта или пакета, али је за рад коришћен *Yarn [36]*. У празан директоријум се прво унесе команда приказана на слици 5.2.1.



Слика 5.2.1 иницијализација *Yarn-а*

Након покретање ове команде, у празном директоријуму је креирана *package.json* датотека. Овај фајл је задужен за складиштење метаподатака пројекта, али се овде налазе и зависности, скрипте, верзија и друге важне информације о самом пројекту. Да би укључили *Hardhat* у пројекат неопходно је унети команду као на слици 5.2.2.



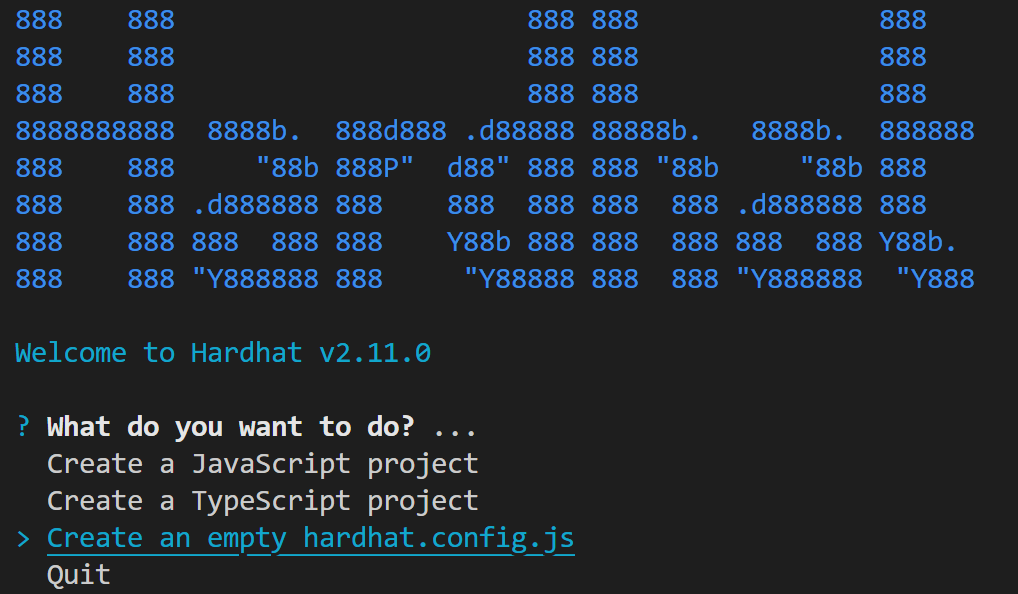
Слика 5.2.2 поставка *Hardhat-a* у пројекат

Покретањем ове команде, као зависност је сачуван *Hardhat*. Да бисмо започели рад са овим окружењем, потребно је терминалу проследити унос као на слици 5.2.3.



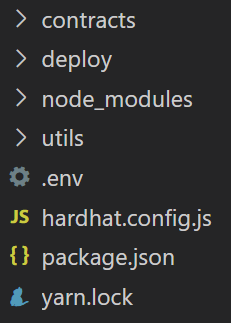
Слика 5.2.3 иницијализација *Hardhat-a*

Након ове команде, у оквиру терминала се приказује *Hardhat-ова* форма. Ова форма нуди кориснику наредне опције: креирање *JavaScript* пројекта, креирање *TypeScript* пројекта, креирање празног пројекта који садржи само *hardhat.config.jsоn* конфигурациони фајл или одустанак од креирања пројекта. За потребе пројекта изабрана је опција генерисања празног пројекта. Изглед форме је представљен на слици 5.2.4.



Слика 5.2.4 *Hardhat-a* форма за генерисање пројекта

Након свих горе наведених корака, *Hardhat* окружење ће бити успешно имплементирано у оквиру пројекта. Следећи поступак јесте организација пројекта. У оквиру слике 5.2.5 може се видети хијерархија фолдера и фајлова.

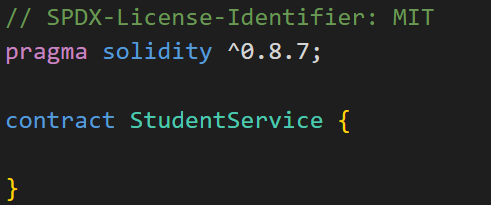


Слика 5.2.5 хијерархија пројекта

Како би се боље разумео пројекат, неопходно је објаснити кључне директоријуме и датотеке из претходне слике:

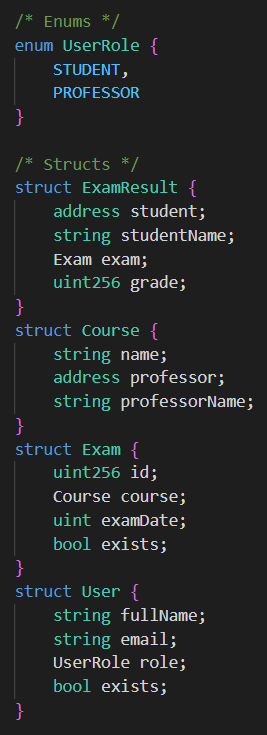
* *contracts -* Садржи паметни уговор написан у *Solidity* језику.
* *deploy –* Садржи две скрипте: једну везано за постављање паметног уговора на блокчејн и другу за прослеђивање важне информације о самом паметном уговору клијенту.
* *utils –* Садржи помоћну класу за верификацију паметног уговора.
* *.*env – Датотека која складишти осетљиве променљиве средине. Овде се чува приватни кључ корисника који поставља паметни уговор на мрежу, али и његови *API* кључеви креирани на *Alchemy* и *Etherscan* платформи.

У оквиру *contracts* директоријума налази се датотека *StudentService.sol.* Ова датотека представља основу самог бекенда јер је у оквиру ње написан паметни уговор у *Solidity* језику. *Solidity* је објектно оријентисан програмски језик високог нивоа који се користи за имплементацију паметних уговора. Он је статички типизиран језик витичастих заграда и подржава концепте као што су наслеђивање, употреба библиотека и комплексних типова. Основа сваког *Solidity* програма јесте паметни уговор. У оквиру *Solidity-а*, паметни уговор се дефинише кључном речју *contract*, након које иде назив уговора и пар витичастих заграда. Овај уговор је концептуално веома сличан класама у објектно оријентисаним језицима, јер у оквиру уговора се налазе компоненте и концепти као што су методе, атрибути, модификатори приступа, конструктор, наслеђивање итд. Међутим, пре дефинисања уговора, неопходно је резервисати 2 линије кода за посебне сврхе. У првој линији кода се специфицира лиценца коју уговор користи. У другој линији кода је описана верзија компајлера коришћена у оквиру фајла. На слици 5.2.6 је приказан иницијални изглед *StudentService.sol* фајла.



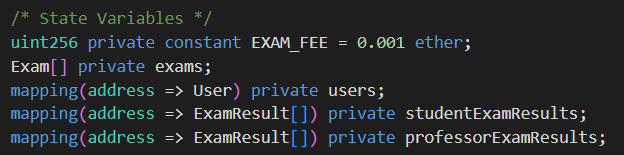
Слика 5.2.6 *StudentService.sol*

У оквиру витичастих заграда ће бити написан сав програмски код. Прва ствар која ће се дефинисати је модел апликације. Модел у овом уговору је конзистентан са класним дијаграмом описаним у претходном поглављу и имплементиран је употребом структура. Структуре у *Solidity* језику су у великој мери сличне структурама у другим програмским језицима. То су комплексни типови података који садрже више једноставних и комплексних атрибута. Структуре су дефинисане кључном речју *struct*. Атрибути се дефинишу у структи тако што се наводи прво тип, а након њега назив атрибута. Пре структура је дефинисана и једна енумерација која одређује улогу корисника и садржи две вредности: студент и професор. На слици 5.2.7 је приказана енумерација и све структуре.



Слика 5.2.7 модел паметног уговора

Након модела, наредни блок кода представља скуп променљивих. Конкретно, ове променљиве се користе ради чувања стања паметног уговора и играју сличну улогу као база података, јер складиште важне информације. Када се варијабла стања дефинише, прво се наводи тип података варијабле, након чега иде модификатор приступа и на крају назив варијабле. Пре назива променљиве, може се додати модификатор *constant* или *immutable.* Просто речено, први модификатор сигнализира да се варијабли додељује вредност само приликом њене дефиниције, док други модификатор омогућава да се вредност променљиве дефинише једном и само једном у оквиру паметног уговора. На слици 5.2.8 су приказане све варијабле стања.

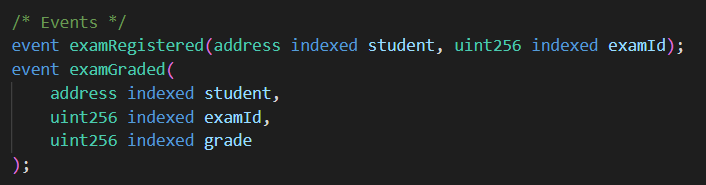


Слика 5.2.8 варијабле стања

Свака варијабла на слици 5.2.8 има приватни модификатор приступа и самим тим видљива је само у оквиру уговора. Прва варијабла је константа типа неозначени број од 256 бита и он представља цену пријаве једног испита. Наредна варијабла представља низ свих испита. Како би се разумеле последње три променљиве, важно је разумети *mapping* тип. По структури *mapping* је налик хеш табели и користи се за складиштење парова кључ-вредност. Прва од ове три променљиве представља скуп свих корисника у систему. Кључ је адреса новчаника корисника, а вредност садржи податке о кориснику са том адресом. Последње две варијабле складиште информације о резултатима испита. Тачније, прва варијабла као кључ чува адресу новчаника студента, а као вредност чува низ свих испита на које је изашао. Кључ друге варијабле је адреса професора, а вредност представља низ испита које је оценио.

У наставку програма налазе се догађаји или *event* компоненте. Уз помоћ *event* компоненти, могуће је успоставити комуникацију са клијентском апликацијом и обавестити је да је дошло до промене на блокчејну након одређене трансакције. Најчешће им се придодају параметри који представљају важне информације о трансакцији. Ови параметри могу садржати кључну реч *indexed* која олакшава претрагу догађаја преко тих параметара. *На* слици 5.2.8 налазе се 2 догађаја:

* *examRegistered* – догађај који се емитује након регистрације испита. Као параметре прима адресу студента и шифру испита.
* *examGraded* – догађај који се емитује након оцењивања испита. Као параметре прима адресу студента, шифру испита и уписану оцену.



Слика 5.2.8 догађаји

Следећи сегмент уговора чине ставке које су карактеристичне за *Solidity* и називају се модификатори. Улога модификаторе јесте да промени понашање функције на коју се примени. Генерално се користи за проверу испуњења услова пре или после извршења функције. На овај начин смањује се редундантност кода и омогућује поновна употреба у различитим деловима кода. Сваки модификатор се дефинише кључном речју *modifier* која је праћена називом модификатора и његовим телом где се дефинише код. За потребе пројекта, ови модификатори се користе за руковање са изузецима.



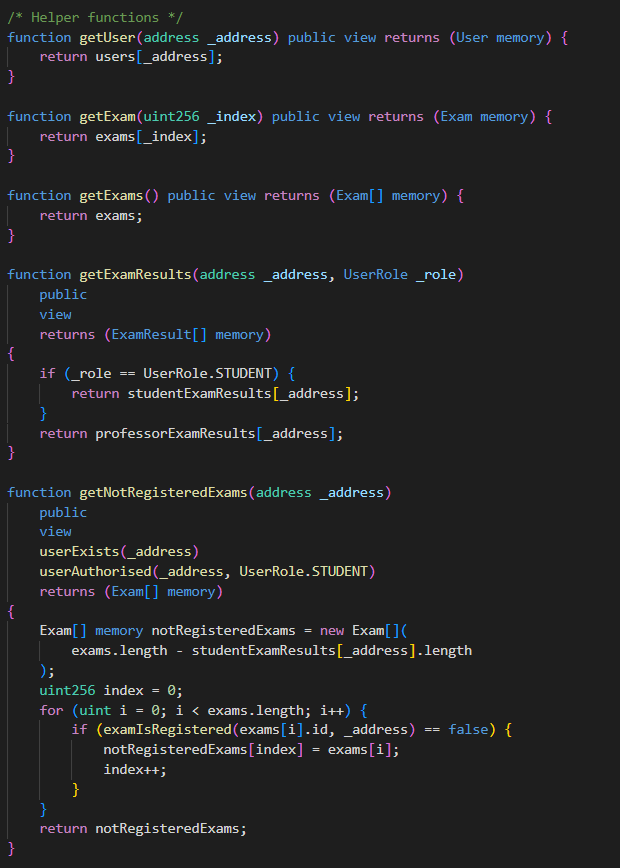
Слика 5.2.9 модификатори

Сваки модификатор на крају тела садржи симбол \_;. Ово само означава да желимо да се програмски код модификатора изврши пре функције. Улога модификатора на слици 5.2.9 је:

* *userExists* – проверава да ли корисник са датом адресом постоји.
* *examExists* – проверава да ли испит са датом шифром постоји.
* *userAuthorised* – проверава да ли је корисник ауторизован за извршавање функције.
* *sufficientETH* – Проверава да ли је прослеђено довољно средства за пријаву испита.
* *examRegisteredLate* – Проверава да ли је испит пријављен мање од 2 дана пре одржавања.
* *examHistoryExists –* проверава да ли је студент са датом адресом већ пријавио испит са датом шифром.

Као и други језици са објектно оријентисаном парадигмом, и *Solidity* садржи конструкторе. У оквиру ове посебне функције може се писати код и ова функција ће се извршити само једном након креирања уговора. За потребе пројекта, конструктор се користио како би се поставиле иницијалне вредности у оквиру горе наведених варијабли стања.

Последњи део уговора чини скуп функција. Функције у *Solidity* језику су описане са кључном речју *function* након чега иде назив функције и низ параметара. Поред овога, може се дефинисати модификатор приступа функције, повратна вредност функције, али и низ *modifier-a*. Ради прегледности, функције су подељене у 2 групе: помоћне и главне функције. На слици 5.2.10 се налазе помоћне функције за добављање података.

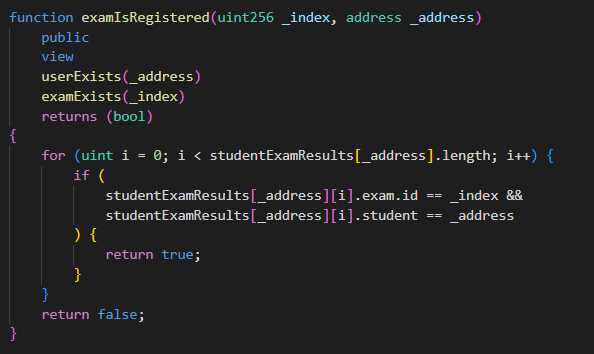


Слика 5.2.10 помоћне функције за добављање података

Специфично за ове тзв. *getter* функције јесте атрибут *view* дефинисан након кључне речи *public.* Овај атрибут се користи уколико функција не мења стање уговора, него се користи само за читање вредности варијабли стања. Самим тим, пошто променљиве стања остају нетакнуте, позивање ових функција неће захтевати гас. Такође, у оквиру повратне вредности функције, поред типа, налази се и кључна реч *memory.* Ова реч указује да се та вредност складишти само привремено и након извршавања се брише. Улога функција са слике 5.2.10 је:

* *getUser* – добавља корисника са датом адресом.
* *getExam* – добавља испит са датом адресом.
* *getExamResults* – добавља резултате испита за корисника са датом адресом и улогом.
* *getNotRegisteredExams* – добавља све испите које студент са датом адресом није пријавио.

Поред ових функција, налази се још једна помоћна функција чија улога јесте да провери да ли је корисник са одређеном адресом пријавио испит са датом шифром. Програмски код ове функције се налази на слици 5.2.11.



Слика 5.2.11 помоћна функција за проверу пријаве испита

У оквиру паметног уговора налазе се 2 главне функције:

* *registerExam* – функција за пријаву испита. Уколико су прошле све провере *modifier-a,* проналази се вредност *studentExamResults* за студента који је започео пријаву и вредност *professorExamResults* за професора који држи пријављени испит. Овим вредностима се додаје *ExamResult* објекат са оценом 0 што указује на неоцењени испит. На крају функције емитује се догађај *examRegistered.*
* *gradeExam* – функција за оцењивање испита. Уколико су прошле све провере *modifier*-a, проналази се испит са прослеђеном шифром и адресом студента који је пријавио испит. Након што професор упише оцену, ажурира се резултат тог испита у *studentExamResults* за студента који је пријавио испит као и резултат испита у *professorExamResults* за професора који је оценио испит. На крају функције емитује се догађај *examGraded.*

Ове 2 функције су приказане на слици 5.2.12.



Слика 5.2.12 главне функције

Након што је написан паметни уговор, неопходно је подесити конфигурацију за постављање истог на блокчејн мрежу. Сва нужна конфигурација се налази у *hardhat.config.js* датотеци. Податке које је неопходно дефинисати овде су:

* *defaultNetwork* – Подразумевана мрежа за постављање уговора.
* *networks* – низ мрежа на које се може поставити уговор, свака мрежа дефинисана у овој ставци мора садржати:
  + *chainId* – идентификатор ланца који се користи за валидацију повезане мреже.
  + *url –* линк ка коришћеном чвору.
  + *accounts* – скуп налога који *Hardhat* користи.
* *etherscan* – подаци о *Etherscan-у*, мора садржати бар атрибут:
  + *apiKey* – *API* кључ *Etherscan* налога.
* *solidity* – верзија *Solidity* компајлера.
* *namedAccounts* – омогућава асоцирање имена са адресама и конфигурисање по ланцу користи се најчешће за дефинисање деплојера и учесника.

На слици 5.2.13 налази се подешен конфигурациони фајл за сврху решења.



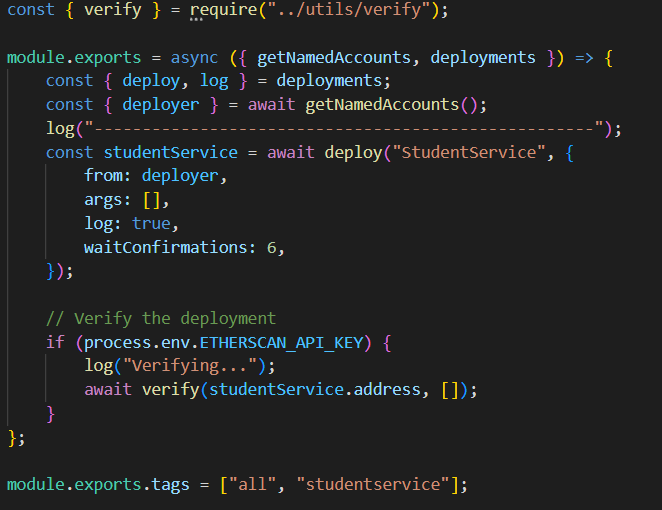
Слика 5.2.13 *hardhat.config.js*

Променљиве *ALCHEMY\_API\_KEY, ETHERSCAN\_API\_KEY* и *PRIVATE\_KEY* су извучене из *.env* датотеке.

Након подешавања конфигурационог фајла, могу се написати скрипте за постављање уговора у оквиру *deploy* директоријума. Срећом, *Hardhat* нуди функцију за постављање паметног уговора на мрежу која такође користи битне информације дефинисане у конфигурационом фајлу, програмер једино мора да проследи 2 аргумента:

* назив паметног уговора.
* *објекат* који садржи следеће атрибуте:
  + *from -*  налог који је задужен за постављање уговора.
  + *args –* аргументи паметног уговора.
  + *log –* да ли корисник жели испис логова приликом постављања.
  + *waitConfirmations –* број потврда које треба сачекати након које се паметни уговор укључи у ланац.

Целокупна логика за постављање уговора налази се у *deploy-student-service.js* датотеци. Садржај ове датотеке приказан је на слици 5.2.14.



Слика 5.2.14 *deploy-student-service.js*

Након постављања уговора, врши се и верификација уговора на *Etherscan-у*. Направљена је *verify.js* скрипта у *utils* директоријуму. У оквиру ове скрипте користи се *Hardhat* уграђена функција за верификацију уговора на коју је придодато руковање изузецима. Ова скрипта је приказана на слици 5.2.15.



Слика 5.2.15 *verify.js*

У оквиру *deploy* директоријума налази се и скрипта *deploy-setup-frontend.js* која се користи за прослеђивање *Application Binary Interface* (*ABI) [36]* и адресе паметног уговора фронтенд делу апликације. Ова скрипта је приказана на слици 5.2.16.



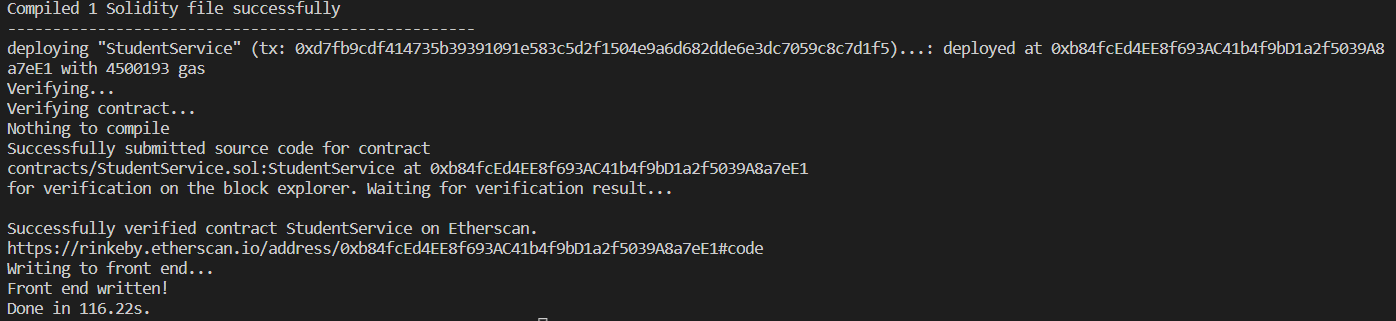
Слика 5.2.16 *deploy-setup-frontend.js*

Како бисмо урадили компајлирање, постављање и верификацију паметног уговора, довољно је унети команду као на слици 5.2.17.



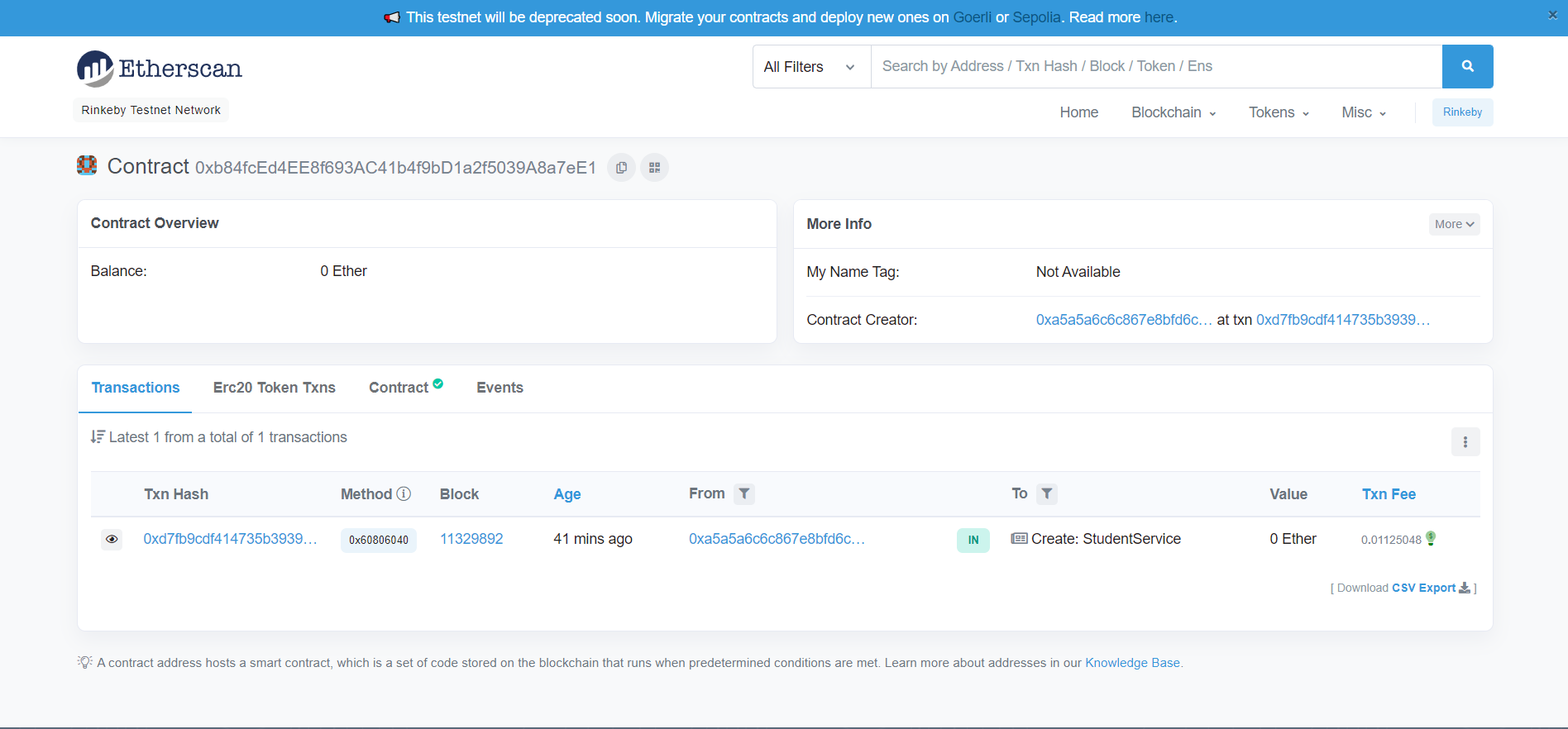
Слика 5.2.17 команда за постављање

Уколико не дође до грешке, на конзоли ће бити испит представљен на слици 5.2.18.



Слика 5.2.18 конзолни испис након постављања

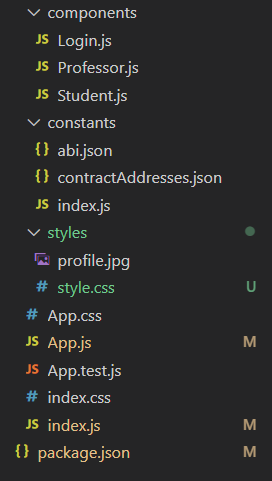
Након постављања, паметни уговор би требао бити доступан на *Etherscan* платформи. Приказ уговора на *Etherscan-у* је представљен на слици 5.2.19.



Слика 5.2.19 паметни уговор на *Etherscan* платформи

## Фронтенд део апликације

За развој фронтенд дела апликације, коришћена је једна од најпознатији *JavaScript* библиотека: *React [37].* Ова библиотека је по природи декларативна, ефикасна и флексибилна чија је главна употреба за развој корисничког интерфејса. Структура *React* пројекта је представљена на слици 5.3.1.



Слика 5.3.1 структура *React* пројекта

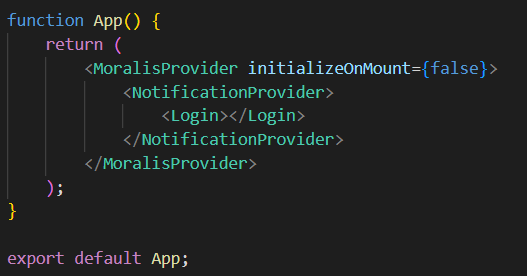
Важни елементи пројекта су:

* *components* – директоријум у којем се налазе кључни елементи сваког *React* програма звани компоненте чија је улога изградња корисничког интерфејса.
* *constants* – директоријум у којем се налазе *ABI* и адреса паметног уговора, ове информације су добијене из *deploy-setup-frontend.js* која је описана у претходном одељку.
* *styles –* директоријум у којем се налазе датотеке коришћене за стилизовање компоненти.
* *App.js –* главна компонента, компонента која се налази на врху хијерархије.
* *index.js* – улазна тачка апликације*.*

Нажалост, *React* као библиотека нема уграђену подршку за рад са блокчејном и паметним уговорима, због чега је било неопходно инсталирати наредне зависности:

* *react-moralis [38]* – библиотека која олакшава развој децентрализованих апликација. Пружа функционалисти уз помоћу којих могу да се позивају функције постављеног паметног уговора, добави стање повезаног новчаника са страницом итд. За потребе пројекта коришћене су наредне функционалности ове библиотеке:
  + *useMoralis* – функционалност која добавља информације о повезаном налогу као што су адреса налога, да ли је налог аутентификован итд.
  + useWeb3Contract – функционалност која омогућава комуникацију са паметним уговором.
* *web3uikit [39]* – библиотека која пружа стилизиране и лагане компоненте корисничког интерфејса за развој *Web 3.0* апликација. За потребе пројекта коришћене су наредне функционалности ове библиотеке:
  + *ConnectButton* – прилагођена компонента која има имплементирану логику за повезивање криптоновчаника са страницом.
  + *useNotification* – компонента обавештења која се приказује након успешно или неуспешне операције*.*

Како би *React* апликација могла да користи функционалности *react-moralis* библиотеке, неопходно је садржај *App.js* датотеке умотати *MoralisProvider* компонентом. Такође, како би могле да се користе нотификације које пружа *web3uikit* библиотека, потребно је додатно умотати коренску компоненту *NotificationProvider* компонентом. Структура *App.js* фајла је приказана на слици 5.3.2.

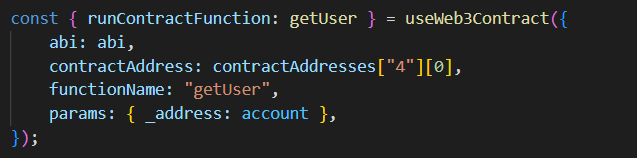


Слика 5.3.2 *App.js*

Како би се позвале функције постављеног паметног уговора из прошлог одељка, користи се *useWeb3Contract* функционалност коју пружа *react-moralis* библиотека. За коришћење ове функционалности, неопходно је дефинисати жељени назив функције у оквиру компоненте и проследили наредне параметре:

* *abi* – представља *ABI* паметног уговора.
* *contractAddress* – представља адресу постављеног паметног уговора.
* *functionName* – представља назив функције која се позива из паметног уговора.
* *params* – представља параметре функције паметног уговора.
* *msgValue* – представља стринг вредност количине *WEI (*криптовалута која је 1018пута мања од *ETH*)која се шаље паметном уговору (опционо поље).

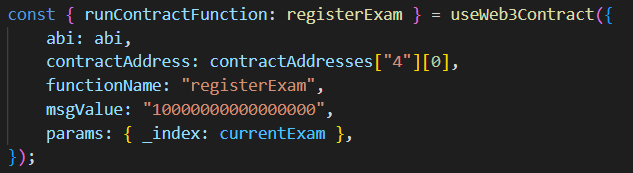
На слици 5.3.3 налази се изглед функције за добављање повезаног корисника која се налази у *Login.js* компоненти.



Слика 5.3.3 *getUser* функција

Вредности параметара *abi* и *contractAddress* су добављени из *constants* директоријума, док је вредност параметра ­\_addressпроменљива *account.* Ова променљива је извучена из *useMoralis*  функције и представља адресу тренутно повезаног новчаника.

*useWeb3Contract* је најчешће коришћена функционалност, јер омогућава позивање свих функција које су дефинисане у паметном уговору. Ради сажетости, неће бити приказани сви делови кода где је ова функционалност коришћена, него ће само бити представљена употреба useWeb3Contract за 2 главне функције, пријава испита и оцењивање испита, на сликама 5.3.4 и 5.3.5 респективно.



Слика 5.3.4 пријава испита



Слика 5.3.5 оцењивање испита

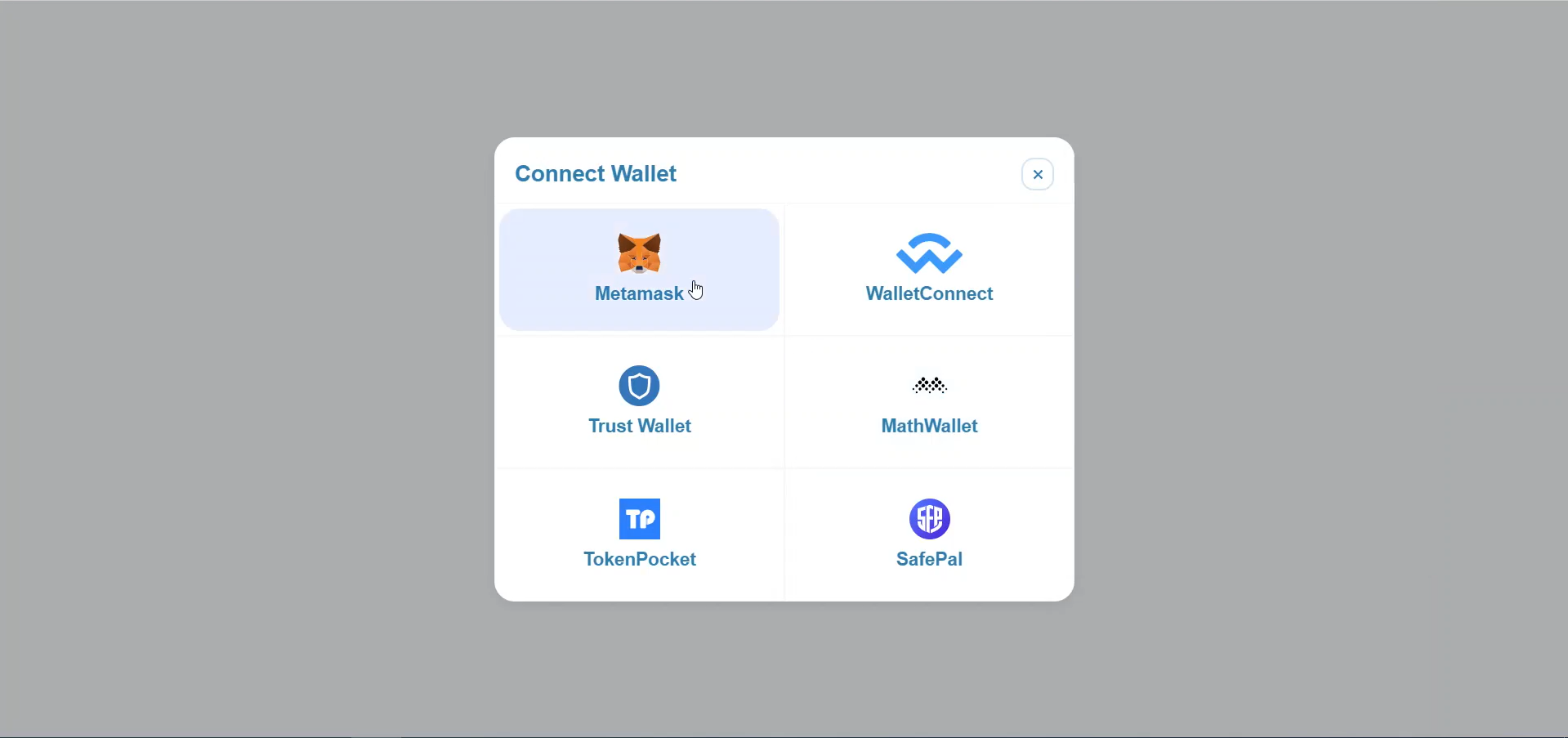
## Финална апликација

У овом поглављу биће описана апликација у њеној целости. Приликом првог уласка на апликацију, кориснику је приказана страница за пријаву. Ова страница је приказана на слици 5.4.1.



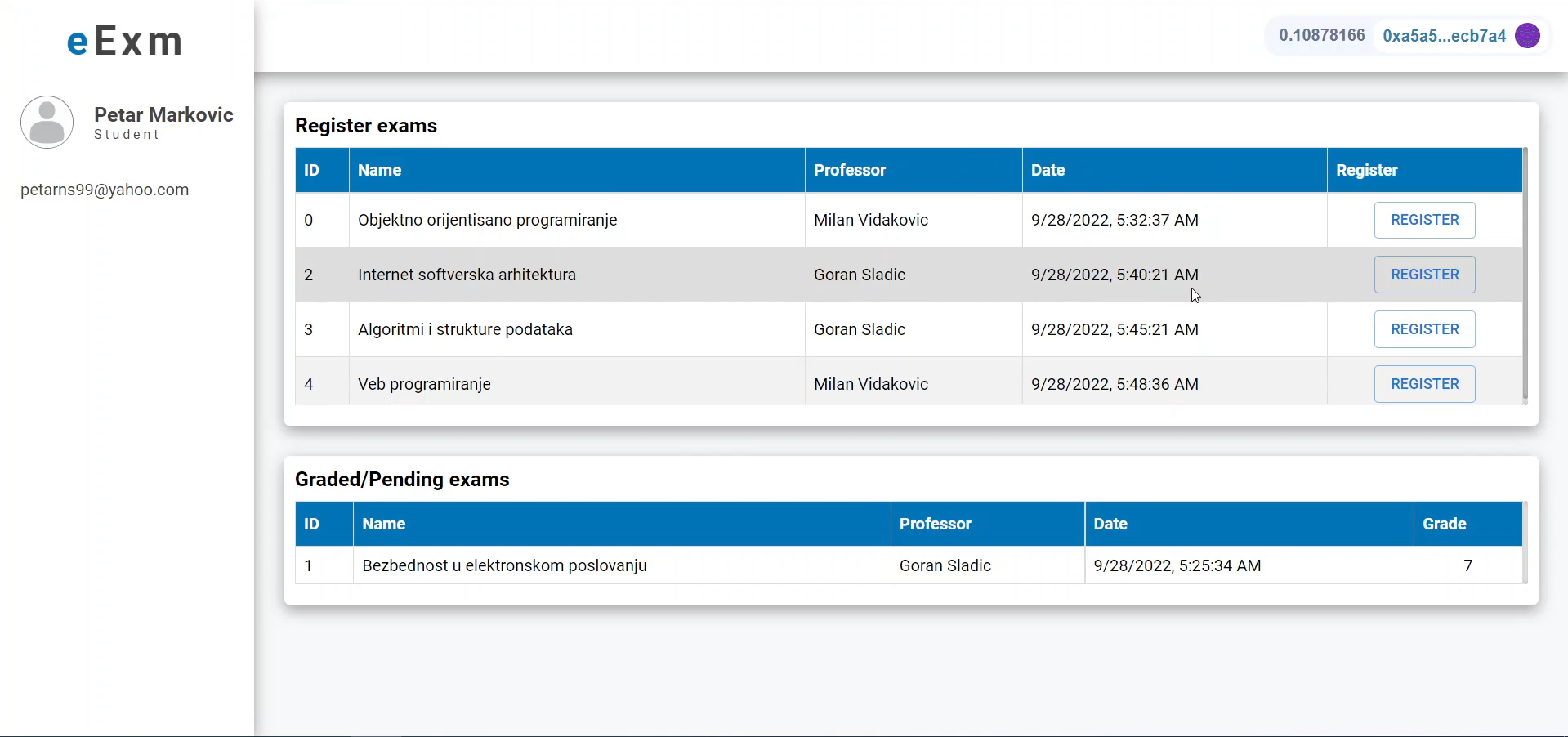
Слика 5.4.1 страница за пријаву

Притиском на дугме *Connect Wallet,* кориснику се приказује дијалог где може да изабере конкретну врсту криптоновчаника коју жели да повеже са апликацијом. Овај дијалог је представљен на слици 5.4.2.



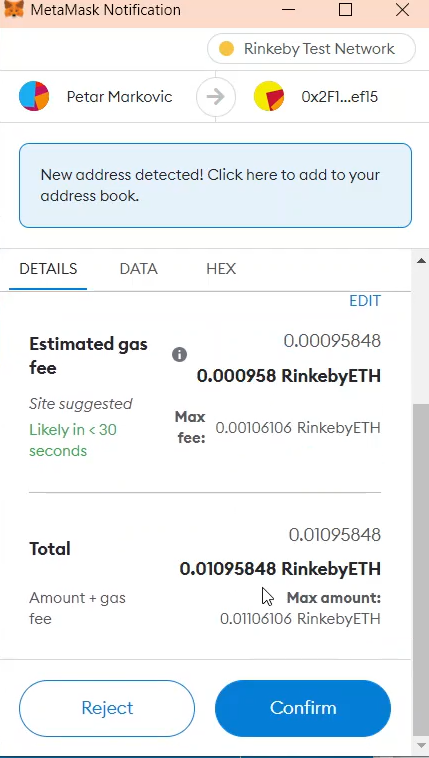
Слика 5.4.2 дијалог за избор новчаника

Након одабира новчаника, уколико су подаци о повезаном налогу сачувани у оквиру паметног уговора, прелази се на главну страницу. Ако је аутентификовани корисник студент, ова страница изгледа као на слици 5.4.3.



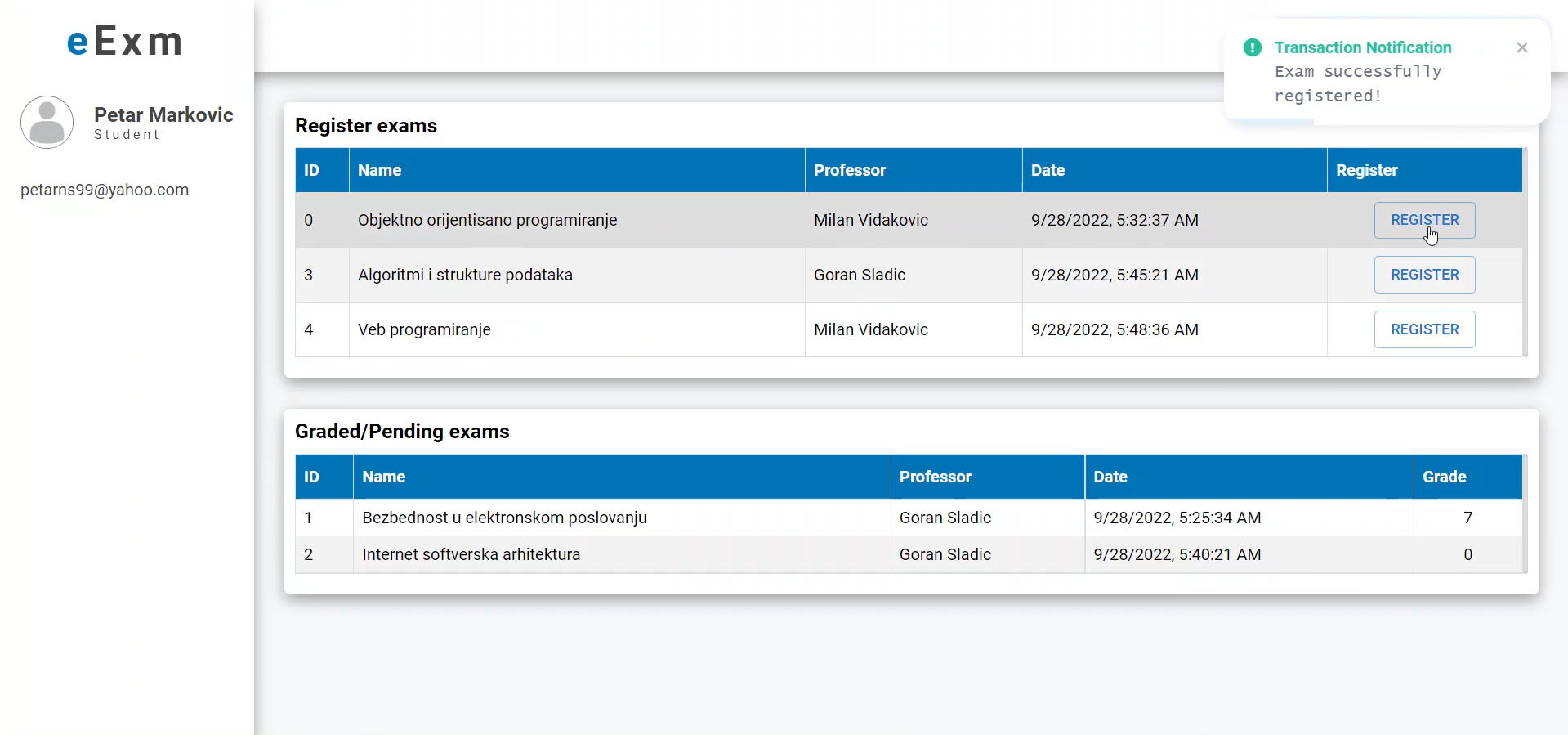
Слика 5.4.3 главна страница студента

У оквиру ове странице, горња табела представља испите које корисник може да пријави, док је доња табела скуп оцењених и пријављених испита. Такође, у горњем десном ћошку је дугме које садржи информације о салду и адреси новчаника. Студент може да пријави било који испит у горњој табели притиском на дугме *REGISTER* у оквиру колоне жељеног испита. Након притиска овог дугмета, отвара се обавештење у оквиру новчаника где корисник треба да потврди да ли жели да обави ову трансакцију у износу од 0.01 *ETH.* Ово обавештење је описано на слици 5.4.4.



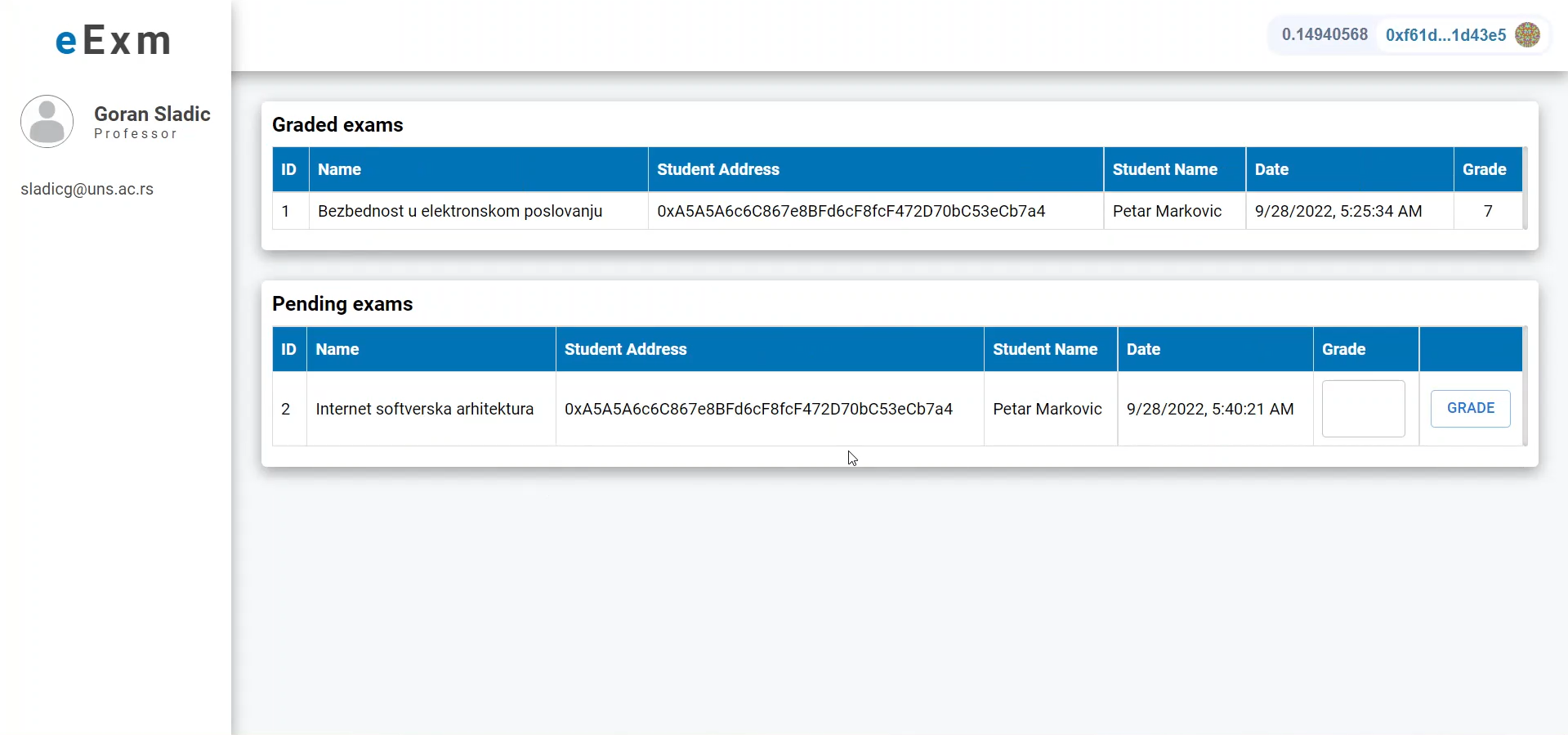
Слика 5.4.4 обавештење новчаника

Уколико студент притисне на дугме *Confirm*, мора сачекати одређено време да се ова трансакција потврди. Након што се ова пријава сачува на блокчејн, кориснику стиже нотификација. Овај ток се може видети на слици 5.4.5.



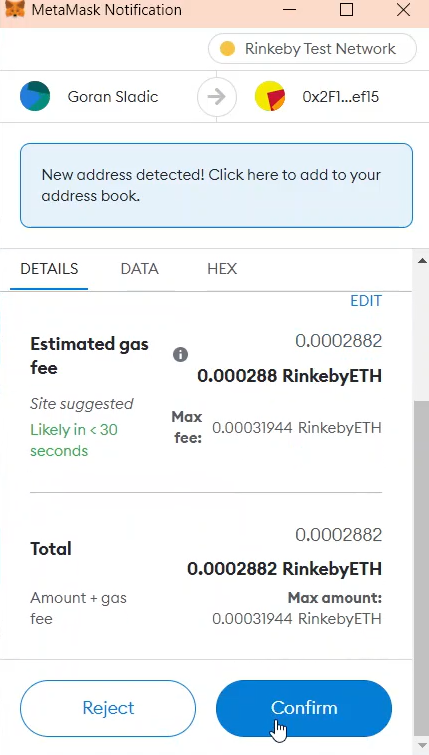
Слика 5.4.5 успешно пријављен испит

Ако се као корисник пријави професор, његова главна страна ће изгледати другачије и она је приказана на слици 5.4.6.



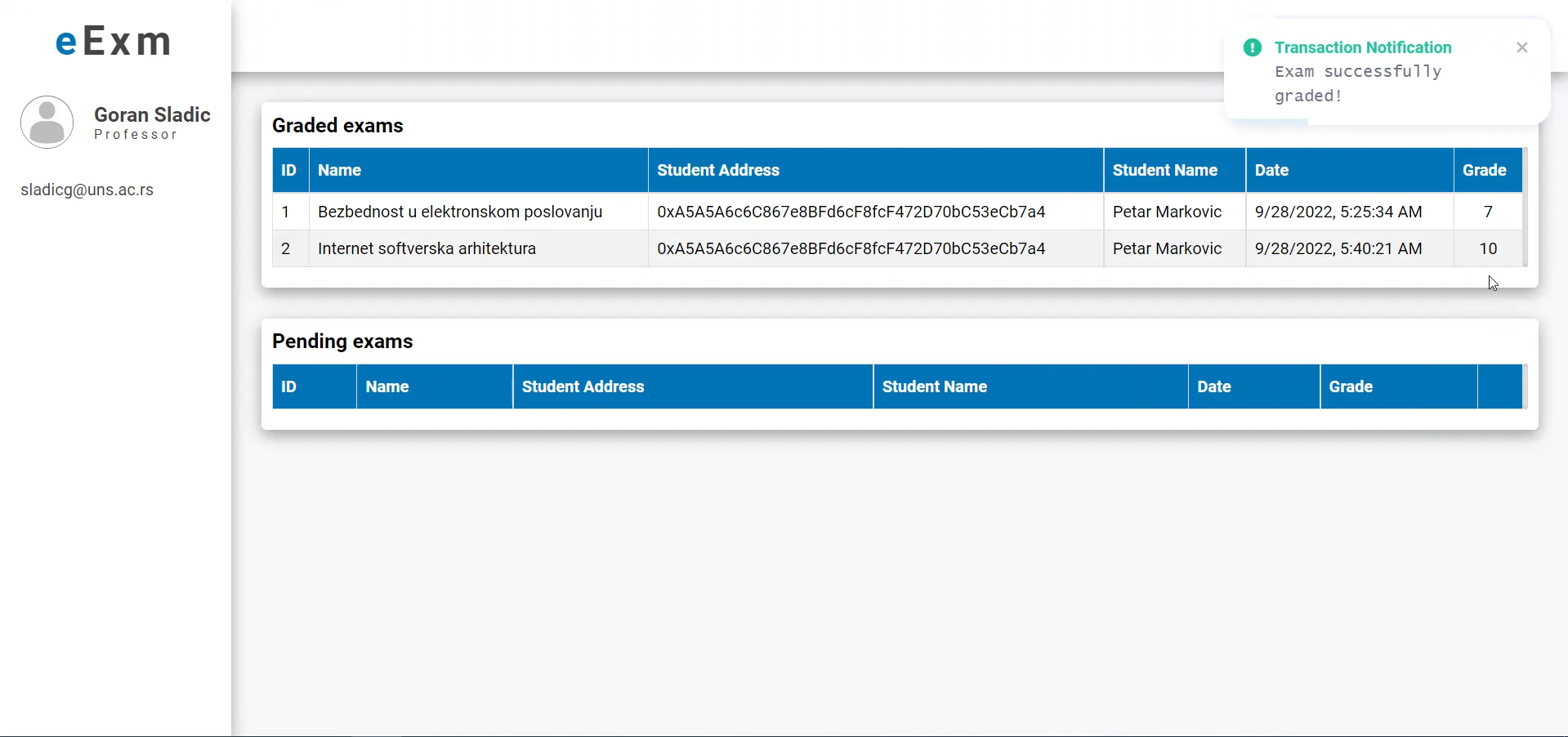
Слика 5.4.6 главна страница професора

У оквиру ове странице, горња табела представља испите које је корисник, док је доња табела скуп пријављених испита који нису оцењени. У доњој табели професор може да оцени одговарајући испит уносом оцене у претпоследњој колони и притиском на дугме *GRADE* у последњој колони*.* Након притиска овог дугмета, отвара се обавештење у оквиру новчаника где корисник треба да потврди да ли жели да обави ову трансакцију и плати само одређену количину гаса. Ово обавештење је описано на слици 5.4.7.



Слика 5.4.7 обавештење новчаника

Уколико професор притисне на дугме *Confirm*, мора сачекати одређено време да се ова трансакција потврди. Након што се овај оцењени испит сачува на блокчејн, кориснику стиже нотификација. Овај ток се може видети на слици 5.4.8.



Слика 5.4.8 успешно оцењен испит

# Закључак

У уводу рада су обрађене различите врсте система са великим фокусом на централизоване и децентрализоване системе. Направљено је директно поређење између централизованих и децентрализованих система. Закључак је да је главна разлика између ова 2 приступа у самом нивоу контроле, где централизовани систем има једно ауторитативно тело које управља свим ресурсима док у оквиру децентрализованог система не постоји централни власник, него сваки чвор складишти своје ресурсе. Поред ових структура, описан је и дистрибуирани систем. Дистрибуирани систем је дефинисан као рачунарска средина где су све компоненте расподељене по рачунарима у оквиру једне заједничке мреже.

Како је рад у великој мери имплементиран употребом блокчејн технологије, било је неопходно издвојити један сегмент који објашњава концепте и теоријске основе исте. Блокчејн је јавно доступна дистрибуирана главна књига. Предност блокчејн технологије јесте његова безбедност која је постигнута механизмом који се назива консензус алгоритам. Ово је поступак уз помоћ ког сви чворови блокчејн мреже могу да направе компромис о стању самог блокчејна. Први пут се блокчејн као технологија користи за сврхе платформе *Bitcoin* која је употребом криптографских доказа формирала систем плаћања криптовалутама. Ова блокчејн платформа је туринг непотпуна, јер осим вршења трансакција, не пружа могућност обављања комплекснијих задатака. Као алтернатива, креиране су платформе које су омогућиле уметање програмског кода и логике у сам блокчејн. Једна таква платформа је *Ethereum.* *Ethereum* технологија има свој посебан програмски језик по називу *Solidity.* *Solidity* се користи за писање паметних уговора. Паметни уговори су програми који се извршавају на *Ethereum* блокчејну.

У овом раду је имплементирана децентрализована апликација студентске службе. Како би се олакшао развој, тестирање, компајлирање и постављање паметног уговора који представља основу апликације, коришћена је *HardHat* развојна платформа. Како би се постигла комуникација са овим паметним уговором који је постављен на *Ethereum* блокчејн, на клијентској страни је било неопходно подесити *Moralis* платформу.

Правци даљег развоја апликације би били фокусирани на креирање компатабилне мобилне апликације. Поред тога, потенцијално би се могло додати још функционалности или проширити постојеће (отказивање испита, овера семестра, одбрана дипломског рада итд.). Такође, са становишта оптимизације, сам паметни уговор би се могао рефакторисати тако да захтева мање гаса. Ово се може постићи реформатирањем кода или складиштењем података о корисницима и испитима ван самог ланца.

# Литература

1. Centralized vs Decentralized Logistic Distribution, [https://www.analyticssteps.com/blogs/centralized-vs-decentralized-logistic-distribution](https://www.analyticssteps.com/blogs/centralized-vs-decentralized-logistic-distribution%20)
2. Decentralized Vs. Centralized: A Detailed Comparison, [https://www.blockchain-council.org/blockchain/decentralized-vs-centralized/](https://www.blockchain-council.org/blockchain/decentralized-vs-centralized/%20)
3. What Are the Pros and Cons of Centralized and Decentralized Distribution, [https://parcelindustry.com/article-5707-What-Are-the-Pros-and-Cons-of-Centralized-and-Decentralized-Distribution.html](https://parcelindustry.com/article-5707-What-Are-the-Pros-and-Cons-of-Centralized-and-Decentralized-Distribution.html%20%20)
4. Decentralized vs Centralized: A Complete Comparison,

[https://101blockchains.com/decentralized-vs-centralized/](https://101blockchains.com/decentralized-vs-centralized/%20)

1. Distributed Systems - The Complete Guide,

<https://www.confluent.io/learn/distributed-systems/>

1. What Are Distributed Systems,

[https://www.splunk.com/en\_us/data-insider/what-are-distributed-systems.html](https://www.splunk.com/en_us/data-insider/what-are-distributed-systems.html%20)

1. Blockchain Facts: What Is It, How It Works, and How It Can Be Used,

[https://www.investopedia.com/terms/b/blockchain.asp](https://www.investopedia.com/terms/b/blockchain.asp%20)

1. Haber, S., Stornetta, W.S. How to time-stamp a digital document. J. Cryptology 3, 99–111 (1991),

<https://link.springer.com/content/pdf/10.1007/BF00196791.pdf>

1. Bitcoin A Peer-to-Peer Electronic Cash System,

<https://bitcoin.org/bitcoin.pdf>

1. What is blockchain technology,

[https://www.ibm.com/topics/what-is-blockchain](%20https:/www.ibm.com/topics/what-is-blockchain)

1. What Is Blockchain Technology How Does It Work,

[https://builtin.com/blockchain](%20https:/builtin.com/blockchain)

1. Benefits of blockchain,

<https://www.ibm.com/topics/benefits-of-blockchain>

1. Consensus Algorithm,

[https://www.techtarget.com/whatis/definition/consensus-algorithm](%20https:/www.techtarget.com/whatis/definition/consensus-algorithm)

1. Consensus Algorithms in Blockchain,

[https://www.geeksforgeeks.org/consensus-algorithms-in-blockchain/](%20https:/www.geeksforgeeks.org/consensus-algorithms-in-blockchain/)

1. Consensus Mechanism (Cryptocurrency), <https://www.investopedia.com/terms/c/consensus-mechanism-cryptocurrency.asp>
2. A Guide to Understand Blockchain Consensus Algorithms, <https://appinventiv.com/blog/blockchain-consensus-algorithms-guide/>
3. Ethereum,

<https://ethereum.org/en/>

1. Turing Complete,

<https://academy.binance.com/en/glossary/turing-complete>

1. What Is Ethereum,

<https://www.investopedia.com/terms/e/ethereum.asp>

1. Introduction to smart contracts, <https://ethereum.org/en/developers/docs/smart-contracts/>
2. Smart Contracts Definition,

<https://www.investopedia.com/terms/s/smart-contracts.asp>

1. What are smart contracts on blockchain,

<https://www.ibm.com/topics/smart-contracts>

1. Еthereum Virtual Machine,

<https://ethereum.org/en/developers/docs/evm/>

1. What Is an Ethereum Virtual Machine (EVM), <https://coinmarketcap.com/alexandria/glossary/ethereum-virtual-machine-evm>
2. Everything you need to know about Ethereum Virtual Machine (EVM), <https://www.analyticssteps.com/blogs/everything-you-need-know-about-ethereum-virtual-machine-evm>
3. What is Web 1.0, 2.0, and 3.0 and How They Compare, <https://www.simplilearn.com/what-is-web-1-0-web-2-0-and-web-3-0-with-their-difference-article>
4. What are Web1, Web2, and Web3 in simple terms, <https://www.cryptopolitan.com/what-are-web1-web2-and-web3-in-simple-terms/>
5. Web 2 vs. Web 3: What’s the Difference and Why It Matters, <https://www.spiceworks.com/tech/tech-general/articles/web-2-vs-web-3/>
6. MetaMask: The crypto wallet for Defi, Web3 Dapps and NFTs, <https://metamask.io/>
7. Ethereum Test Networks, <https://docs.ethhub.io/using-ethereum/test-networks/>
8. Alchemy, <https://www.alchemy.com/>
9. Etherscan, <https://etherscan.io/>
10. Hardhat, <https://hardhat.org/>
11. Solidity, <https://docs.soliditylang.org/en/v0.8.16/>
12. Ether, <https://ethereum.org/en/eth/>
13. ABI, <https://www.quicknode.com/guides/solidity/what-is-an-abi>
14. React, <https://reactjs.org/>
15. react-moralis, <https://www.npmjs.com/package/react-moralis>
16. Web3UI Kit, <https://web3uikit.com/>
17. Centralized vs Decentralized vs Distributed,

<https://medium.com/delta-exchange/centralized-vs-decentralized-vs-distributed-41d92d463868>

1. What is blockchain, <https://money.com/what-is-blockchain/>
2. Bitcoin vs Ethereum Smart Contracts,

<https://xiaohuiliu.medium.com/bitcoin-vs-ethereum-smart-contracts-921e0a12b043>

1. Sailian blockchain training: the core elements of Web3 - blockchain, encrypted assets, smart contract and Oracle, <https://qdmana.com/2022/03/202203040140511860.html?fbclid=IwAR2BBw9XZOINZqog-OhQw9WvwatcIU51ZTHw2ms6A0fec4qGT-CYT1L1kuY>

# Биографија

Петар Марковић рођен је 29.10.1999. године у Нишу. Похађао је основну школу „Прва војвођанска бригада“ у Новом Саду, коју је завршио 2014. године, а затим је уписао општи смер гимназије „Светозар Марковић“, такође у Новом Саду. Средње образовање је стекао 2018. године, и исте године је уписао Факултет техничких наука, смер софтверско инжењерство и информационе технологије. Положио је све испите предвиђене планом и програмом са просечном оценом 9.20.