Push-ilmoitusten integrointi Flutter-mobiilisovellukseen

Santeri Kallio

Opinnäytetyö, AMK
Joulukuu 2023
Tieto- ja viestintätekniikan tutkinto-ohjelma

jamk | Jyväskylän ammattikorkeakoulu
University of Applied Sciences

jamk

Kuvailulehti

Kallio, Santeri

Push-ilmoitusten integrointi Flutter-mobiilisovellukseen

Jyväskylä: Jyväskylän ammattikorkeakoulu. Joulukuu 2023, 37 sivua

Tieto- ja viestintätekniikan tutkinto-ohjelma. Opinnäytetyö AMK.

Julkaisun kieli: suomi

Julkaisulupa avoimessa verkossa: kyllä

Tiivistelmä

Nykypäivinä useista mobiilisovelluksista löytyy ominaisuus lähettää push-ilmoituksia käyttäjän mobiililaitteeseen. Jyväskylän yliopiston digipalveluilla heräsi tarve jatkokehittää heidän Flutterilla toteutettua MyJYU-nimistä mobiilisovellusta, lisäämällä siihen juuri tämä ominaisuus. Push-ilmoitus-ominaisuuden odotettiin tuottavan sovellukselle lisäarvoa ja tarjoavan käyttäjille paremman käyttökokemuksen. Toimeksiannoksi muodostui sekä ominaisuuden suunnittelu että toteutus hyödyntämällä monipuolisesti eri teknologioita.

Ominaisuuden suunnittelu alkoi tutkimalla, miten ja millä teknologioilla olisi mahdollista tehokkaasti toteuttaa ja integroida push-ilmoitus-ominaisuus olemassa olevaan mobiilisovellukseen. Tutkimuksen päätteeksi päädyttiin hyödyntämään Googlen tarjoamaa Firebase Cloud Messaging -pilvipalvelua ilmoitusten toimittamiseen. Tämän lisäksi nousi tarve kehittää myös kaksi taustapalvelua Pythonia hyödyntämällä, sekä luonnollisesti MyJYU-sovellus vaati muutoksia koodiinsa myös. Ominaisuuden vaatimukset dokumentoitiin tarkasti, jotta kehittämisprosessi kulkisi mahdollisimman sujuvasti.

Työn toteutus alkoi taustapalveluiden kehittämisellä. Ensimmäisenä oli toteutettava rajapintasovellus, joka hallinnoi käyttäjien mobiililaitteiden Firebase-tunnisteiden linkittämisen heidän Jyväskylän yliopiston käyttäjätileihin. Toisena oli kehitettävä taustapalvelu, joka kuuntelee viestijonoa laitteisiin lähetettävistä ilmoituksista ja lähettää nämä Firebase Cloud Messaging -palvelun kautta käyttäjien laitteisiin. Lopuksi MyJYU-sovellukseen oli lisättävä tuki rajapintasovellukseen yhdistämisestä ja ilmoitusten vastaanottamiselle Firebasesta. Työn aikana noudatettiin ohjelmoinnin hyviä käytänteitä ja työ dokumentoitiin kokonaisuudessaan.

Työn tuloksina saatiin valmiiksi vaaditut taustapalvelut ja MyJYU-sovellukseen tuki vastaanottaa ilmoituksia. Ominaisuuden pohjalta avautuu erinomaiset mahdollisuudet jatkokehittämiselle ja uusille käyttötapauksille, sillä ilmoitusta varten tarvitsisi lähettää vain ilmoitus viestijonoon, jonka jälkeen taustapalvelut hoitaisivat loput.

Avainsanat (asiasanat)

Firebase, Flutter, Mobiilikehitys, push-ilmoitus, Python, viestijono

Muut tiedot (salassa pidettävät liitteet)

N/A

jamk

Description

Kallio, Santeri

Integrating push notifications to a Flutter mobile app

Jyväskylä: JAMK University of Applied Sciences, December 2023, 37 pages

Degree Programme in Information and Communication Technology. Bachelor´s thesis.

Permission for open access publication: Yes

Language of publication: Finnish

Abstract

Nowadays, many mobile applications feature the capability to send push notifications to users' mobile devices. The digital services at the University of Jyväskylä recognized the need to enhance their mobile application named MyJYU, implemented with Flutter, by incorporating this functionality. The addition of the push notification feature was expected to provide added value to the application and offer users and improved user experience. The task consisted of both the design and implantation of the feature, utilizing various technologies extensively.

The design process commenced with an exploration of how and with which technologies it would be best to effectively implement and integrate the push notification feature into the existing mobile application. After research, the decision was made to leverage Google's Firebase Cloud Messaging service for delivering notifications. Additionally, there was a need to develop two backend services using Python, and naturally, the MyJYU application required modifications to its code as well. The requirements of the feature were meticulously documented to ensure a smooth development process.

The implementation phase began with the development of the background services. Firstly, an API application had to be implemented to manage the linking of users' Firebase tokens to their University of Jyväskylä user accounts. Secondly, a background service had to be created to listen to the message queue for notifications to be sent to devices and dispatch these through the Firebase Cloud Messaging service to users' devices. Finally, support for connecting to the API application and receiving notifications from Firebase had to be added to the MyJYU application. Throughout the process, the best practices of programming were followed, and the work was comprehensively documented.

The results of the project included the completion of the required backend services and the addition of support in the MyJYU application for receiving notifications. The feature opens up excellent possibilities for further development and use cases, as it would only be necessary to send a notification to the message queue, after which the backend services would take care of the rest.

Keywords/tags (subjects)

Firebase, Flutter, message queue, Mobile development, Push notification, Python

Miscellaneous (Confidential information)

N/A

Sisältö

1 Lähtökohdat ... 3
1.1 Toimeksiantaja ... 3
1.2 Tavoitteet ... 3
1.3 Tutkimuksellinen kehittämistyö ... 4
2 Push-ilmoitukset ... 4
2.1 Määritelmä ... 4
2.2 Hyödyt ... 5
3 Teknologiavalinnat ... 6
3.1 Yleistä ... 6
3.2 Flutter ... 6
3.3 Firebase Cloud Messaging ... 6
3.4 Python ... 7
3.5 RabbitMQ ... 8
3.6 Docker ... 8
3.7 PostgreSQL ... 8
4 CASE: MyJYU-mobiilisovelluksen ilmoitukset ... 8
4.1 Suunnittelu ... 8
4.1.1 PostgreSQL-tietokanta ... 9
4.1.2 Laiterekisteröintipalvelu ... 9
4.1.3 Ilmoitusten lähettelijä ... 10
4.1.4 MyJYU frontend ... 10
4.2 Toteutus ... 11
4.2.1 Yleistä ... 11
4.2.2 Laiterekisteröintipalvelu ... 11
4.2.3 Ilmoitusten lähettelijä ... 18
4.2.4 MyJYU frontend ... 28
4.3 Testaus ja julkaisu ... 34
5 Tulokset ... 34
6 Pohdinta ... 35
Lähteet ... 37

Kuviot

Kuvio 1. Esimerkki-ilmoitus iOS ja Android käyttöjärjestelmillä. 5
Kuvio 2. FCM:n arkkitehtuurillinen kuvaus. 7
Kuvio 3. Tietokantakuvaus. 9
Kuvio 4. Ohjelman luonti ja riippuvuudet 11
Kuvio 5. Tietokantayhteyden muodostus. 12
Kuvio 6. Schemas.py tiedosto. 13
Kuvio 7. "register-fcm-token"-rajapintaosoite. 14
Kuvio 8. "get_user_info"-metodi. 15
Kuvio 9. "handle_missing_keys" -metodi. 15
Kuvio 10. "update_existing_user" -metodi. 16
Kuvio 11. "insert_new_user" -metodi. 16
Kuvio 12. "get-topic"-rajapintaosoite. 17
Kuvio 13. Dockerfile 18
Kuvio 14. Firebase Admin SDK:n initialisointi 19
Kuvio 15. Schemas.py -tiedosto. 20
Kuvio 16. Tietokantyhteys-metodi aikakatkaisulla. 21
Kuvio 17. RabbitMQ-yhteys. 22
Kuvio 18. Callback-metodi. 23
Kuvio 19. send_notification_to_user -metodi. 24
Kuvio 20. get_fcm_tokens_and_locale_for_user -metodi. 25
Kuvio 21. send_notification_and_get_valid_tokens -metodi. 25
Kuvio 22. get_localized_title_and_body -metodi. 25
Kuvio 23. construct_and_send_notification_to_token -metodi. 26
Kuvio 24. handle_unregistered_token -metodi. 26
Kuvio 25. send_notification_to_topic -metodi. 27
Kuvio 26. log_notification_responses -metodi. 28
Kuvio 27. Firebasen initialisointi. 28
Kuvio 28. Firebase-asetuksia. 29
Kuvio 29. send_fcm_token.dart -tiedosto. 30
Kuvio 30. Topicejen listaus sovelluksessa. 31
Kuvio 31. LifeCycleManager-widget. 32
Kuvio 32. _checkInternetConnection -metodi. 33
Kuvio 33. Ilmoitus MyJYU:ssa. 35

1 Lähtökohdat

1.1 Toimeksiantaja

Opinnäytetyön toimeksiantajana toimii Jyväskylän yliopisto (JYU), joka on yksi Suomen johtavista korkeakouluista. JYU:n digipalvelut, osana yliopistopalveluita, ovat vastuussa monenlaisten digitaalisten palveluiden tuottamisesta ja ylläpidosta. Heidän tehtävänään on tarjota teknistä tukea niin opiskelijoille kuin henkilökunnallekin, jotta nämä voivat hyödyntää digitaalisia työkaluja ja palveluita tehokkaasti (Jyväskylän yliopiston digipalveluiden verkkosivut n.d.).

Yksi JYU:n digipalveluiden kehittämistä sovelluksista on "MyJYU" -mobiilisovellus, joka on suunniteltu helpottamaan opiskelijoiden arkea tarjoamalla heille pääsy monipuolisiin palveluihin, kuten opiskelijaravintoloiden ruokalistoihin tai opiskelu- ja tapahtumakalenteriin (MyJYU-sovellus helpottaa arkeasi n.d.). Sovellus on tärkeä osa opiskelijoiden päivittäistä elämää ja toimii linkkinä heidän ja yliopiston tarjoamien resurssien välillä.

JYU:lla heräsi tarve parantaa MyJYU-sovellusta entisestään lisäämällä siihen push-ilmoitusominaisuus. Push-ilmoitukset voivat tarjota opiskelijoille entistä tehokkaamman tavan saada tärkeää ja ajankohtaista tietoa yliopiston tapahtumista, päivityksistä ja muista merkittävistä asioista. Tavoitteena on tehdä MyJYU-sovelluksesta vielä hyödyllisempi ja saavutettavampi opiskelijoiden arjessa. Opinnäytetyö pyrkii vastaamaan tähän tarpeeseen ja kehittämään push-ilmoitusominaisuuden MyJYU-sovellukselle niin, että se palvelee parhaiten opiskelijayhteisön tarpeita ja odotuksia.

1.2 Tavoitteet

Opinnäytetyön keskeisenä tavoitteena oli jatkokehittää Flutterilla toteutettua mobiilisovellusta lisäämällä siihen uusi ja hyödyllinen ominaisuus, push-ilmoitusten vastaanottamisen mahdollisuus. Tämä tavoite kattoi laajan skaalan teknisiä ja käyttäjäkeskeisiä näkökohtia, ja sen toteuttaminen vaati sekä mobiilisovelluksen että taustapalvelun kehittämistä.

Yksi merkittävimmistä haasteista oli taustapalvelun luominen, joka on vastuussa ilmoitusten lähettämisestä mobiililaitteisiin. Tämä taustapalvelu mahdollistaa ilmoitusten lähettämisen suurille

käyttäjäryhmille, kuten opiskelijoille tai henkilökunnalle, ja tarjoaa myös mahdollisuuden liittää käyttäjätunnukset mobiililaitteisiin. Tämä ominaisuus puolestaan mahdollistaa henkilökohtaisten ilmoitusten lähettämisen pelkän käyttäjätunnuksen perusteella, mikä on erityisen hyödyllistä esimerkiksi uusien opintosuoritusten ilmoittamisessa käyttäjille.

Keskeisenä tavoitteena push-ilmoitusten lisäämiselle on tarve tarjota käyttäjille parempi käyttäjäkokemus ja tuottaa lisäarvoa sovellukselle. Näiden tavoitteiden saavuttaminen edellyttää monipuolista osaamista mobiilisovelluskehityksestä ja taustapalveluiden hallinnasta. Opinnäytetyössä tullaan tarkastelemaan syvällisesti näitä osa-alueita ja niiden vaikutusta lopputulokseen. Lisäksi tutkimustyössä huomioidaan myös käyttäjien tarpeet ja odotukset, jotta push-ilmoitusominaisuus saadaan suunniteltua ja toteutettua mahdollisimman hyvin vastaamaan käyttäjien tarpeita.

1.3 Tutkimuksellinen kehittämistyö

Tutkimusmenetelmällisyys tässä opinnäytetyössä perustui käytännönläheiseen kehittämistyöhön, jonka tavoitteena oli luoda ja integroida push-ilmoitusominaisuus MyJYU-mobiilisovellukseen. Tämä tutkimuksellinen kehittämistyö yhdistää tutkimuksen ja käytännön sovelluskehityksen tarkoituksenaan tuottaa konkreettisia tuloksia ja ratkaisuja Jyväskylän yliopiston digipalveluiden tarpeisiin.

Tutkimuksen ensisijainen tarkoitus oli selvittää, miten push-ilmoitusominaisuus voidaan tehokkaimmin integroida MyJYU-sovellukseen ottaen huomioon käyttäjien tarpeet ja tekniset vaatimukset. Tämä edellyttää syvällistä tutkimusta mobiilisovelluskehityksen parhaista käytännöistä, push-ilmoitusjärjestelmistä ja käyttäjäkokemukseen vaikuttavista tekijöistä.

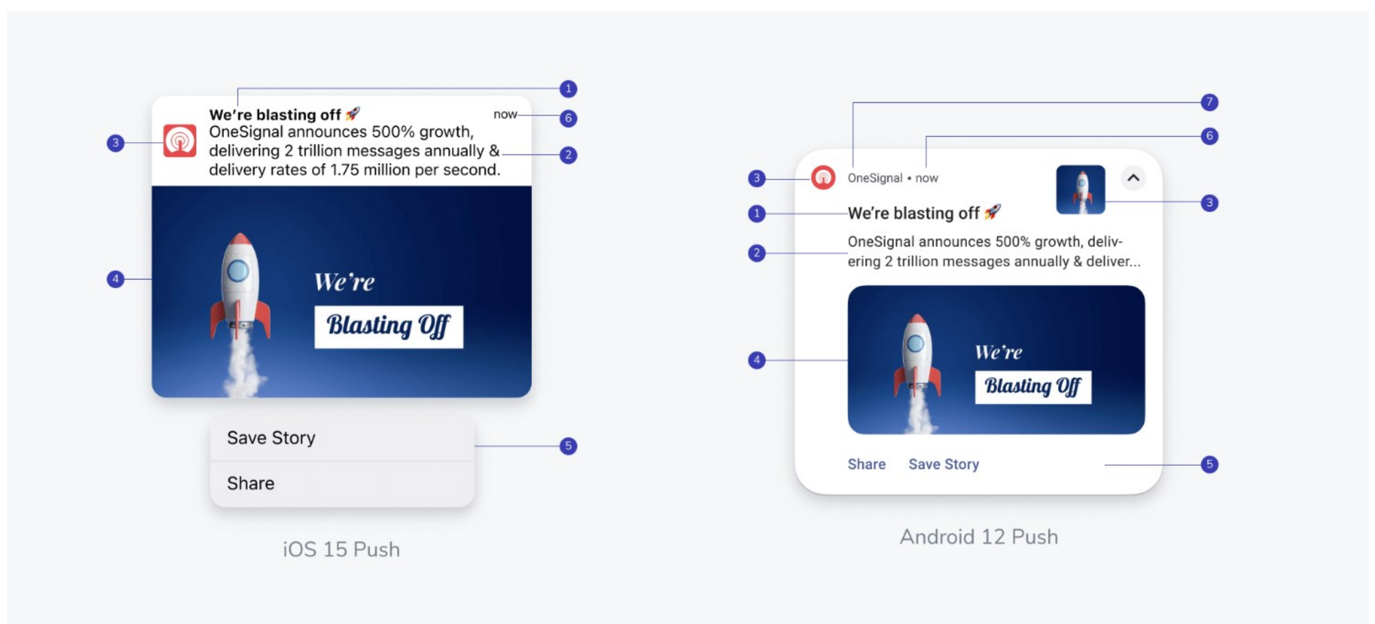
2 Push-ilmoitukset

2.1 Määritelmä

Push-ilmoitukset ovat käyttäjien laitteisiin tulevia sovelluskohtaisia ilmoituksia. Sovelluksen tai verkkosivun taustalla oleva palvelin ”työntää” (tästä nimi ”push”-ilmoitukset) viestejä suoraan käyttäjän laitteeseen ilman, että sovelluksen tarvitsee olla päällä, tai että käyttäjän tarvitsisi olla

vuorovaikutuksessa sovelluksen kanssa sillä hetkellä. Nykyisin verkkoselaimet ja yleisimmät mobiililaitteiden käyttöjärjestelmät, kuten Android ja iOS estävät oletuksena ilmoitusten vastaanottamisen, joka tarkoittaa sitä, että sovellusten tarvitsee pyytää käyttäjältä erikseen lupa ilmoitusten vastaanottamiseen (Mixon & Steele 2023).

Mobiilisovellusten push-ilmoitusten tavanomainen formaatti kuvion 1 mukaisesti sisältää otsikon (1), viestin sisällön (2), ikonin (3), aikaleiman (6) ja mahdollisesti kuvan (4) ja painikkeita (5), jotka sovelluksen mukaan tekevät eri asioita. Android käyttöjärjestelmää käyttävät mobiililaitteet myös näyttävät sovelluksen nimen (7) (OneSignal 2023).



Kuvio 1. Esimerkki-ilmoitus iOS ja Android käyttöjärjestelmillä (OneSignal 2023).

2.2 Hyödyt

Push-ilmoitukset ovat tehokas markkinointiväline, joka tarjoaa monia etuja yrityksille. Ne ovat lyhyitä, ytimekkäitä ja lähes mahdotonta ohittaa, mikä tekee niistä ihanteellisen välineen kohdennettuun ja aikariippuvaan viestintään. Korzeń (2022) kertoo blogijulkaisussaan, että AirShipin tekemän tutkimuksen mukaan push-ilmoitukset hyvin käytettynä voivat lisätä sovelluksen käyttäjien säilyttämistä jopa 190 %, mutta väärinkäytettynä ne voivat myös aiheuttaa käyttäjien menettämistä.

Push-ilmoitusten tärkeys markkinoijille korostuu tilastoista, jotka näyttävät, että noin 60 % mobiilikäyttäjistä ottaa ne käyttöön laitteillaan, ja 57 % kokee ne hyödyllisinä (Dogtiev 2023).

3 Teknologiavalinnat

3.1 Yleistä

Tässä luvussa käydään läpi projektissa käytettävät teknologiavalinnat. Teknologiavalinnat ovat tarkasti harkittuja, mutta osa niistä ovat suoria riippuvuuksia olemassa olevaan infrastruktuuriin Jyväskylän yliopiston digipalveluilla. Tarkat teknologiavalinnat ja niiden havainnollistaminen auttavat, kun suunnitellaan mitä tehdään ja millä.

3.2 Flutter

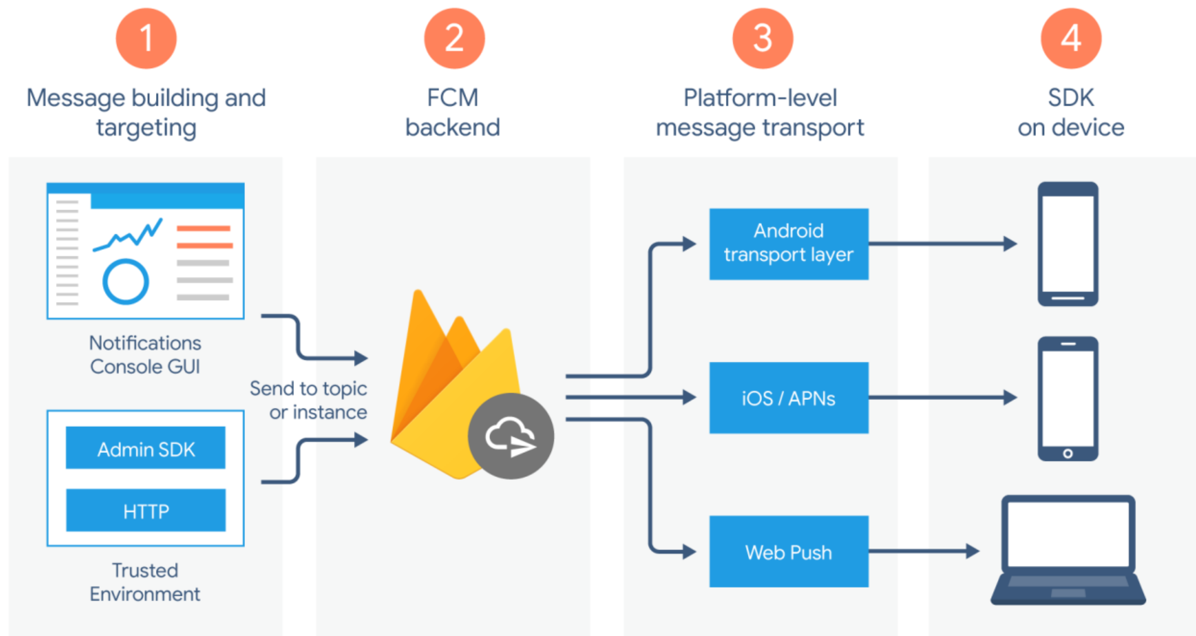
Flutter on Googlen kehittämä avoimen lähdekoodin käyttöliittymäkehitystyökalu sovelluskehitystä varten, jonka ensimmäinen vakaa julkaisu (versio 1.0) julkaistiin vuonna 2018 (Sneath 2018). Flutterilla on mahdollista kehittää sekä mobiilisovelluksia Android ja iOS laitteille että internetselaimille ja työpöytälaitteille yhdestä ja samasta koodikannasta (Flutterin dokumentaatio n.d.).

Googlen ylläpitämässä Dart ja Flutter kirjastorepositoryssä ”pub.dev”:ssä on lukuisia Flutter- ja Dart-yhteisön luomia kirjastoja, joita voi käyttää apunaan sovelluskehityksessä, muun muassa Firebasen omat kehitystyökalut Flutter-kehitykselle.

Koska MyJYU on alun perin toteutettu Flutterilla, on myös sen jatkokehittäminen mahdollista ainoastaan Flutterilla.

3.3 Firebase Cloud Messaging

Firebase Cloud Messaging (FCM) on Googlen tytäryhtiön, Firebasen kehittämä pilvipalvelu, joka tarjoaa ratkaisuja ilmoitusten lähettämiseen useille eri alustoille. Kuvio 2 kuvaa, miten ilmoitus kulkee eri komponenttien läpi aina päätelaitteeseen asti.



Kuvio 2. FCM:n arkkitehtuurillinen kuvaus (FCM Architectural Overview 2023).

1. Ilmoitus rakennetaan Firebasen omassa käyttöliittymässä tai itse kehitetyssä palvelussa, joka hyödyntää Firebasen tarjoamaa Admin SDK:ta. Admin SDK:n käyttäminen omassa palvelussa mahdollistaa viestien automatisoinnin ja räätälöimisen parhaiten omiin tarkoituksiinsa. Ilmoituksia on mahdollista lähettää ”topiceihin”, joita laitteet ovat tilanneet, sekä suoraan yksittäisiin laitteisiin laitteen oman FCM-tunnisteen avulla.
2. FCM backend vastaanottaa lähetetyt ilmoitukset, lähettää ne eteenpäin alustakohtaisille viestinjakajille.
3. Alustakohtaiset viestinjakajat lähettävät viestit päätelaitteille. Android laitteiden jakaja on Android transport layer (ATL) ja iOS laitteiden jakaja on Apple Push Notification service (APNs).
4. Laite vastaanottaa viestin ja käsittelee sen riippuen siitä, onko sovellus auki, taustalla vai pois päältä. (FCM Architectural Overview 2023).

3.4 Python

Python on Python Software Foundationin kehittämä avoimen lähdekoodin ohjelmointikieli. Python on monipuolinen ohjelmointikieli, jota voidaan hyödyntää tehokkaasti taustapalveluissa. Jyväskylän yliopiston digipalveluilla on laajalti kehitetty palveluita Pythonia käyttäen, joten jatkokehittämisen ja infrastruktuurin yhtenäisyyden kannalta on järkevää käyttää Pythonia taustapalveluiden tuottamisessa. Toisena vaihtoehtona olisi ollut Java, jota Ravoof (2023) kuvailee blogijulkaisussaan monimutkaisemmaksi ja puolestaan Pythonia hän pitää aloittelijaystävällisempänä. Näistä syistä valitsin Pythonin.

3.5 RabbitMQ

RabbitMQ on yksi käytetyimmistä viestijonojärjestelmistä (RabbitMQ verkkosivut 2023). Johansson (2019) kuvailee RabbitMQ:n toimintaa seuraavasti: viestit saapuvat jonoon yhdestä palvelusta ja toinen palvelu, joka kuuntelee viestijonoa, prosessoi viestit määrätyllä tavalla. Tämän työn näkökulmasta halutaan lähettää viestijonoon viestejä, jotka sisältävät tarvittavan tiedon push-ilmoituksen rakentamiseksi. Kuuntelijan rooli puolestaan on prosessoida viesti push-ilmoitukseksi ja lähettää se laitteeseen.

3.6 Docker

Docker on alusta, jolla voidaan muuntaa sovelluksia ”konteiksi”. Kontti on eristetty ympäristö, jossa sovellus voi pyöriä itsekseen, joka helpottaa sovelluksien integrointia muuhun infrastruktuuriin. (Docker overview n.d). JYU digipalveluilla hyödynnetään laajalti Dockeria koko sovellusinfrastruktuurissa, joten tämä on luonnollinen työkaluvalinta. Docker mahdollistaa nopean päivityksen ja julkaisun, sillä kontit ovat helppo rakentaa ja laittaa pyörimään palvelimelle.

3.7 PostgreSQL

PostgreSQL on avoimen lähdekoodin tietokantahallintajärjestelmä. Tämä valinta myös juurtaa syynsä siihen, että JYU digipalveluilla suurin osa käytössä olevista tietokannoista ovat tiotakantapalvelimilla pyöriviä PostgreSQL-tietokantoja. Projekti vaatii vain yksinkertaisen tietokannan ja kehityksen aikana PostgreSQL:n itse julkaisemaa Docker Imagea on helppo käyttää ja muovata omiin tarkoituksiin.

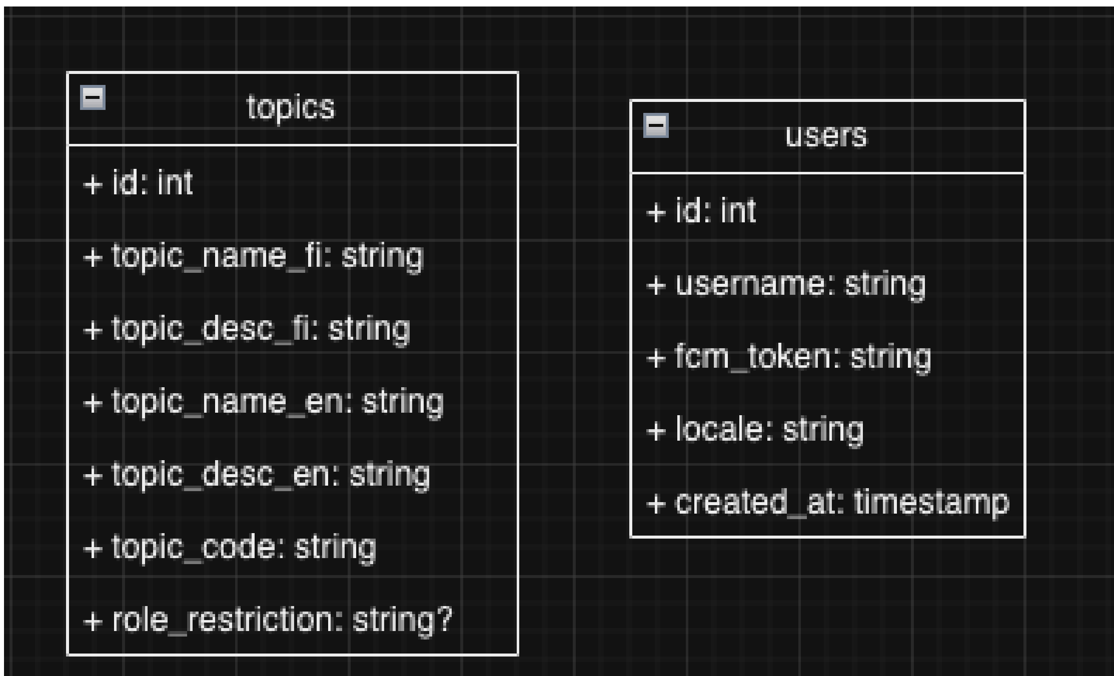
4 CASE: MyJYU-mobiilisovelluksen ilmoitukset

4.1 Suunnittelu

Tarkka suunnittelu helpottaa sovelluskehitysprojektin läpivientiä huomattavasti. Alakappaleissa on kuvattu projektin eri osien vaatimuksia yksityiskohtaisesti.

4.1.1 PostgreSQL-tietokanta

Luodaan aluksi yksinkertainen PostgreSQL tietokanta, joka sisältää kaksi taulua (ks. Kuvio 3). Taulu “topics” pitää kirjaa olemassa olevista ilmoitustopiceista ja vastaavasti “users” pitää kirjaa FCM-tokenien ja käyttäjien linkityksistä aikaleimoineen.



Kuvio 3. Tietokantakuvaus.

4.1.2 Laiterekisteröintipalvelu

Jotta voitaisiin lähettää yksittäisille käyttäjille kohdennettuja ilmoituksia, tarvitaan taustapalvelu, joka yhdistää käyttäjän mobiililaitteen Firebase Cloud Messaging tokenin (FCM-token) käyttäjän käyttäjänimeen. Samaan palveluun tullaan yhdistämään myös ilmoitustopicejen hallinta ja jakelu. Seuraavaksi luodaan Rest-API rajapinta hyödyntäen Pythonin FastAPI-kirjastoa. Tässä rajapinnassa tulee olemaan kaksi endpointia (rajapintaosoitetta), POST-endpoint ”/register-fcm-token”, joka yhdistää laitteen FCM-tokenin käyttäjänimeen, ja GET-endpoint ”/get-topics”, joka palauttaa listan kaikista ilmoitustopiceista.

POST-endpoint ”/register-fcm-token” vastaanottaa MyJYU:sta käyttäjän oAuth access tokenin, laitteen FCM-tokenin ja aikaleiman sisältävän JSONin. Access tokenin avulla haetaan Jyväskylän yliopiston identiteettipalvelusta käyttäjän käyttäjänimi ja kielikoodi, jonka jälkeen käyttäjänimi, FCM-token, kielikoodi ja aikaleima tallenetaan tietokannan ”Users”-tauluun. Käyttäjänimi ja kielikoodi olisivat myös mahdollista lähettää suoraan MyJYU:sta, mutta koska kyseessä on julkinen rajapinta, niin access tokenin käyttö takaa sen, että tietokantaan ei ole mahdollista syöttää kuin käyttäjän oikeaa dataa. Kielikoodia hyödynnetään oikean kielisen ilmoituksen lähettämisessä (englanniksi tai suomeksi).

GET-endpoint ”/get-topics” palauttaa tietokannan ”Topics”-taulusta JSON-listan kaikista olemassa olevista ilmoitustopiceista. Topic sisältää topicin nimen ja kuvauksen suomeksi ja englanniksi, sekä myös topicin koodin. Topiceiden hallinta, kuten poistaminen ja uusien luominen on jätetty pois rajapinnasta sen julkisuuden takia, joten muutokset täytyy tehdä suoraan tietokantaan.

4.1.3 Ilmoitusten lähettelijä

Suunnitellaan Firebase Admin SDK:ta hyödyntävä mikropalvelu, joka hallitsee ilmoitusten lähettämisen topiceihin ja käyttäjien laitteisiin. Ratkaisuna luodaan Pythonilla ohjelma, joka kuuntelee RabbitMQ-viestijonoa. Viestijonoon on mahdollista lähettää viestejä, jotka sisältävät JSONin, jossa on joko yksittäiselle käyttäjälle tarkoitettu viesti tai topiciin tarkoitettu viesti. Kuuntelija käsittelee viestin sisältäneen JSONin ja ohjaa sen joko topiciin, joka vastaanotetaan jokaisessa laitteessa, joka on tilannut kyseisen topicin, tai mikäli viesti on yksittäiselle käyttäjälle, ohjataan se käyttäjän laitteisiin.

4.1.4 MyJYU frontend

MyJYU:n koodi vaatii monia muutoksia, jotta sovellus pystyisi vastaanottamaan ilmoituksia. Projektiin on lisättävä uusia kirjastoriippuvuuksia, kuten firebase_core, jolla voidaan toteuttaa ilmoitusten vastaanottamisominaisuus. Tämän lisäksi on ohjelmoitava miten sovellus reagoi, kun käyttäjä klikkaa ilmoitusta. Huomioon on otettava tapaukset, jolloin sovellus on auki ja aktiivisena, auki taustalla tai poissa päältä. Lisäksi sovelluksen kirjautumisprosessiin on lisättävä rajapintakutsut FCM-tokenin rekisteröimiselle, sekä topicien hakuun.

4.2 Toteutus

4.2.1 Yleistä

Tässä luvussa pureudutaan tekniseen toteutukseen yksityiskohtaisesti. Toteutuksia on tarkoitus tarkastella siltä näkökannalta, että saadaan ymmärrys, miten asia tehdään, ja miksi se tehdään juuri niin.

4.2.2 Laiterekisteröintipalvelu

Luodaan uusi Python-projekti omalla virtuaaliympäristöllä hyödyntämällä ”venv”-pakettia. Projektin oma virtuaaliympäristö pitää projektissa tarvittavat riippuvuudet vain siinä ympäristössä. Aloitetaan ohjelman kirjoitus riippuvuuksien tuonnilla kuvion 4 mukaisesti.

```
1   from fastapi import FastAPI, HTTPException
2   import psycopg2
3   import logging
4   import requests
5   import os
6   from schemas import FCMTokenRegistrationRequest, TopicResponse
7
8
9   logging.basicConfig(level=logging.INFO)
10
11  app = FastAPI()
```

Kuvio 4. Ohjelman luonti ja riippuvuudet

Rivit 1–6 ovat tarvittavien riippuvuuksien tuonteja projektiin. Tämän lisäksi riippuvuudet täytyy myös asentaa virtuaaliympäristöön, jotta niitä voi käyttää. Rivillä 9 asetetaan loggauksen taso INFO-tasolle, joka tarkoitta, että ohjelma näyttää kaiken loggauksen INFO-tasosta lähtien. Syy tälle muutokselle on, että oletuksena loggauksen perustaso on WARNING. Rivillä 11 luodaan FastAPI-olio. Seuraavaksi on haettava ympäristömuuttujat sovelluksen koodiin kuvion 5 esittämän koodikatkelman mukaisesti.

```
13  DB_HOST = os.environ.get("DB_HOST")
14  DB_PORT = os.environ.get("DB_PORT")
15  DB_NAME = os.environ.get("DB_NAME")
16  DB_USER = os.environ.get("DB_USER")
17  DB_PASSWORD = os.environ.get("DB_PASSWORD")
18  USER_INFO_URL = os.environ.get("USER_INFO_URL")
19
20  def get_db_connection():
21      return psycopg2.connect(
22          host=DB_HOST, port=DB_PORT, dbname=DB_NAME, user=DB_USER, password=DB_PASSWORD
23      )
```

Kuvio 5. Tietokantayhteyden muodostus.

Rivit 13–18 asettavat ympäristömuuttujat tietokantayhteyttä varten. Ympäristömuuttujat ovat ohjelman ulkoa päin asetettavia muuttujia. Koska tietokantaparametrit sisältävät arkaluonteista tietoa, kuten salasanoja, on järkevää tehdä näistä ympäristömuuttujia, jolloin näitä parametreja ei vahingossa pusketa versionhallintaan. User_info_url on myös ympäristömuuttuja siltä varalta, että se voi joskus muuttua, jolloin voidaan vain muuttaa ympäristömuuttuja, eikä tarvitse tehdä uutta julkaisua palvelusta. Rivit 20–23 ottavat yhteyden PostgreSQL tietokantaan hyödyntäen psycopg2-kirjastoa ja asetettuja ympäristömuuttujia.

Seuraavaksi luodaan skeemat rajapintaosoitteille kuvion 6 esittämällä tavalla.

```python
from pydantic import BaseModel
from typing import Optional


class FCMTokenRegistrationRequest(BaseModel):
    access_token: str
    fcm_token: str
    created_at: str

class TopicResponse(BaseModel):
    id: int
    topic_name_fi: str
    topic_desc_fi: str
    topic_name_en: str
    topic_desc_en: str
    topic_code: str
    role_restriction: Optional[str] = None

```

Kuvio 6. Schemas.py tiedosto.

Rivillä 5–8 oleva FCMTokenRegistrationRequest-luokkaa hyödynnetään käyttäjän FCM-tokenin rekisteröinnissä ja rivien 10–17 TopicResponse-luokkaa hyödynnetään topicejen haussa. Nämä luokat mahdollistavat sen, että rajapintaosoitteilla on jonkinlainen oletus, minkälaista tietoa ne palauttavat tai vastaanottavat.

Toteutusta jatketaan luomalla laiterekisteröinnin rajapintaosoite ”/register-fcm-token” (ks. Kuvio 7).

```python
26  @app.post("/register-fcm-token")
27  async def register_fcm_token(data: FCMTokenRegistrationRequest):
28      conn = None
29      cursor = None
30      try:
31          conn = get_db_connection()
32          cursor = conn.cursor()
33
34          access_token = data.access_token
35          fcm_token = data.fcm_token
36          created_at = data.created_at
37
38          response_json = get_user_info(access_token)
39
40          if "preferred_username" in response_json and "locale" in response_json:
41              username = response_json.get("preferred_username")
42              locale = response_json.get("locale")
43
44              logging.info(f"Username: {username}")
45              logging.info(f"Locale: {locale}")
46          else:
47              handle_missing_keys(response_json)
48
49          # Check if the fcm_token already exists for any user
50          cursor.execute("SELECT username FROM users WHERE fcm_token = %s", (fcm_token,))
51          existing_user = cursor.fetchone()
52
53          if existing_user:
54              existing_username = existing_user[0]
55              return update_existing_user(cursor, username, locale, created_at, fcm_token, existing_username)
56          else:
57              return insert_new_user(cursor, username, fcm_token, created_at, locale)
58
59      except Exception as e:
60          if conn:
61              conn.rollback()
62          raise HTTPException(status_code=500, detail=str(e))
63      finally:
64          if cursor:
65              cursor.close()
66          if conn:
67              conn.close()
```

Kuvio 7. ”register-fcm-token”-rajapintaosoite.

Kuvion 7 metodi ”register_fcm_token” rekisteröi käyttäjän laitteen FCM-tokenin käyttäjän käyttäjänimeen. Metodi ottaa argumenttina FCMTokenRegistrationRequest-luokan. Metodi ottaa yhteyden tietokantaan ja asettaa muuttujat saamansa datan perusteella. Rivillä 38 kutsutaan metodia ”get_user_info”, jota tarkastellaan tarkemmin kuviossa 8.

```python
69  def get_user_info(access_token: str):
70      userinfo_url = USER_INFO_URL
71      headers = {"Authorization": f"Bearer {access_token}"}
72      response = requests.get(userinfo_url, headers=headers)
73      logging.info(f"Response from userinfo: {str(response.json())}")
74      return response.json()
```

Kuvio 8. ”get_user_info”-metodi.

Kuvion 8 metodissa kutsutaan Jyväskylän yliopiston käyttäjätietorajapintaa MyJYU:sta saadulla access tokenilla. Metodi palauttaa käyttäjän tiedot JSON-tiedostona.

Kuvion 7 rivillä 40–47 asetetaan muuttujat käyttäjänimelle ja kielikoodille käyttäjätietorajapinnasta saadun datan perusteella. Mikäli saatu JSON ei sisällä kenttiä ”preferred_username” tai ”locale”, kutsutaan metodia ”handle_missing_keys” (ks. Kuvio 9). Metodi palauttaa virheen puuttuvien kenttien perusteella.

```python
76  def handle_missing_keys(response_json):
77      if "preferred_username" not in response_json:
78          logging.error("Username is missing in the JSON response")
79      if "locale" not in response_json:
80          logging.error("Locale is missing in the JSON response")
```

Kuvio 9. ”handle_missing_keys” -metodi.

Kuvion 7 riveillä 49–57 tarkistetaan tietokannasta, että onko jollekin käyttäjälle rekisteröity FCM-token, jota yritetään rekisteröidä. Jos tulos löytyy, kutsutaan metodia ”update_existing_user” (ks. Kuvio 10), muussa tapauksessa kutsutaan metodia ”insert_new_user” (ks. Kuvio 11).

```
82  def update_existing_user(cursor, username, locale, created_at, fcm_token, existing_username):
83      if existing_username == username:
84          cursor.execute(
85              "UPDATE users SET created_at = %s, locale = %s WHERE fcm_token = %s",
86              (created_at, locale, fcm_token),
87          )
88          return {"message": f"This token is already registered for user {username}, updated at: {created_at}"}
89      else:
90          cursor.execute(
91              "UPDATE users SET username = %s, locale = %s, created_at = %s WHERE fcm_token = %s",
92              (username, locale, created_at, fcm_token),
93          )
94          return {
95              "message": f"This token was previously registered for user {existing_username}, "
96                         f"token is now registered for user {username}, updated at: {created_at}"
97          }
```

Kuvio 10. ”update_existing_user” -metodi.

Kuvion 10 riveillä 83–88 päivitetään olemassa olevan FCM-tokenin rekisteröinnin aikaleima, mikäli käyttäjänimi on sama, mikä on tietokannassa. Riveillä 90–97 puolestaan käsitellään tapaus, jossa rekisteröitävä FCM-token on jo rekisteröity toiselle käyttäjälle. Tämä tapaus voi ilmetä, jos käyttäjä antaa oman laitteensa toiselle käyttäjälle. Tapaus on hyvin epätodennäköinen, mutta on hyvä varautua jokaiseen tilanteeseen. Ohjelma päivittää käyttäjänimen ja aikaleiman FCM-tokenin rekisteröintiin.

```
def insert_new_user(cursor, username, fcm_token, created_at, locale):
    cursor.execute(
        "INSERT INTO users (username, fcm_token, created_at, locale) VALUES (%s, %s, %s, %s)",
        (username, fcm_token, created_at, locale),
    )
    return {"message": f"FCM token registered successfully for user {username} at {created_at}, locale: {locale}"}
```

Kuvio 11. ”insert_new_user” -metodi.

Kuviossa 11 syötetään uusi rivi tietokantaan kaikkine tietoineen. Kuvion 7 riveillä 59–67 viimeistellään ohjelman suoritus. Mikäli rajapinta kohtaa virheen kesken suorituksen käytetään ”conn-rollback”-metodia muutosten palauttamiseen. Lopulta rivin 63 alkavassa ”finally”-blokissa suljetaan tietokantayhteys.

Seuraavaksi luodaan ”get-topics”-rajapintaosoite (ks. Kuvio 12).

```
107    # Get topics
108    @app.get("/get-topics")
109    def get_topics():
110        conn = None
111        cursor = None
112        try:
113            conn = get_db_connection()
114            cursor = conn.cursor()
115
116            # Query the topics from the database
117            cursor.execute(
118                "SELECT * FROM topics"
119            )
120
121            topics = cursor.fetchall()
122
123            # Convert the database results into TopicResponse instances
124            topic_responses = [
125                TopicResponse(
126                    id=topic[0],
127                    topic_name_fi=topic[1],
128                    topic_desc_fi=topic[2],
129                    topic_name_en=topic[3],
130                    topic_desc_en=topic[4],
131                    topic_code=topic[5],
132                    role_restriction=topic[6],
133                )
134                for topic in topics
135            ]
136
137            return topic_responses
138
139        except Exception as e:
140            raise HTTPException(status_code=500, detail=str(e))
141        finally:
142            if cursor:
143                cursor.close()
144            if conn:
145                conn.close()
```

Kuvio 12. ”get-topic”-rajapintaosoite.

Kuvion 12 metodi palauttaa tietokannasta kaikki olemassa olevat topicit kaikkine tietoineen. Riveillä 113–121 otetaan yhteys tietokantaan ja suoritetaan SQL-kysely, joka hakee taulusta ”topics” kaiken datan listaan. Tulokset muunnetaan for-loopilla TopicResponse-olioiksi ja rajapinta palauttaa topicit JSON-listana. Lopulta riveillä 139–145 palautetaan virhe, mikäli sellainen kohdataan ja sitten suljetaan tietokantayhteys.

Tämän jälkeen luodaan requirements.txt tiedosto komennolla ”pip freeze > requirements.txt”. Tämä komento luo tekstitiedoston, jossa on jokainen projektin riippuvuus versionumeroineen. Tätä tiedostoa hyödynnetään Dockerfilen luonnissa (ks. Kuvio 13).

```
1  FROM python:3.11.3-slim
2
3  ADD app.py schemas.py requirements.txt /app/
4  WORKDIR /app
5
6  RUN python3 -m venv venv && \
7      venv/bin/pip install -r requirements.txt
8
9  CMD ["venv/bin/uvicorn", "app:app", "--host", "0.0.0.0", "--port", "8000"]
10
11 EXPOSE 8000
```

Kuvio 13. Dockerfile

Dockerfile sisältää kaiken tarvittavan tiedon, jolla voidaan luoda projektista Docker-kontti, joka puolestaan voidaan laittaa pyörimään verkkopalvelimelle.

4.2.3 Ilmoitusten lähettelijä

Tässäkin palvelussa laiterekisteröintirajapinnan tapaan luodaan uusi Python projekti, virtuaaliympäristö ja asennetaan tarvittavat riippuvuudet. Riippuvuuksien tuonnin ja ympäristömuuttujien asettamisen jälkeen kutsutaan kuvion 14 mukaista metodia.

```
19    cred = credentials.Certificate("my-jyu-firebase-adminsdk-key.json")
20    firebase_admin.initialize_app(cred)
```

Kuvio 14. Firebase Admin SDK:n initialisointi

Rivillä 20 kutsutaan firebase_admin-kirjaston initialize_app metodia, joka mahdollistaa muiden Firebase-komentojen käyttämisen myöhemmin. Rivin 19 cred-muuttuja sisältää tiedot ja salasanat käytettävästä Firebase-projektista.

Seuraavaksi luodaan skeemat topiciin lähtevästä viestistä ja yksittäiselle käyttäjälle lähtevästä viestistä (ks. Kuvio 15).

```python
from pydantic import BaseModel, constr


class SendNotificationToUserRequest(BaseModel):
    username: constr(min_length=1, strip_whitespace=True)
    notification_title_fi: constr(min_length=1, strip_whitespace=True)
    notification_body_fi: constr(min_length=1, strip_whitespace=True)
    notification_title_en: constr(min_length=1, strip_whitespace=True)
    notification_body_en: constr(min_length=1, strip_whitespace=True)
    route: constr(min_length=1, strip_whitespace=True)

    class Config:
        extra = "forbid"

"""
{
    "username": "example_username",
    "notification_title_fi": "example_title_fi",
    "notification_body_fi": "example_body_fi",
    "notification_title_en": "example_title_en",
    "notification_body_en": "example_body_en",
    "route": "example_route"
}
"""

class SendNotificationToTopicRequest(BaseModel):
    topic: constr(min_length=1, strip_whitespace=True)
    notification_title_fi: constr(min_length=1, strip_whitespace=True)
    notification_body_fi: constr(min_length=1, strip_whitespace=True)
    notification_title_en: constr(min_length=1, strip_whitespace=True)
    notification_body_en: constr(min_length=1, strip_whitespace=True)

    class Config:
        extra = "forbid"

"""
{
    "topic": "example_topic",
    "notification_title_fi": "example_title_fi",
    "notification_body_fi": "example_body_fi",
    "notification_title_en": "example_title_en",
    "notification_body_en": "example_body_en"
}
"""
```

Kuvio 15. Schemas.py -tiedosto.

Kuvion 15 rivit 4–13 sisältävät skeeman yksittäiselle käyttäjälle lähtevästä viestistä. Parametrien tyyppinä käytetään constr. Tälle tyypille voi asettaa rajoituksia, mitä merkkijonon on noudatettava. Tässä tapauksessa rajoitukset ovat, että merkkijonon pituus on oltava vähintään yksi, ja lisäksi merkkijonosta poistetaan lopussa olevat ylimääräiset välilyönnit. Rivin 12–13 lisäys varmistaa, että mikäli muunnettava JSON sisältää ylimääräisiä kenttiä, muuntaminen pysäytetään ja palautetaan validaatiovirhe. Topiciin lähtevän viestin skeema noudattaa samoja sääntöjä. Lisäksi tiedostossa on kommentoitu esimerkkiformaatti muunnettavasta JSONista.

Tässä projektissa käytetään samalla tavalla psycopg2-kirjastoa tietokantayhteyden muodostamiseen, mutta mukana on myös aikakatkaisumetodi, joka yrittää yhteyttä uudelleen muutaman kerran, jonka jälkeen se palauttaa virheen (ks. Kuvio 16).

```
36  def get_db_connection():
37      try:
38          return psycopg2.connect(
39              host=DB_HOST, port=DB_PORT, dbname=DB_NAME, user=DB_USER, password=DB_PASSWORD
40          )
41      except psycopg2.OperationalError as e:
42          raise ConnectionError(f"Could not connect to the database: {e}")
43
44  MAX_RETRIES = 4
45  RETRY_DELAY = 5
46
47  def get_db_connection_with_retry():
48      retry_count = 0
49
50      while retry_count < MAX_RETRIES:
51          try:
52              return get_db_connection()
53          except ConnectionError as e:
54              logging.error(f"Database connection error: {e}")
55
56              retry_count += 1
57
58              delay = RETRY_DELAY * 2 ** retry_count
59
60              logging.info(f"Retrying database connection in {delay} seconds...")
61              time.sleep(delay)
62
63      logging.error("Max retries reached. Could not establish a database connection.")
64      raise ConnectionError("Max retries reached for database connection")
65
```

Kuvio 16. Tietokantayhteys-metodi aikakatkaisulla.

Tämä aikakatkaisu on kehitetty sitä varten, jos tietokanta on alhaalla. Uudelleenyritysten jälkeen luettu viesti palaa takaisin viestijonoon, jolloin viesti ei mene hukkaan.

RabbitMQ-viestijonoyhteys otetaan kuvion 17 mukaisilla koodiriveillä.

```
230    # Create a connection to RabbitMQ
231    credentials = pika.PlainCredentials(RABBITMQ_USER, RABBITMQ_PASSWORD)
232    parameters = pika.ConnectionParameters(host=RABBITMQ_HOST, virtual_host=RABBITMQ_VIRTUAL_HOST, credentials=credentials)
233    connection = pika.BlockingConnection(parameters)
234    channel = connection.channel()
235
236    # Declare the queue (create if it doesn't exist)
237    channel.queue_declare(queue=QUEUE_NAME, durable=True)
238
239    # Set up the callback to handle incoming messages
240    channel.basic_consume(queue=QUEUE_NAME, on_message_callback=callback)
241
242    # Start listening for messages
243    channel.start_consuming()
244
```

Kuvio 17. RabbitMQ-yhteys.

Kuvion 17 riveillä 231–234 muodostetaan yhteys RabbitMQ-viestijonoon hyödyntäen ”pika”-kirjastoa ja projektin ympäristömuuttujia. Rivillä 237 määritetään, mitä jonoa kuunnellaan ja luodaan se, jos sitä ei ole olemassa hyödyntäen ”durable=True” asetusta. Rivi 240 määrittää, mitä metodia kutsutaan uuden viestin saapuessa ja tässä tapauksessa se on ”callback”-metodi. Rivi 243 käynnistää kuuntelijan.

Jotta kuuntelija toimisi, täytyy luoda sille callback-metodi (ks. Kuvio 18).

```
67  def callback(ch, method, properties, body):
68      try:
69          # Decode the JSON message
70          message = json.loads(body)
71
72          # Determine the type of the message and validate against the appropriate model
73          if 'username' in message:
74              parsed_message = SendNotificationToUserRequest(**message)
75              logging.info(f"sending message to user {parsed_message.username}")
76              send_notification_to_user(parsed_message, ch, method)
77          elif 'topic' in message:
78              parsed_message = SendNotificationToTopicRequest(**message)
79              logging.info(f"sending message to topic {parsed_message.topic}")
80              send_notification_to_topic(parsed_message)
81              ch.basic_ack(delivery_tag=method.delivery_tag)
82          else:
83              logging.info("Received JSON message does not match any expected format.")
84              ch.basic_nack(delivery_tag=method.delivery_tag, requeue=False)
85              return
86
87      except json.JSONDecodeError as e:
88          logging.info("Error decoding JSON:", e)
89          ch.basic_nack(delivery_tag=method.delivery_tag, requeue=False)
90      except ValidationError as e:
91          logging.info("Validation error:", e)
92          ch.basic_nack(delivery_tag=method.delivery_tag, requeue=False)
93      except ConnectionError as e:
94          logging.error("Database connection error:", e)
95          ch.basic_nack(delivery_tag=method.delivery_tag, requeue=True)
```

Kuvio 18. Callback-metodi.

Kuten jo mainittu, callback-metodia kutsutaan joka kerta, kun kuuntelija havaitsee uuden viestin viestijonossa. Rivillä 70 metodi yrittää dekoodata JSONin viestistä. Jos viestin sisältö ei ole JSON, siirrytään riveille 87-89, joissa palautetaan virhe ”JSONDecodeError” ja hylätään viesti pois jonosta ilman, että sitä laitetaan takaisin jonoon ”requeue=False” parametrin avulla. Jos taas viesti on JSON, seuraavaksi siirrytään tarkastamaan, onko JSONin muoto oikea. Rivillä 73 tarkistetaan, että sisältääkö JSON ”username”-kentän, jos sisältää, niin oletetaan että viesti on lähtemässä käyttäjälle, joten parsitaan JSONin kentät SendNotificationToUserRequest-olioon. Parsiminen onnistuu vain, jos parsittava JSON sisältää juuri ne kentät mitä SendNotificationToUser-luokassa on määritetty rajoituksineen. Eli tapauksissa, joissa jotkin kentät ovat tyhjiä tai on ylimääräisiä kenttiä, heitetään rivin 90–92 ValidationError ja hylätään viesti. Kun JSONin parsiminen onnistuu, siirrytään send_notification_to_user-metodiin (ks. kuvio 19). Samalla tavalla, jos viestin JSON sisältää kentän

”topic”, parsitaan se omaan olioonsa ja siirrytään metodiin send_notification_to_topic, jota tarkastellaan myöhemmin. Tapauksessa, jossa JSON sisältää ”username” ja ”topic” kentät, palautetaan ValidationError, syynä ylimääräiset kentät.

```
 98  def send_notification_to_user(data: SendNotificationToUserRequest, ch, method):
 99      conn = None
100      cursor = None
101
102
103      conn = get_db_connection_with_retry()
104
105      if conn is not None:
106          try:
107              cursor = conn.cursor()
108              username = data.username
109
110              logging.info(f"Sending notification to user {username}")
111              user_info = get_fmc_tokens_and_locale_for_user(cursor, username)
112
113              if user_info:
114                  fcm_tokens, locales = zip(*user_info)
115
116                  valid_tokens = send_notification_and_get_valid_tokens(
117                      fcm_tokens, locales, data, cursor, conn
118                  )
119
120                  if valid_tokens:
121                      logging.info(f"Sent notification to {len(valid_tokens)} valid devices")
122                      ch.basic_ack(delivery_tag=method.delivery_tag)
123                  else:
124                      logging.error("No valid tokens found for user")
125                      ch.basic_nack(delivery_tag=method.delivery_tag, requeue=False)
126              else:
127                  logging.error(f"User not found: {username}")
128                  ch.basic_nack(delivery_tag=method.delivery_tag, requeue=False)
129
130
131          finally:
132              if cursor is not None:
133                  cursor.close()
134              if conn is not None:
135                  conn.close()
136
```

Kuvio 19. send_notification_to_user -metodi.

Tämän metodi lähettää viestin yksittäisen käyttäjän laitteisiin. Rivillä 103 otetaan yhteys tietokantaan ja rivillä 111 kutsutaan get_fcm_tokens_and_locale_for_user -metodia (ks. Kuvio 20).

```
139 def get_fmc_tokens_and_locale_for_user(cursor, username):
140     cursor.execute("SELECT fcm_token, locale FROM users WHERE username = %s", (username,))
141     return cursor.fetchall()
```

Kuvio 20. get_fcm_tokens_and_locale_for_user -metodi.

Tämä metodi hakee käyttäjänimen perusteella tietokannasta FCM-tokenit ja locale-koodin. Jos metodi onnistuu palauttamaan tietoja, kutsutaan kuvion 19 rivillä 116 “send_notification_and_get_valid_tokens” -metodia (ks. Kuvio 21).

```
143 def send_notification_and_get_valid_tokens(tokens, locales, data, cursor, conn):
144     valid_tokens = []
145 
146     for token, locale in zip(tokens, locales):
147         try:
148             notification_title, notification_body = get_localized_title_and_body(
149                 locale, data.notification_title_fi, data.notification_body_fi,
150                 data.notification_title_en, data.notification_body_en
151             )
152 
153             if notification_title is not None and notification_body is not None:
154                 construct_and_send_notification_to_token(token, data.route, notification_title, notification_body)
155                 valid_tokens.append(token)
156             else:
157                 logging.warning("Notification title or body is missing.")
158         except UnregisteredError:
159             handle_unregistered_token(token, cursor, conn)
160     return valid_tokens
```

Kuvio 21. send_notification_and_get_valid_tokens -metodi.

Tämä metodi kutsuu muita metodeja saadakseen talteen kaikki FCM-tokenit, joihin viestit lähtivät onnistuneesti. Hyödynnetään for-looppia viestien lähettämiseen. Riveillä 148–151 kutsutaan get_localized_title_and_body -metodia, joka valitsee käyttäjän locale-koodin perusteella, minkä kielinen ilmoitus lähetetään (ks. Kuvio 22).

```
162 def get_localized_title_and_body(locale, title_fi, body_fi, title_en, body_en):
163     if locale == "fi":
164         return title_fi, body_fi
165     elif locale == "en" or locale is None:
166         return title_en, body_en
```

Kuvio 22. get_localized_title_and_body -metodi.

Tämä metodi palauttaa suomenkieliset tiedot, jos käyttäjän locale-koodi on ”fi” tai englanninkieliset, jos locale-koodi on ”en” tai None, eli null. Locale-koodin ei koskaan pitäisi olla null, mutta on hyvä varautua tähänkin tilanteeseen. Ilmoituksen kielityksen jälkeen kutsutaan metodia ”construct_and_send_notification_to_token” (ks. Kuvio 23).

```
168 def construct_and_send_notification_to_token(token, route, title, body):
169     message = messaging.Message(
170         notification=messaging.Notification(
171             title=title,
172             body=body
173         ),
174         token=token,
175         data={
176             "route": route,
177         }
178     )
179     messaging.send(message)
180     logging.info(f"Sending message to token: {token}")
```

Kuvio 23. construct_and_send_notification_to_token -metodi.

Riveillä 169–178 muodostetaan lähetettävä ilmoitus Firebase Admin SDK -paketin avulla. Onnistuneet lähetykset tallentavat tokenin ”valid_tokens” listaan (ks. Kuvio 21, rivi 155). Mikäli ilmoituksen lähettäminen ei onnistu vanhentuneen tokenin takia, palautetaan UnregisteredError ja kutsutaan metodia ”handle_unregistered_method”, joka poistaa vanhentuneen FCM-tokenin tietokannasta (ks. Kuvio 24).

```
218 def handle_unregistered_token(token, cursor, conn):
219     logging.warning(f"Token {token} is no longer valid, deleting from database")
220     cursor.execute("DELETE FROM users WHERE fcm_token = %s", (token,))
221     conn.commit()
```

Kuvio 24. handle_unregistered_token -metodi.

Viestin lähetys topiciin on hyvin samanlainen, mutta tietokantaoperaatioita ei tarvita (ks. Kuvio 25).

```python
182 def send_notification_to_topic(data: SendNotificationToTopicRequest):
183     try:
184         topic_fi, topic_en = construct_topic_names(data.topic)
185
186         message_fi = construct_topic_message(
187             data.notification_title_fi, data.notification_body_fi, topic_fi, data.route
188         )
189
190         message_en = construct_topic_message(
191             data.notification_title_en, data.notification_body_en, topic_en, data.route
192         )
193
194         response_fi = messaging.send(message_fi)
195         response_en = messaging.send(message_en)
196
197         log_notification_responses(data.topic, response_fi, response_en)
198     except Exception as e:
199         logging.info("Error sending notification to topic:", e)
200
201 def construct_topic_names(topic):
202     topic_fi = f"{topic}_fi"
203     topic_en = f"{topic}_en"
204     return topic_fi, topic_en
205
206 def construct_topic_message(title, body, topic, route):
207     return messaging.Message(
208         notification=messaging.Notification(
209             title=title,
210             body=body
211         ),
212         topic=topic,
213         data={
214             "route": route,
215         }
216     )
```

Kuvio 25. send_notification_to_topic -metodi.

Aluksi muodostetaan erikseen topicit suomenkielisille ja englanninkielisille ilmoituksille ”construct_topic_names” -metodilla. Sen jälkeen muodostetaan ilmoitukset ”construct_topic_message” -metodilla ja lähetetään ne. Lähettämisen jälkeen logataan lähetettyjen viestien ID:t ”log_notification_responses” -metodilla (ks. Kuvio 26).

```
224  def log_notification_responses(topic, response_fi, response_en):
225      logging.info(f"Sent notification to topic {topic}")
226      logging.info(f"Response for Finnish version: {response_fi}")
227      logging.info(f"Response for English version: {response_en}")
```

Kuvio 26. log_notification_responses -metodi.

Tästä palvelusta luodaan myös Docker kontti, joka laitetaan pyörimään JYU:n palvelimille.

4.2.4 MyJYU frontend

Taustapalveluiden jälkeen täytyy lisätä tuki MyJYU-sovellukseen ilmoitusten vastaanottamiseen. Aloitetaan Firebasen lisääminen hyödyntäen Googlen omia ohjeita, joiden mukaan ensin asennetaan Firebase CLI, komentorivityökalu Firebasea varten. Tämän jälkeen asennetaan FlutterFire CLI, komennolla ”dart pub global activate flutterfire_cli”. FlutterFire CLI mahdollistaa Firebasen lisäämisen Flutter-projektiin helposti ja automatisoidusti. Komento luo ”firebase_options.dart” tiedoston projektin ”lib/” -kansioon. (Add Firebase to your Flutter app 2023).

Seuraavaksi lisätään tarvittavat riippuvuudet projektin ”pubspec.yaml” -tiedostoon, firebase_core, firebase_messaging ja flutter_local_notifications. Seuraavaksi on lisättävä Firebasen initialisointi projektin main.dart -tiedostoon (ks. Kuvio 27).

```
30  await Firebase.initializeApp(
31    options: DefaultFirebaseOptions.currentPlatform,
32  );
```

Kuvio 27. Firebasen initialisointi.

Kuvion 27 koodi on välttämätön osa Firebasea, joka käynnistää Firebase instanssin sovelluksessa. Ilman tätä kuvion 28 lisäykset eivät tekisi yhtään mitään.

```
34  await FirebaseMessaging.instance.setForegroundNotificationPresentationOptions
35    alert: true, // Required to display a heads up notification
36    badge: true,
37    sound: true,
38  );
39
40  FirebaseMessaging.onMessage.listen((RemoteMessage message) {
41    const AndroidNotificationDetails androidPlatformChannelSpecifics =
42        AndroidNotificationDetails(
43      'high_importance_channel',
44      'High Importance Notifications',
45      importance: Importance.high,
46      priority: Priority.high,
47    );
48    const NotificationDetails platformChannelSpecifics =
49        NotificationDetails(android: androidPlatformChannelSpecifics);
50
51    final AndroidNotification? android = message.notification?.android;
52
53    if (message.notification != null && android != null) {
54      NotificationService().flutterLocalNotificationsPlugin.show(
55            message.messageId.hashCode,
56            message.notification?.title ?? '',
57            message.notification?.body ?? '',
58            platformChannelSpecifics,
59            payload: message.data['route'] ?? '/',
60          );
61    }
62  });
63  FirebaseMessaging.onBackgroundMessage(_firebaseMessagingBackgroundHandler);
64
65  await NotificationService().initNotifications();
```

Kuvio 28. Firebase-asetuksia.

Kuvion 28 riveillä 34–62 määritetään asetuksia ilmoituksiin, jotka saapuvat sovelluksen ollessa aktiivisena. Rivillä 53–60 on määritetty, mitä tietoa ilmoituksesta näytetään ”flutter_local_notifications”-kirjaston avulla. Rivi 63 kutsuu ”firebase_messaging”-kirjaston ”onBackGroundMessage”-metodia käyttäen viestiä, joka saapuu sovelluksen ollessa pois päältä tai taustalla. Viestin sisältö saadaan metodilla ”_firebaseMessagingBackgroundHandler”, joka kuuntelee viestejä myös sovelluksen ollessa pois päältä.

Alustusten jälkeen on toteutettava rajapintakutsut laiterekisteröintitaustapalveluun (ks. Kuvio 29).

```
import 'dart:convert';
import 'package:dio/dio.dart';
import 'package:flutter/cupertino.dart';
import 'package:myjyu/config/flavor_config.dart';

class SendFCMToken {
  Future<void> sendFCMToken(String accessToken, String fcmToken) async {
    final Dio dio = Dio();
    final String url = FlavorConfig.instance.values['fcm_token_registration'];
    final String createdAt = DateTime.now().toUtc().toIso8601String();

    final Map<String, dynamic> data = <String, dynamic>{
      'access_token': accessToken,
      'fcm_token': fcmToken,
      'created_at': createdAt,
    };

    try {
      final Response<dynamic> response = await dio.post(
        url,
        data: jsonEncode(data),
        options: Options(
          headers: <String, dynamic>{
            'Content-Type': 'application/json',
          },
        ),
      );
      if (response.statusCode != 200) {
        throw Exception('Failed to send Firebase token');
      }
    } catch (e) {
      debugPrint(e.toString());
    }
  }
}

```

Kuvio 29. send_fcm_token.dart -tiedosto.

Kuvion 29 koodissa on toteutettu metodi, joka lähettää FCM-tokenin laiterekisteröintitaustapalveluun. Kutsua käytetään käyttäjän kirjautumisen yhteydessä. Topicit haetaan myös rajapinnasta, jonka jälkeen ne tallennetaan omaan olioonsa, joka puolestaan tallennetaan sovelluksen ”säiliöolioon”, joka sisältää jokaisen eri rajapintakutsun datat.

Topicejen valitseminen toteutetaan sovelluksen asetuksissa. Luodaan uusi sivu, joka hakee rajapintakutsun tallennetut tiedot ja näyttää ne listassa (ks. Kuvio 30).

```
65          Expanded(
66            child: ListView.builder(
67              itemCount: filteredTopics.length,
68              itemBuilder: (context, index) {
69                final TopicItem topic = filteredTopics[index];
70                final String topicName = languageCode == 'fi'
71                    ? topic.topicNameFi
72                    : topic.topicNameEn;
73                final String topicDescription = languageCode == 'fi'
74                    ? topic.topicDescFi
75                    : topic.topicDescEn;
76
77                return SwitchListTile(
78                  title: Text(topicName),
79                  subtitle: Text(topicDescription),
80                  value: _isSubscribedToTopic(topic.topicCode),
81                  onChanged: !_notificationsEnabled
82                      ? null
83                      : (value) {
84                          setState(() {
85                            if (value) {
86                              _subscribeToTopic(topic.topicCode);
87                            } else {
88                              _unsubscribeFromTopic(topic.topicCode);
89                            }
90                          });
91                        },
92                  inactiveThumbColor: Colors.grey[300],
93                  inactiveTrackColor: Colors.grey,
94                ); // SwitchListTile
95              },
96            ), // ListView.builder
97          ), // Expanded
```

Kuvio 30. Topicejen listaus sovelluksessa.

Kuvio 30 on koodikatkelma, joka piirtää sovellukseen listan saatavilla olevista topiceista. Hyödynnetty ”ListView.builder”-metodi hyödyntää listaa topiceista, jotka ovat suodatettu käyttäjän roolin perusteella. Esimerkiksi vieraskäyttäjällä ei näy opiskelijan topic. Metodi palauttaa jokaista topicia kohden ”SwitchListTile”-widgetin, jossa on vipu topicin tilaukselle.

Lopulta jäljelle jää toteuttaa ominaisuus, jolla ohjataan käyttäjä oikealle sivulle ilmoituksen ”route”-kentän mukaisesti (ks. Kuvio 31).

```
19  class _LifeCycleManagerState extends ConsumerState<LifeCycleManager>
20      with
21          // ignore: prefer_mixin
22          WidgetsBindingObserver {
23    @override
24    void initState() {
25      WidgetsBinding.instance.addObserver(this);
26
27      FirebaseMessaging.onMessageOpenedApp.listen(_handleBackGroundmessage);
28
29      super.initState();
30    }
31
32    @override
33    void dispose() {
34      WidgetsBinding.instance.removeObserver(this);
35
36      super.dispose();
37    }
38
39    @override
40    void didChangeAppLifecycleState(AppLifecycleState state) {
41      if (state == AppLifecycleState.resumed) {
42        // ignore: unused_result
43        ref.refresh(shouldUpdateDataProvider);
44        if (ref.watch(shouldUpdateDataProvider)) {
45          ref.read(repositoryProvider.notifier).getRepositoryData();
46        }
47      }
48    }
49
50    @override
51    Widget build(BuildContext context) {
52      return Container(
53        child: widget.child,
54      );
55    }
56
57    void _handleBackGroundmessage(RemoteMessage message) {
58      navigatorKey.currentState?.pushNamed(
59        message.data['route'] ?? '/',
60      );
61    }
62  }
```

Kuvio 31. LifeCycleManager-widget.

Kuvion 31 “LifeCycleManager”-widget huomaa muutokset sovelluksen tilassa. Widgetin initState-metodiin riville 27 lisätään kuuntelija, joka kutsuu metodia _handleBackgroundMessage, kun sovellus avataan taustalta ilmoitusta klikatessa. Metodi (rivit 57–61) käyttää globaalia navigaatioavainta ohjatakseen sovelluksen ilmoituksen ”route”-parametrin mukaiselle sivulle. Tilanteessa, jolloin sovellus on päällä, kun ilmoitusta klikataan, ohjataan myös käyttäjä navigointiavaimen avulla oikealle sivulle. Mikäli sovellus on pois päältä, kun ilmoitusta klikataan, on lisättävä ohjaus kirjautumisprosessiin (ks. Kuvio 32).

```
59    Future<void> _checkInternetConnection() async {
60      await Future.delayed(const Duration(seconds: 1), () {});
61      if (!mounted) {
62        return;
63      }
64      final bool hasConnection = await hasInternetConnection();
65      if (!mounted) {
66        return;
67      }
68      final bool hasOfflineData =
69          await ref.read(repositoryProvider.notifier).hasOfflineData();
70      if (!mounted) {
71        return;
72      }
73      if (hasConnection) {
74        setState(() {
75          _showButtonOfflineMode = false;
76        });
77        await _loginAndGetData();
78        if (widget.loginWithRefreshToken) {
79          /// If there is a route to navigate to, then navigate to it after logging in.
80          if (_messageRouteData.isNotEmpty) {
81            await Navigator.of(context).pushNamed(_messageRouteData);
82          }
83        }
84      } else {
85        setState(() {
86          _showButtonOfflineMode = hasOfflineData;
87        });
88        _showNoConnetionDialog();
89      }
90    }
```

Kuvio 32. _checkInternetConnection -metodi.

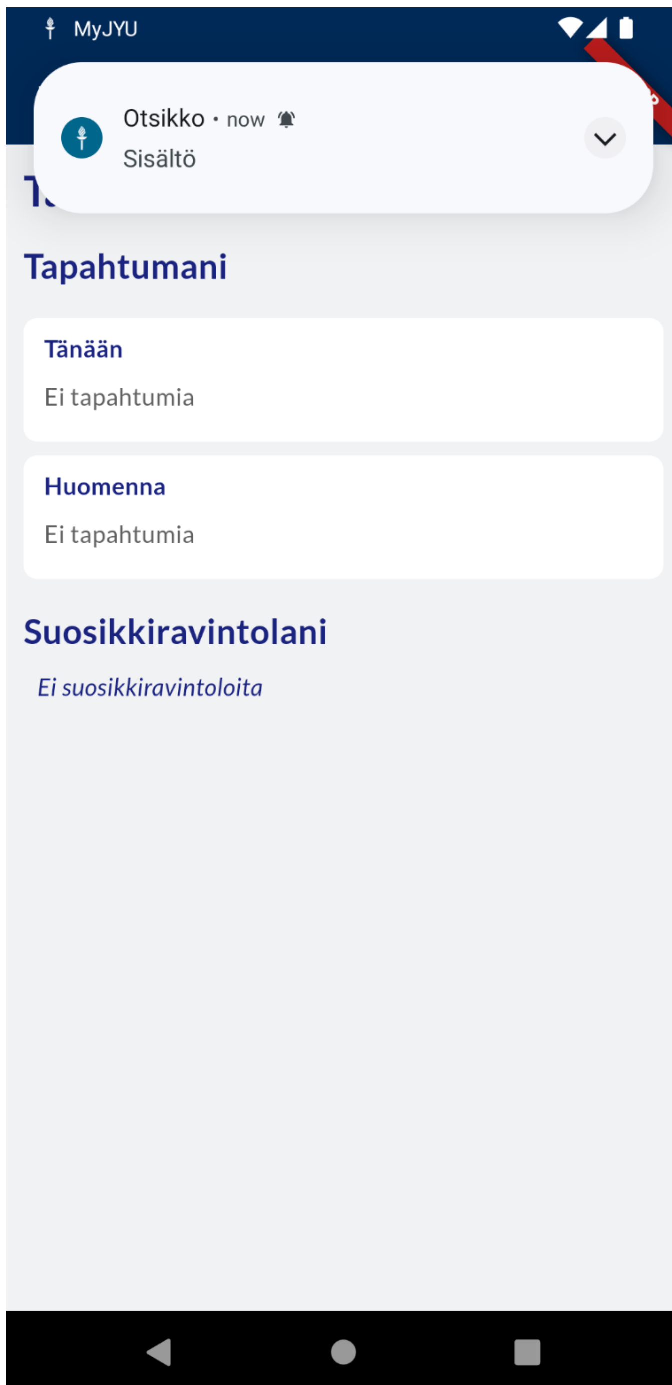
Kuvion 32 metodi, joka tarkastaa internetyhteyden, kutsutaan sovelluksen saapuessa kirjautumissivulle. Rivillä 73–83 aloitetaan kirjautumisprosessi, jos internetyhteys löytyy. Rivillä 78 tarkistetaan, kirjautuuko käyttäjä refresh tokenilla, ja jos tämä pitää paikkansa, ohjataan käyttäjä ”route”-parametrin mukaiselle sivulle.

4.3 Testaus ja julkaisu

Testaus toteutetaan käyttäen Applen TestFlightia, jolla julkaistaan testiversio sovelluksesta suljetulle testausryhmälle, tässä tapauksessa Jyväskylän yliopiston digipalveluiden henkilökunta. Android-version testaukseen käytetään Google Play -kaupan vastaavaa suljettua testausta. Sovelluspäivitys tullaan julkaisemaan muille käyttäjille Applen App Storessa ja Google Play –kaupassa myöhemmällä ajankohdalla joulukuun 2023 aikana.

5 Tulokset

Työn tuloksena saatiin valmiiksi tarvittavat taustapalvelut ja MyJYU-sovelluksen ilmoitusominaisuuden lisäys. Kuviossa 33 on työssä tehtyjen palveluiden avulla lähetetty push-ilmoitus Android-emulaattorilla pyörivään MyJYU-sovellukseen. Ominaisuus toimii myös Apple-laitteilla.



Kuvio 33. Ilmoitus MyJYU:ssa.

6 Pohdinta

Työssä opittiin yksi tapa, miten voidaan toteuttaa ilmoitusominaisuus mobiilisovellukseen. Työ vaati tutustumista sekä itse MyJYU-sovelluksen olemassa olevaan koodiin sekä moneen uuteen teknologiaan, joita en ollut ennestään käyttänyt. Taustapalveluiden kehittäminen Pythonilla ja Docker konttien hyödyntäminen olivat uusia ja opettavaisia kokemuksia.

Ilmoituksia voidaan lähettää melkein mistä tahansa, joten ominaisuudella on paljon jatkokehitysmahdollisuuksia, esimerkiksi opintosuorituksista tulevat ilmoitukset. Tämä vaatisi kehitystä opintotietojärjestelmässä, johon pitäisi lisätä ominaisuus lähettää viestejä RabbitMQ-viestijonoon, jota työssä kehitetty taustapalvelu kuuntelee. Tämä voisi nostaa käyttäjäien interaktiota MyJYU-sovelluksen kanssa huomattavasti, sillä uudet suoritukset ovat opiskelijoille varmasti kiinnostavia.

Sovelluksien testaus jäi hyvin vähäiseksi tai käytännössä olemattomaksi. Laiterekisteröintirajapinta ja viestijonokuuntelija ovat molemmat melko yksinkertaisia sovelluksia, joihin olisi mahdollista luoda kattavat yksikkötestit melko kohtalaisella työmäärällä, mutta aikarajoitusten takia nämä jäivät tekemättä. Palveluissa on kuitenkin suuresti käytetty lokitusta, joka auttaa mahdollisissa virhetilanteissa tulevaisuudessa.

Tietokannan siivousta turhista tokeneista olisi voinut tehostaa luomalla jonkinlainen ohjelma, joka tarkastaa tietokannasta vanhoja FCM-tokeneita ja poistaa ne, estäen tietokannan koon kasvamisen liian isoksi. Koska käyttäjän FCM-tokenin aikaleima päivitetään tietokantaan joka kerta, kun MyJYU sovellukseen kirjaudutaan, voitaisiin vanhoista aikaleimoista olettaa, että käyttäjä ei enää käytä sovellusta, jolloin FCM-token tietokannassa on turha. Aika, kunnes FCM-token poistettaisiin tietokannasta, pitäisi olla riittävän suuri, että opiskelijoiden FCM-tokenit eivät poistuisi kesälomien aikana sovelluksen käyttämättä jättämisen takia.

Kaiken kaikkiaan ominaisuus on julkaisua vaille valmis, joten koen, että tämä työ oli onnistunut hyvin.

Lähteet

Add Firebase to your Flutter app. 2023. Firebase dokumentaatio. Päivitetty 27.11.2023. Viitattu 3.12.2023. https://firebase.google.com/docs/flutter/setup?platform=ios .

Docker overview. N.d. Dockerin dokumentaatio. Viitattu 3.12.2023 https://docs.docker.com/get-started/overview/ .

Dogtiev, A. 2023. Push Notifications Statistics. Viitattu 24.11.2023 https://www.businessofapps.com/marketplace/push-notifications/research/push-notifications-statistics/ .

FAQ. N.d. Flutterin dokumentaatio. Viitattu 23.11.2023. https://docs.flutter.dev/resources/faq .

FCM Architectural Overview. 2023. Firebase Cloud Messaging dokumentaatio. Viitattu 24.11.2023 https://firebase.google.com/docs/cloud-messaging/fcm-architecture .

Johansson, L. Part 1: RabbitMQ for beginners – What is RabbitMQ. Viitattu 3.12.2023. https://www.cloudamqp.com/blog/part1-rabbitmq-for-beginners-what-is-rabbitmq.html .

Jyväskylän yliopiston digipalveluiden verkkosivut. N.d. Viitattu 20.11.2023. https://www.jyu.fi/digipalvelut/fi .

Korzeń, K. 2022 Benefits of push notifications. Viitattu 24.11.2023 https://crustlab.com/blog/benefits-of-push-notifications/ .

Mixon, E., Steele, C. 2023. Definition, push notification. Päivitetty 2/2023. Viitattu 20.11.2023. https://www.techtarget.com/searchmobilecomputing/definition/push-notification .

Mobile Push Notifications. 2023. OneSignal dokumentaatio. Päivitetty 8/2023 Viitattu 21.11.2023. https://documentation.onesignal.com/docs/mobile-push-notifications-guide .

MyJYU-sovellus helpottaa arkeasi! N.d. Viitattu 20.11.2023 https://www.jyu.fi/fi/opiskelijalle/kandi-ja-maisteriopiskelijan-ohjeet/uuden-opiskelijan-kasikirja/myjyu-sovellus-helpottaa-arkeasi .

RabbitMQ verkkosivut. N.d. Verkkosivu. Viitattu 3.12.2023. https://www.rabbitmq.com/ .

Ravoof, S. 2023. Python vs Java: Pick What’s Best for Your Project. Verkkoblogi. Viitattu 4.12.2023. https://kinsta.com/blog/python-vs-java/ .

Sneath, T. 2018. Flutter 1.0: Google’s Portable UI Toolkit. Viitattu 23.11.2023. https://developers.googleblog.com/2018/12/flutter-10-googles-portable-ui-toolkit.html .