KAUNAS UNIVERSITY OF TECHNOLOGY

FACULTY OF INFORMATICS

T120B169 Fundamentals of App Development

University schedule app

IFE-8, Vladas Bukinas

IFE-8, Martynas Kemežys

IFE-8, Augustė Viršilaitė

Date: 2021.11.15

Kaunas, 2021

Table of contents

Description of the app

In times like these, when a global pandemic has interfered with everyone's lives, it is now more useful than ever to have a day-to-day plan. Not only does knowing what to expect in the upcoming days help us keep up with our responsibilities, but it also lets us prepare to safely reenter society.

Places that are more susceptible to COVID-19 outbreaks include educational institutions [1], therefore, they can greatly benefit from organizing the flow of students. Keeping this in mind, our team has decided to create a **university schedule app**, which mainly focuses on preventing the spread of COVID-19 in universities.

The app has two types of users: lecturers and students. Lecturers are able to create one-time or recurring lectures and manipulate them. Created lectures are displayed in the calendar for intended students. They can register their attendance, which then becomes visible for the responsible lecturer. In case of a positive COVID-19 case, the lecturer, in whose lecture it took place, is able to notify the attendees and provide them with necessary information. Besides that, lecturers and students see upcoming lectures, which improves the ease of planning.

Students can also create single (one-time) events, which are only visible to them, in order to customize their schedules and expand usability of the app.

The university schedule app is hoped to be a useful and easy-to-use tool for safely reopening educational institutions.

Functionality of the app

List of functions

1. connect project to a remote database;
2. add a calendar which displays events;
3. create a function for adding one-time events;
4. create a function for adding recurring events;
5. add a function for deleting events in the calendar;
6. implement app navigation through a sidebar menu;
7. implement the filtering of events by color;
8. add a function for viewing events in the calendar;
9. add authentication functions;
10. add a function for viewing upcoming events;
11. add a function for viewing profile;
12. add a function for an event search;
13. add a function for increasing events' progress.

Solution

Task #1: connect project to a remote database

Originally, the project used *Android Room* [2] local database. However, since some data needs to be accessed by different users (e. g., students need to see the events added by lecturers), project was switched to *Firestore* [3] remote database. Code snippets, as seen in **Figure 1**, were added.

```
fdb.firestoreSettings = FirebaseFirestoreSettings.Builder().build()
mAuth = FirebaseAuth.getInstance()
```

Figure 1. Example of code used for authentication of remote database

Task #2: add a calendar which displays events

Calendar was implemented using the *Android Week View* library [4]. This library provides a component which shows the day of the week and the time of the day for an upcoming week. The component's UI can be seen in **Figure 2** and parts of code in **Figure 3** and **Figure 4**.

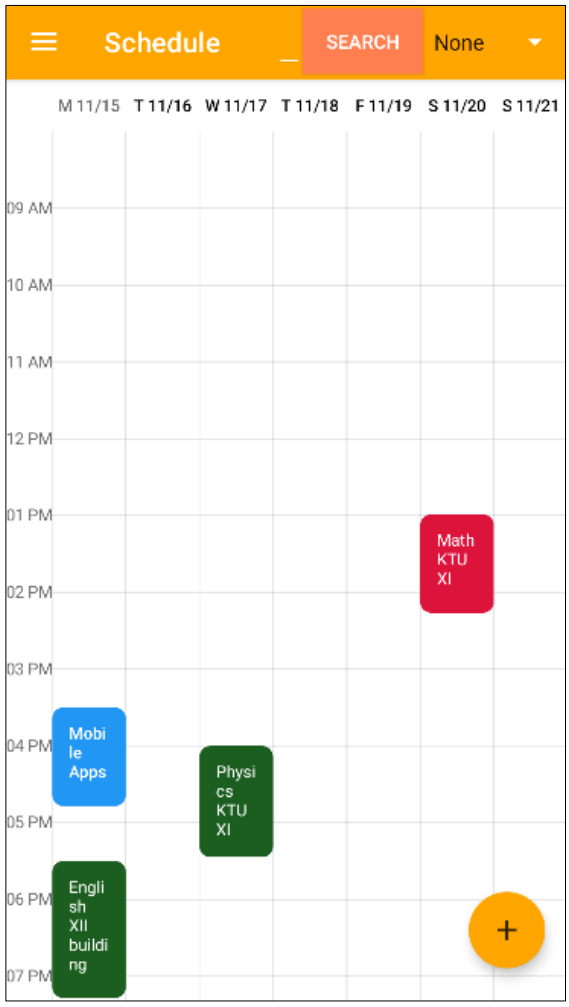

Figure 2. Screenshot of the *Android Week View* component

```
override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
): View? {
    binding = FragmentScheduleBinding.inflate(inflater)

    val adapter =   ScheduleAdapter(clickListener = this::onLongClick)

    viewModel.events.observe(viewLifecycleOwner){  it: List<Event>
        adapter.submitList(it)
    }

    binding.weekView.minHour  = 8
    binding.weekView.maxHour  = 20

    binding.weekView.numberOfVisibleDays = 7
    binding.weekView.minDateAsLocalDate = convertLongToLocalDate(semesterStart)
    binding.weekView.maxDateAsLocalDate = convertLongToLocalDate(semesterEnd)

    binding.weekView.showFirstDayOfWeekFirst

    binding.weekView.adapter = adapter
    binding.lifecycleOwner = viewLifecycleOwner

    binding.addEvent.setOnClickListener{  it: View!
        view?.findNavController()?.navigate(R.id.action_scheduleFragment_to_createEventFragment)
    }

    return binding.root
}
```

Figure 3. Code of the *ScheduleFragment*

```
class ScheduleAdapter( private val clickListener: (data:Event) -> Unit) : WeekView.SimpleAdapter<Event>() {
    override fun onCreateEntity(item: Event): WeekViewEntity {
        val style = WeekViewEntity.Style.Builder()
                .setBackgroundColor(item.color)
                .build()
       return WeekViewEntity.Event.Builder(item)
               .setId(item.id)
               .setTitle(item.title)
               .setStartTime(item.startTime)
               .setEndTime(item.endTime)
               .setSubtitle(item.location)
               .setStyle(style)
               .build()
    }
    override fun onEventLongClick(data: Event) {
        if (data is Event) {
            clickListener(data)
        }
    }
}
```

Figure 4. Code of the *ScheduleAdapter*

Task #3: create a function for adding one-time events

A function for adding events (lectures) was implemented by creating a local database with the help of the *Android Room* library [2], which simplifies manipulation of data. After that, a form with 6 input fields: event title, date, start time, duration, location and event color, was added. The form gets validated using the *Kotlin Flow* library [5], which asynchronously checks if the entered values are correct. Finally, the form can be accessed by clicking the "+" symbol that is visible on the bottom-right corner of the *Schedule* component (see **Figure 2**). The implemented function's UI is displayed in **Figure 5** and parts of code in **Figure 6** and **Figure 7**.

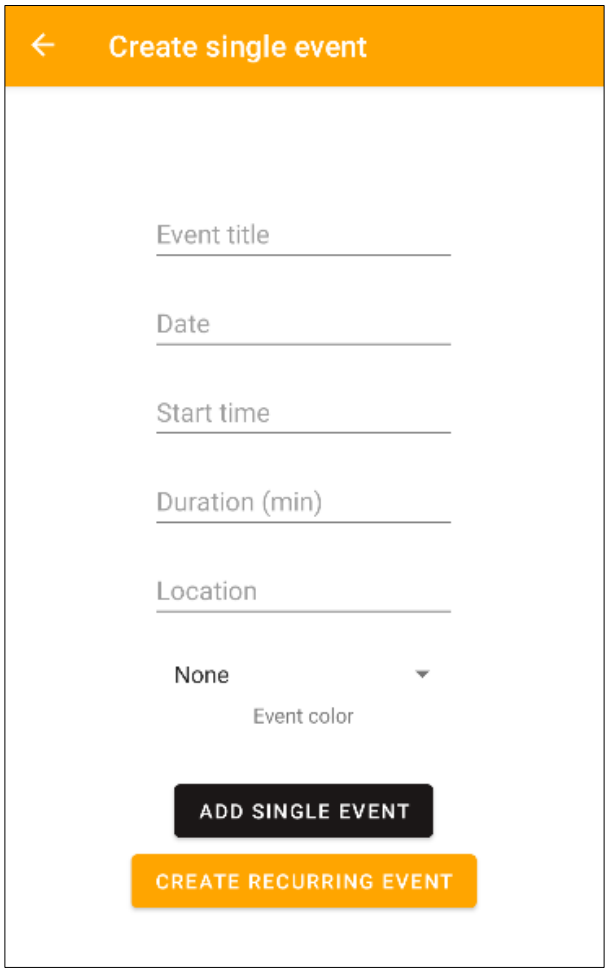

Figure 5. Screenshot of the *CreateEventFragment*

```
override fun onCreateView(
        inflater: LayoutInflater, container: ViewGroup?,
        savedInstanceState: Bundle?
): View? {

    binding = FragmentCreateEventBinding.inflate(inflater, container, attachToRoot: false)

    val spinner: Spinner = binding.selectEventColors
    ArrayAdapter.createFromResource(
            activity?.applicationContext!!,
            R.array.colors,
            android.R.layout.simple_list_item_1
    ).also { adapter ->
        adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item)
        spinner.adapter = adapter
    }

    binding.startTimeInput.isFocusable = false
    binding.startTimeInput.setOnClickListener{ it: View!
        setTimeFromTimePicker(context, binding.startTimeInput)
    }

    binding.selectDayInput.isFocusable = false
    binding.selectDayInput.setOnClickListener{ it: View!
        setDateFromDatePicker(context, binding.selectDayInput)
    }

    with(binding) { this: FragmentCreateEventBinding
        selectDayInput.doOnTextChanged { text, _, _, _ ->
            date.value = text.toString()
        }
        startTimeInput.doOnTextChanged { text, _, _, _ ->
            startTime.value = text.toString()
        }
        eventDurationInput.doOnTextChanged { text, _, _, _ ->
            duration.value = text.toString()
        }
        eventNameInput.doOnTextChanged { text, _, _, _ ->
            event.value = text.toString()
        }
        locationInput.doOnTextChanged { text, _, _, _ ->
            location.value = text.toString()
        } ^with
    }

    val snackBar = activity?.let { Snackbar.make(it.findViewById(R.id.drawer_layout), text: "Event added!", Snackbar.LENGTH_LONG) }

    binding.createEventBtn.setOnClickListener { it: View!
        if (snackBar != null) {
            snackBar.show()
            viewModel.addEvent(date.value,
                    startTime.value,
                    duration.value,
                    event.value,
                    spinner.selectedItem.toString(),
                    location.value)
            binding.selectDayInput.text.clear()
            binding.eventDurationInput.text.clear()
            binding.startTimeInput.text.clear()
            binding.eventNameInput.text.clear()
            binding.locationInput.text.clear()
        }
    }
    lifecycleScope.launch { this: CoroutineScope
        formIsValid.collect { it: Boolean
            binding.createEventBtn.apply { this: Button
                backgroundTintList = ColorStateList.valueOf(
                        Color.parseColor(
                                if (it) onFormValidButtonTintColor else defaultButtonTintColor
                        )
                )
                isClickable = it
            }
        }
    }
    return binding.root
}
```

Figure 6. Code of the *CreateEventFragment*

```
private val formIsValid = combine(date, startTime, duration, event, location)
{ date, startTime, duration, event, location ->
    binding.txtErrorMessage.text = ""

    var valid       =   dateIsValid(date)
    var longDate    =   convertLocalDateToLong(valid)
    val startTimeValues = startTime.split( ...delimiters: ":")
    val dateIsValid =   valid != null && longDate!! <= semesterEnd!! && longDate!! >= semesterStart!!
    val duration    =   duration.length in 1..3 && duration.toInt() <= 300 && duration.toInt() >= 60
    val startTimeIsValid =  startTimeValues[0].length in 1..2 &&
                            startTimeValues[0].toInt() <= 19 &&
                            startTimeValues[0].toInt() >= 8
    val event = event.length < 30 && event.isNotEmpty()
    val location = location.length < 30 && location.isNotEmpty()

    errorMessage = when {
        dateIsValid.not() -> "Date is not valid"
        startTimeIsValid.not() -> "Start time is not valid"
        duration.not() -> "Duration is not valid"
        event.not() -> "Event is not valid"
        location.not() -> "Location is not valid"

        else -> null
    }
    errorMessage?.let { it: String
        if(date.isNotEmpty()) {
            binding.txtErrorMessage.text = it
        }
    }
    dateIsValid and duration and startTimeIsValid and event and location ^combine
}
```

Figure 7. Code of the validation function

Task #4: create a function for adding recurring events

Function for adding recurring events was implemented with the help of *Firestore* [3], which simplifies manipulation of data. After that, a form with 7 input fields: event title, day of the week, start time, duration, event recurrence, location and event color, was added. The form gets validated using the *Kotlin Flow* library [5], which asynchronously checks if the entered values are correct. Finally, the form can be accessed by clicking the "Create recurring event" button that is visible in the bottom of the *Create Event* component (see **Figure 4**). The implemented function's UI is displayed in **Figure 8** and some of its code in **Figure 9**.

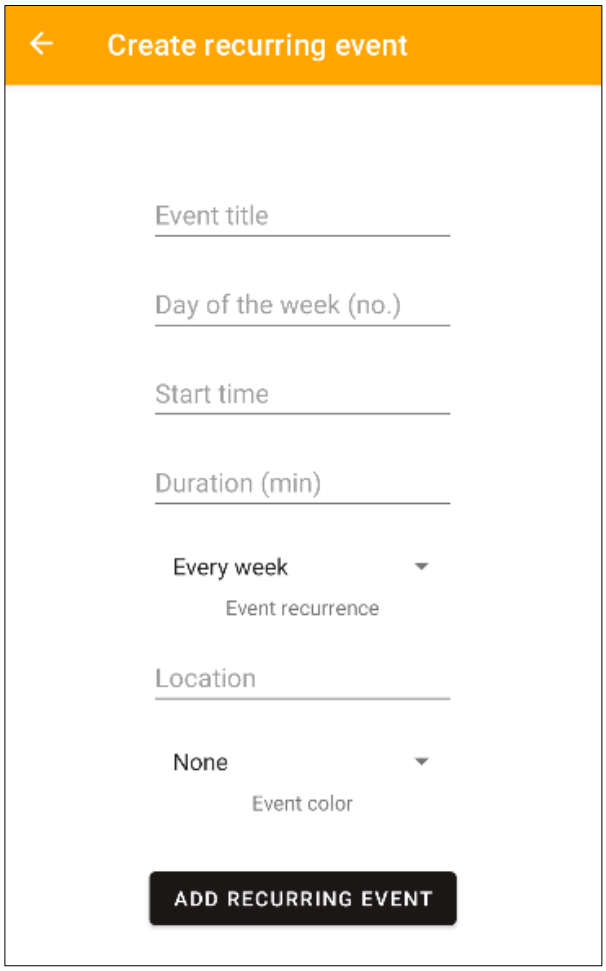

Figure 8. Screenshot of the *MassAddEvents* component

```
class MassAddEvents : Fragment() {
    private lateinit var binding: FragmentMassAddEventsBinding

    private val defaultButtonTintColor = "#1B1717"
    private val onFormValidButtonTintColor = "#4F774F"

    private val weekDay = MutableStateFlow( value: "")
    private val startTime = MutableStateFlow( value: "")
    private val duration = MutableStateFlow( value: "")
    private val event = MutableStateFlow( value: "")
    private val location = MutableStateFlow( value: "")

    private var errorMessage: String? = null

    private lateinit var prefs: SharedPreferences
    private var semesterStart : Long? = null
    private var semesterEnd : Long? = null


    override fun onAttach(context: Context) {
        super.onAttach(context)
        prefs = requireContext().getSharedPreferences("ktu.edu.projektas.app", Context.MODE_PRIVATE)
        semesterStart = prefs.getLong("ktu.edu.projektas.app.semester_start", getCurrentMonthFirstDay()?.toEpochMilli()!!)
        semesterEnd = prefs.getLong("ktu.edu.projektas.app.semester_end", getCurrentMonthLastDay()?.toEpochMilli()!!)
    }

    private val viewModel : ScheduleViewModel by activityViewModels {
        ScheduleViewModelFactory(requireContext(), semesterStart!!, semesterEnd!!)

    }
    private val formIsValid = combine(
        weekDay,
        startTime,
        duration,
        event,
        location
    ) { weekDay, startTime, duration, event, location ->
        binding.txtErrorMessageMass.text = ""

        val startTimeValues = startTime.split( ...delimiters: ":")

        val weekDayValid        =       weekDay.length == 1 && weekDay.toInt() > 0 && weekDay.toInt() <= 7
        val duration            =       duration.length in 1..3 && duration.toInt() <= 300 && duration.toInt() >= 60
        val startTimeIsValid    =       startTimeValues[0].length in 1..2 &&
                startTimeValues[0].toInt() <= 19 &&
                startTimeValues[0].toInt() >= 8
        val event               =       event.length < 30 && event.isNotEmpty()
        val location            =       location.length < 30 && location.isNotEmpty()

        errorMessage = when {
            weekDayValid.not() -> "Day of the week is invalid - has to be expressed as number from 1 to 7"
            startTimeIsValid.not() -> "Start time is invalid - event has to take place between 8:00 and 19:00"
            duration.not() -> "Duration is invalid - event has to last from 60 to 300 minutes"
            event.not() -> "Event title is invalid"
            location.not() -> "Location is invalid"

            else -> null
        }
        errorMessage?.let {  it: String
            if(weekDay.isNotEmpty()) {
                binding.txtErrorMessageMass.text = it
            }
        }
        weekDayValid and duration and startTimeIsValid and event and location  ^combine
    }
```

Figure 9. Code snippet of the *MassAddEvents*

Task #5: add a function for deleting events in the calendar

A function for deleting events was implemented by using the previously mentioned *Android Room* [2] library's database and the *Android Week View* [4] library, which provides a callback *onEventLongClick()*. It was used to trigger the delete function in the *ScheduleViewModel* class. The implemented function's UI is displayed in **Figure 10** and the code in **Figure 11**.

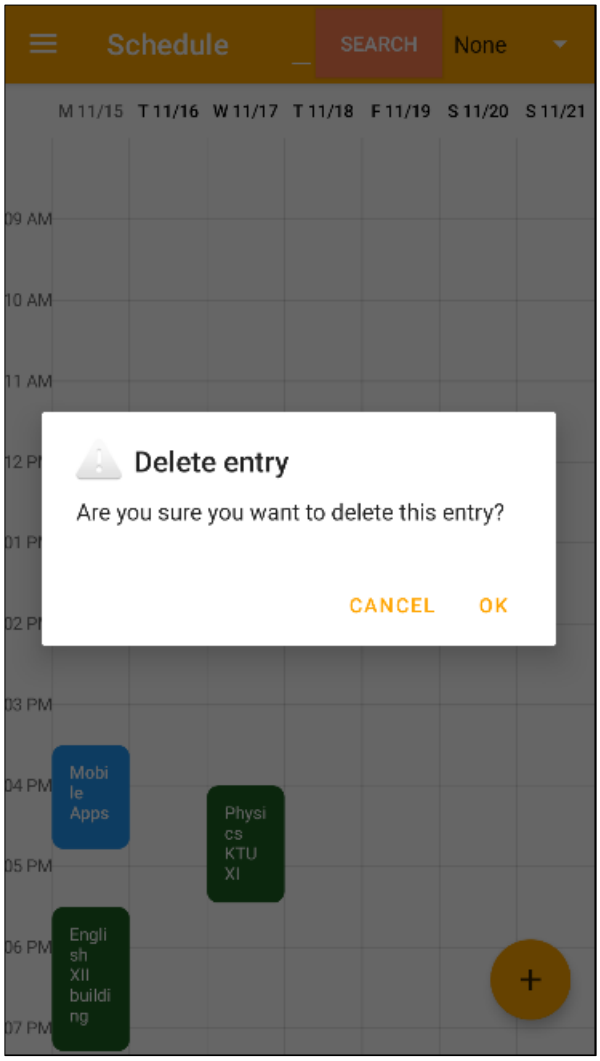

Figure 10. Screenshot of removing events

```
private fun onLongClick(event: Event) {
    AlertDialog.Builder(context)
            .setTitle("Delete entry")
            .setMessage("Are you sure you want to delete this entry?")
            .setPositiveButton(android.R.string.yes) { dialog, which ->
                viewModel.deleteByGroup(event.groupId)
            }
            .setNegativeButton(android.R.string.no,  listener: null)
            .setIcon(android.R.drawable.ic_dialog_alert)
            .show()
}
```

Figure 11. Code of the function for removing events

Task #6: implement app navigation through a sidebar menu

App navigation was implemented by using the *Android NavigationUI* library [7]. The navigation is done mostly through the *Drawer*, which is opened by sliding to the right or clicking on the "hamburger" icon. The implemented function's UI is displayed in **Figure 12** and the code in **Figure 13.**

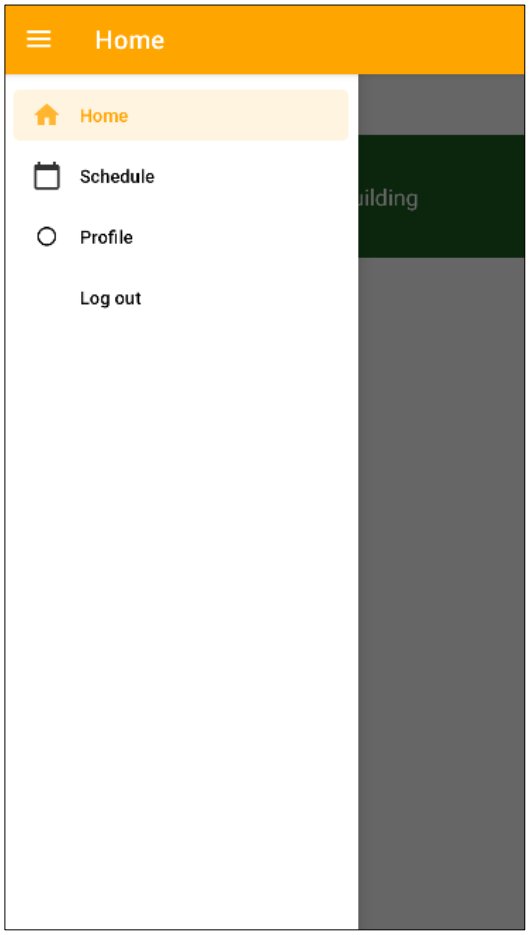

Figure 12. Screenshot of the sidebar menu

```
class MainActivity : AppCompatActivity() {
    private lateinit var navController : NavController
    private lateinit var appBarConfiguration: AppBarConfiguration
    private lateinit var drawerLayout:DrawerLayout
    private lateinit var navigationView : NavigationView

    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        setContentView(R.layout.activity_main)

        navController = findNavController(R.id.nav_host_fragment_container)
        drawerLayout = findViewById(R.id.drawer_layout)
        navigationView = findViewById(R.id.navigationView)
        navigationView.setupWithNavController(navController)

        appBarConfiguration =   AppBarConfiguration(setOf(R.id.homeFragment, R.id.scheduleFragment,R.id.profileFragment), drawerLayout)
        setupActionBarWithNavController(navController, appBarConfiguration)
    }

    override fun onSupportNavigateUp(): Boolean {
        val navController = findNavController(R.id.nav_host_fragment_container)
        return navController.navigateUp(appBarConfiguration) || super.onSupportNavigateUp()
    }
}
```

Figure 13. Code of the app navigation

Task #7: implement the filtering of events by color

A function for filtering events (lectures) by color was implemented by a query, which selects events with a chosen color from the database. The implemented function's UI is displayed in **Figure 14** and the code in **Figure 15**, **Figure 16** and **Figure 17**.

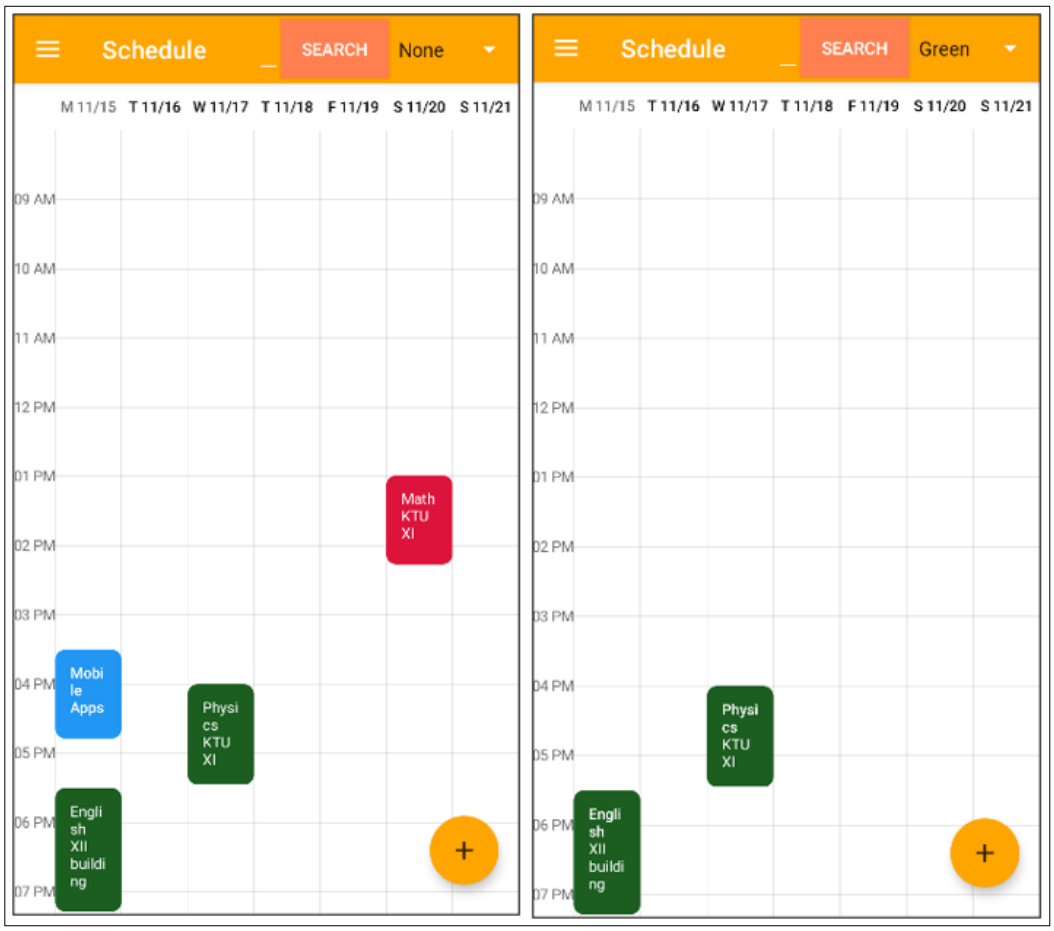

Figure 14. Screenshot of filtering events by color

```
fun getAllEventsByColor(color:String) : LiveData<List<Event>>? {

    val id = getColorCode(color)
    // if the color isn't defined, return all events
    if(id == -1)
        return events

    var data : LiveData<List<Event>>? = null

    viewModelScope.launch {  this: CoroutineScope
        data = db.ScheduleDao().getAllEventsByColor(id).asLiveData()
    }
    return data
}
```

Figure 15. Code of the filtering of the events

```
@Query( value: "SELECT * FROM events WHERE events.color = :color")
fun getAllEventsByColor(color:Int): Flow<List<Event>>
```

Figure 16. Code of the query for filtering

```
override fun onCreateOptionsMenu(menu: Menu, inflater: MenuInflater) {
    inflater.inflate(R.menu.top_menu, menu)

    val item: MenuItem = menu!!.findItem(R.id.spinner)
    spinner = item.actionView as Spinner
    // Fill spinner with color list
    activity?.let {  it: FragmentActivity
        ArrayAdapter.createFromResource(
            it.applicationContext,
            R.array.colors, android.R.layout.simple_spinner_item
        ).also { adapter ->
            adapter.setDropDownViewResource(android.R.layout.simple_spinner_dropdown_item)
            spinner.adapter = adapter
        }
    }
    // On selected listener to change data when spinner is changed
    spinner.onItemSelectedListener = object : OnItemSelectedListener {
        override fun onItemSelected(
            parentView: AdapterView<*>?,
            selectedItemView: View?,
            position: Int,
            id: Long
        ) {
            var selectedItem  = spinner.selectedItem.toString()
            binding.weekView.adapter = null
            // Get events by color
            viewModel.getAllEventsByColor(selectedItem)?.observe(viewLifecycleOwner){  it: List<Event
                adapter.submitList(it)
            }
            binding.weekView.adapter = adapter
        }

        override fun onNothingSelected(parentView: AdapterView<*>?) {
            binding.weekView.adapter = null
            // Reset to normal event data
            viewModel.events.observe(viewLifecycleOwner){  it: List<Event>
                adapter.submitList(it)
            }
            binding.weekView.adapter = adapter
        }
    }
    return super.onCreateOptionsMenu(menu!!, inflater!!)
}
```

Figure 17. Code of the *ScheduleFragment* part for event filtering

Task #8: add a function for viewing events in the calendar

A function for viewing events in the calendar was implemented by adding *EventFragment* and setting data to display. The implemented function's UI is displayed in **Figure 18** and the code snippet in **Figure 19**.

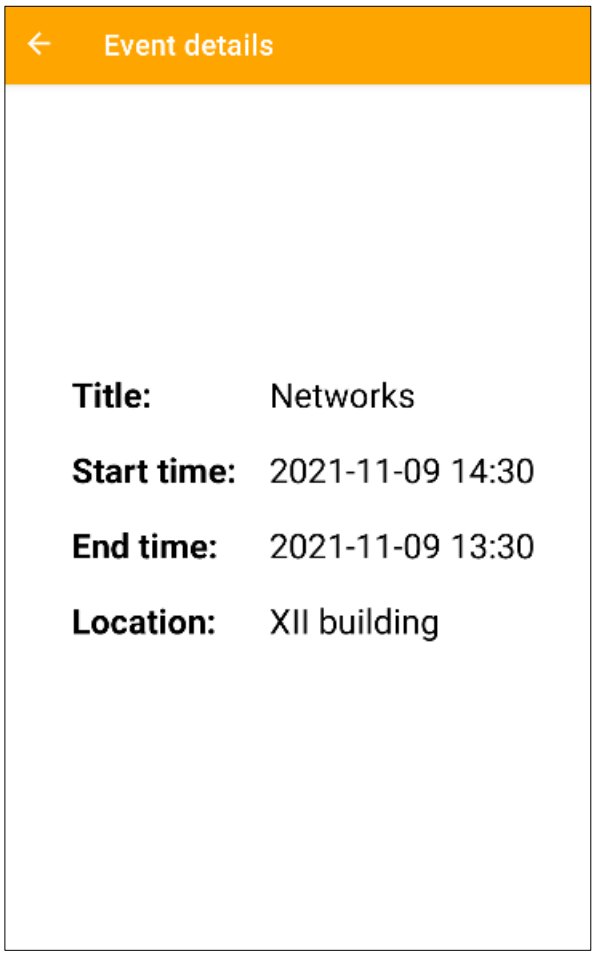

Figure 18. Screenshot of viewing an event

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {

    binding = FragmentEventBinding.inflate(inflater, container, attachToRoot: false)

    val args = EventFragmentArgs.fromBundle(requireArguments())

    binding.eventNameText.text = args.eventName
    binding.startTimeText.text = formatLocalDateTime(longToLocalDateTime(args.startTime.toLong()))
    binding.endTimeText.text = formatLocalDateTime(longToLocalDateTime(args.endTime.toLong()))
    binding.locationText.text = args.location

    binding.lifecycleOwner = viewLifecycleOwner

    return binding.root

}
```

Figure 19. Code of *EventFragment*

Task #9: add authentication functions

Authentication functions, such as *sign up*, *log in* and *log out*, were implemented by using *Firebase* authentication system [7]. The implemented functions' UI is displayed in figures **Figure 20** and **Figure 21,** and code snippets in figures **Figure 22** and **Figure 23**.

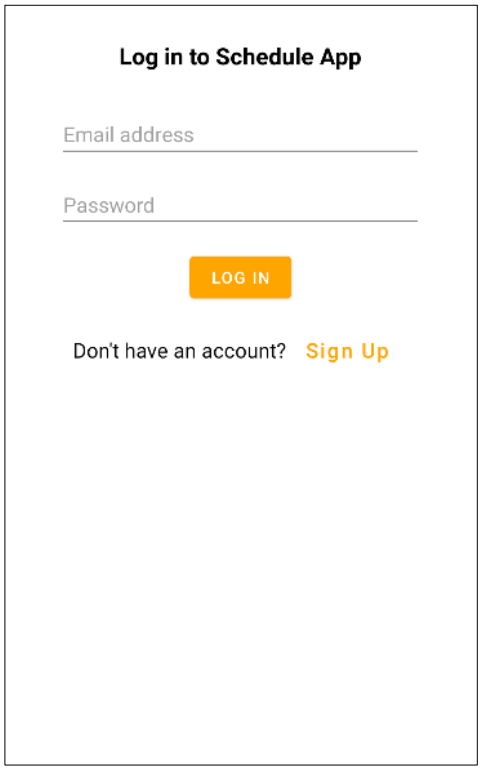

Figure 20. Screenshot of *LoginFragment*

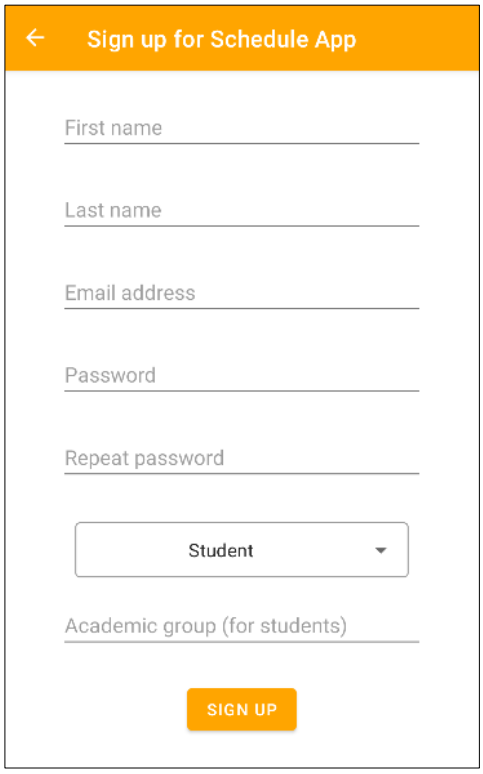

Figure 21. Screenshot of *RegisterFragment*

```
fun userLogin(){
    if(binding.etEmail.text.toString().trim().isEmpty()) {
        binding.etEmail.error = "Email is required!"
        binding.etEmail.requestFocus()
        return
    }
    if(!Patterns.EMAIL_ADDRESS.matcher(binding.etEmail.text.toString()).matches()) {
        binding.etEmail.error = "Please provide a valid email!"
        binding.etEmail.requestFocus()
        return
    }
    if(binding.etPassword.text.toString().trim().isEmpty()) {
        binding.etPassword.error = "Password is required!"
        binding.etPassword.requestFocus()
        return
    }
    if(binding.etPassword.text.toString().length < 6){
        binding.etPassword.error = "Password is too short!"
        binding.etPassword.requestFocus()
        return
    }

    mAuth.signInWithEmailAndPassword(binding.etEmail.text.toString(),binding.etPassword.text.toString()).addOnCompleteListener{
            task ->
        if(task.isSuccessful){
            view?.findNavController()?.navigate(R.id.action_loginFragment_to_homeFragment)
        }
        else {
            activity?.let { Snackbar.make(it.findViewById(R.id.drawer_layout),
                text: "Failed to login, please check your credentials!", Snackbar.LENGTH_LONG) }
                ?.show()
        }
    }

}
```

Figure 22. Code of *LoginFragment*

```
private fun registerUser(){

    if(binding.etFirstName.text.toString().trim().isEmpty()) {
        binding.etFirstName.error = "First name is required!"
        binding.etFirstName.requestFocus()
        return
    }
    if(binding.etLastName.text.toString().trim().isEmpty()) {
        binding.etLastName.error = "Last name is required!"
        binding.etLastName.requestFocus()
        return
    }
    if(binding.etEmail.text.toString().trim().isEmpty()) {
        binding.etEmail.error = "Email is required!"
        binding.etEmail.requestFocus()
        return
    }
    if(!Patterns.EMAIL_ADDRESS.matcher(binding.etEmail.text.toString()).matches()) {
        binding.etEmail.error = "Please provide a valid email!"
        binding.etEmail.requestFocus()
        return
    if(binding.etPassword.text.toString().trim().isEmpty()) {
        binding.etPassword.error = "Password is required!"
        binding.etPassword.requestFocus()
        return
    }
    if(binding.etPassword.text.toString().length < 6){
        binding.etPassword.error = "Provide a longer password!"
        binding.etPassword.requestFocus()
        return
    }
    if(binding.etPassword.text.toString() != binding.etRepeatPassword.text.toString()){
        binding.etRepeatPassword.error = "Passwords must match!"
        binding.etRepeatPassword.requestFocus()
        return
    }
    mAuth.createUserWithEmailAndPassword(binding.etEmail.text.toString(),
        binding.etPassword.text.toString()).addOnCompleteListener{
        task ->
        if(task.isSuccessful){
            activity?.let { Snackbar.make(it.findViewById(R.id.drawer_layout),
                 text: "User has been registered!", Snackbar.LENGTH_LONG) }
                ?.show()

            var user = User(binding.etFirstName.text.toString(),
                binding.etLastName.text.toString(),binding.etEmail.text.toString(),
                binding.etRole.text.toString(), binding.etGroup.text.toString())

            FirebaseAuth.getInstance().currentUser?.let
            { fdb.collection( collectionPath: "users").document(it.uid).set(user) }

            view?.findNavController()?.navigate(R.id.action_registerFragment_to_loginFragment)
        }
        else {
            activity?.let { Snackbar.make(it.findViewById(R.id.drawer_layout),
                 text: "An error has occurred, please try again!",
                Snackbar.LENGTH_LONG) }
                ?.show()
        }
    }
```

Figure 23. Code of *RegisterFragment*

Task #10: add a function for viewing upcoming events

A function for viewing upcoming events was created by using an *Adapter* component and querying data from the *Firestore* [3] database. The upcoming event function is placed in the *HomeFragment* (see **Figure 24**). The implemented function's code snippet is displayed in **Figure 25**.

Figure 24. Screenshot of *HomeFragment*

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {
    binding = FragmentHomeBinding.inflate(inflater, container, attachToRoot: false)
    adapter =   HomeAdapter()

    viewModel.upcomingEvents.observe(viewLifecycleOwner, { list ->
        if(list.isNotEmpty()) {
            setVisible(true)
        } else setVisible(false)
        adapter.submitList(list)
    })
    binding.upcomingEventAdapter.adapter = adapter
    binding.lifecycleOwner = viewLifecycleOwner

    return binding.root
}
private fun setVisible(boolean: Boolean){
    binding.upcomingEventAdapter.visibility = if(boolean) View.VISIBLE else View.GONE
    binding.emptyView.visibility = if(boolean) View.GONE else View.VISIBLE
}
```

Figure 25. Code of *HomeFragment*

Task #11: add a function for viewing profile

A function for viewing user's profile page was implemented by querying data from *Firestore* [3] database. The profile page can be accessed through the sidebar menu (see **Figure 12**). The implemented function's UI is displayed in **Figure 26** and the code snippet in **Figure 27**.

Figure 26. Screenshot of *ProfileFragment*

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?,
    savedInstanceState: Bundle?
): View? {

    binding = FragmentProfileBinding.inflate(inflater, container, attachToRoot: false)
    binding.lifecycleOwner = viewLifecycleOwner

    binding.buttonChange.setOnClickListener { it: View!
        view?.findNavController()?.navigate(R.id.action_profileFragment_to_changePwFragment)
    }
    auth = FirebaseAuth.getInstance()
    uid = auth.currentUser?.uid.toString()

    if(uid.isNotEmpty()){
        readFireStoreData()
    }
    binding.buttonLogout.setOnClickListener { it: View!
        logOut()
    }

    return binding.root
}
private fun logOut(){
    auth.signOut()
    view?.findNavController()?.navigate(R.id.action_profileFragment_to_loginFragment)
}
private fun readFireStoreData() {
    val db = FirebaseFirestore.getInstance()
    db.collection( collectionPath: "users").document(uid).get().addOnSuccessListener { documentSnapshot ->
        val firstName = documentSnapshot.getString( field: "firstName")
        val lastName = documentSnapshot.getString( field: "lastName")
        val email = documentSnapshot.getString( field: "email")
        val role = documentSnapshot.getString( field: "role")
        val group = documentSnapshot.getString( field: "group")
        binding.fullname.text = firstName
        binding.lastname.text = lastName
        binding.email.text = email
        binding.layoutRole.text = role
        binding.layoutGroup.text = group
    }
}
```

Figure 27. Code of *ProfileFragment*

Task #12: add a function for an event search

A function for an event search was implemented by querying data from *Firestore* [3] database and displaying events with titles that match looked up keywords. The search tab can be accessed above the calendar and besides filtering by color. The implemented function's UI is displayed in **Figure 28** and the code snippet in **Figure 29**.

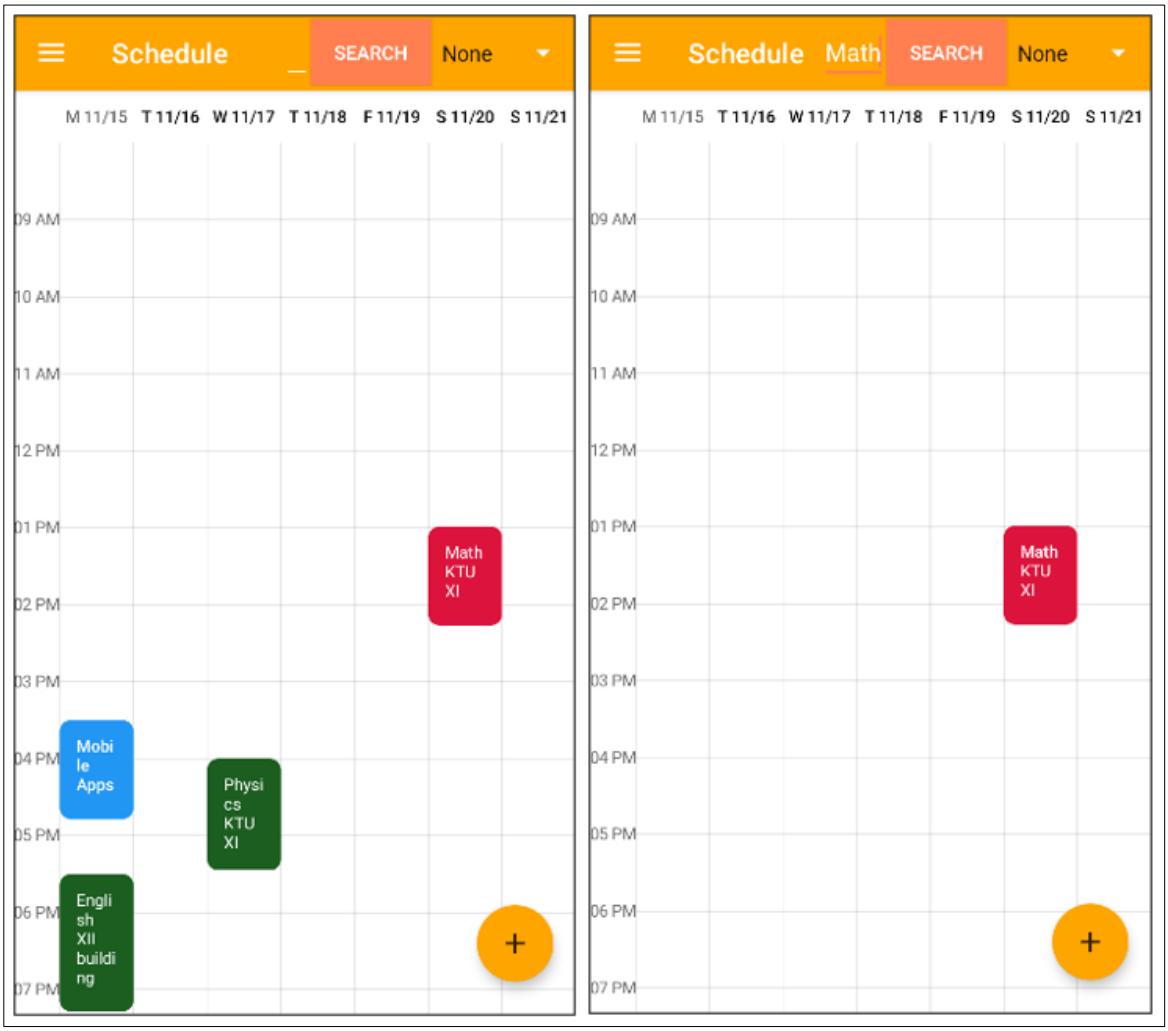

Figure 28. Screenshot of searching for events by title

```
fun getAllEventsByQuery(query:String) : LiveData<List<Event>>? {
    if(query.isEmpty())
        return events

    var data: MutableLiveData<List<Event>> = MutableLiveData<List<Event>>()

    fdb.collection( collectionPath: "events").
    orderBy( field: "title").startAt(query).endAt( ...fieldValues: query+"\uf8ff").addSnapshotListener {
            snapshot, e ->
        if (e != null) {
            Log.w(TAG,  msg: "Listen Failed", e)
            return@addSnapshotListener
        }
        if (snapshot != null) {
            val allEvents = mutableListOf<Event>()
            val documents = snapshot.documents
            documents.forEach {  it: DocumentSnapshot!

                val event = it.toObject(Event::class.java)
                if (event != null) {
                    event.firebaseId = it.id
                    allEvents.add(event!!)
                }
            }
            data.value = Collections.unmodifiableList(allEvents)
        }
    }
    return data
}
```

Figure 29. Code of *ScheduleModeView* part for an event search

Task #13: add a function for increasing events' progress

A function for increasing events' progress was implemented by adding a *drawable* component with a separate layout inside of it and increasing its height with a click of a button. The implemented function's UI is displayed in **Figure 30** and the code snippet in **Figure 31**.

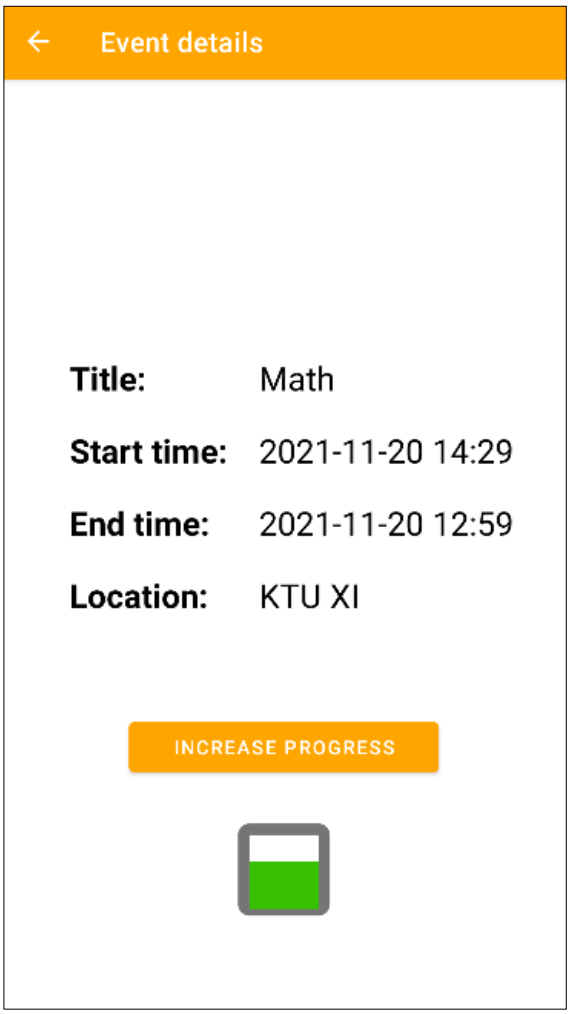

Figure 30. Screenshot of increasing event's progress

```
binding.button.setOnClickListener{
    var graph: View = binding.green
    var params: ViewGroup.LayoutParams = graph.layoutParams
    if(params.height <= 140){
        params.height += 10
    }

    graph =  binding.green;
    graph.layoutParams = params;
}
```

Figure 31. Code snippet of increasing event's progress

Reference list

[1] I. D. S. o. America, „What the Experts Say About COVID-19 Risks,“ [Online]. Available: https://www.idsociety.org/globalassets/idsa/public-health/covid-19/activity-risk.pdf.

[2] „Room,“ [Online]. Available: https://developer.android.com/jetpack/androidx/releases/room.

[3] „Authenticate with Firebase on Android Using a Custom Authentication System,“ [Online]. Available: https://firebase.google.com/docs/auth/android/custom-auth.

[4] T. Hellmund, „Android-Week-View,“ [Online]. Available: https://github.com/thellmund/Android-Week-View.

[5] „Asynchronous Flow,“ [Online]. Available: https://kotlinlang.org/docs/flow.html.

[6] „Navigation,“ [Tinkle]. Online: https://developer.android.com/guide/navigation.

[7] „Cloud Firestore,“ [Online]. Available: https://firebase.google.com/docs/firestore.