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: 2021.10.18

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.

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. add a calendar component which displays events;
2. create a function for adding one-time events;
3. add a function for deleting events in the calendar;
4. implement app navigation through a sidebar menu;
5. implement the filtering of events by color;
6. add a function for viewing events in the calendar.

Solution

Task #1: add a calendar component which displays events

Calendar was implemented using the *Android Week View* library [2]. 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 1** and parts of code in **Figure 2** and **Figure 3**.

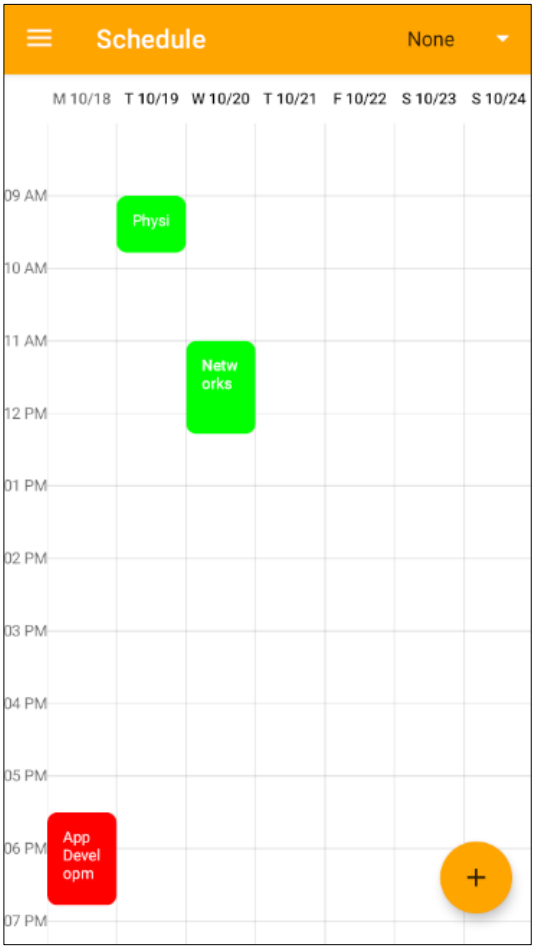

Figure 1. Screenshot of the *Android Week View* component

```
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 2. 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 3. Code of the *ScheduleAdapter*

Task #2: 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 [3], which simplifies manipulation of data. After that, a form with 6 input fields: date, start time, duration, event name, color and location, was added. The form gets validated using the *Kotlin Flow* library [4], 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 1**). The implemented function's UI is displayed in **Figure 4** and parts of code in **Figure 5** and **Figure 6**.

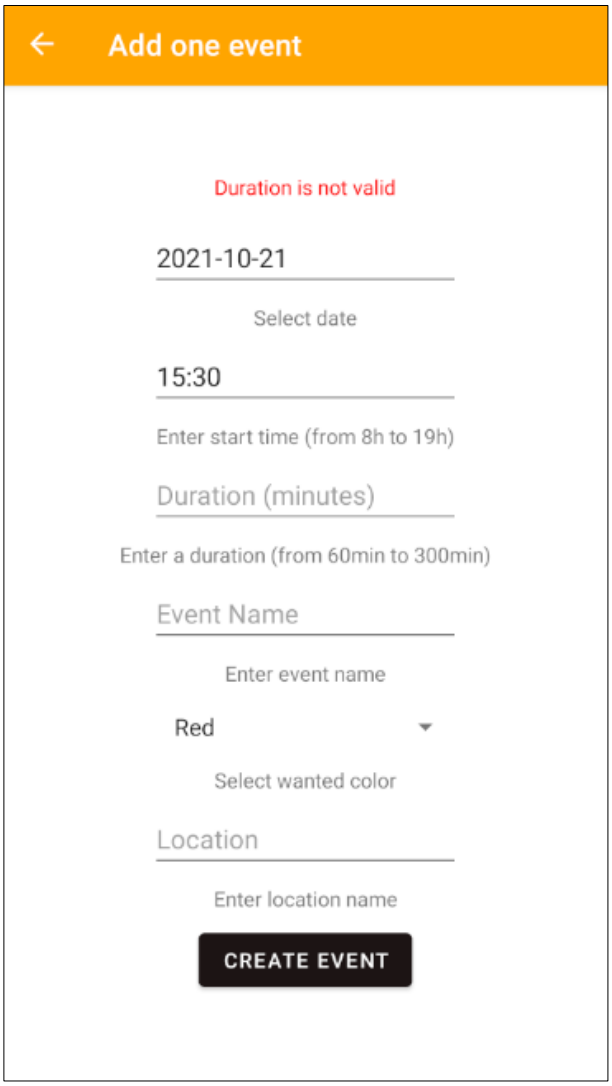

Figure 4. Screenshot of the *CreateEventFragment*

```
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 5. 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 6. Code of the validation function

Task #3: add a function for deleting events in the calendar

A function for deleting events (lectures) was implemented by using the previously mentioned *Android Room* [3] library's database and the *Android Week View* [2] 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 7** and the code in **Figure 8**.

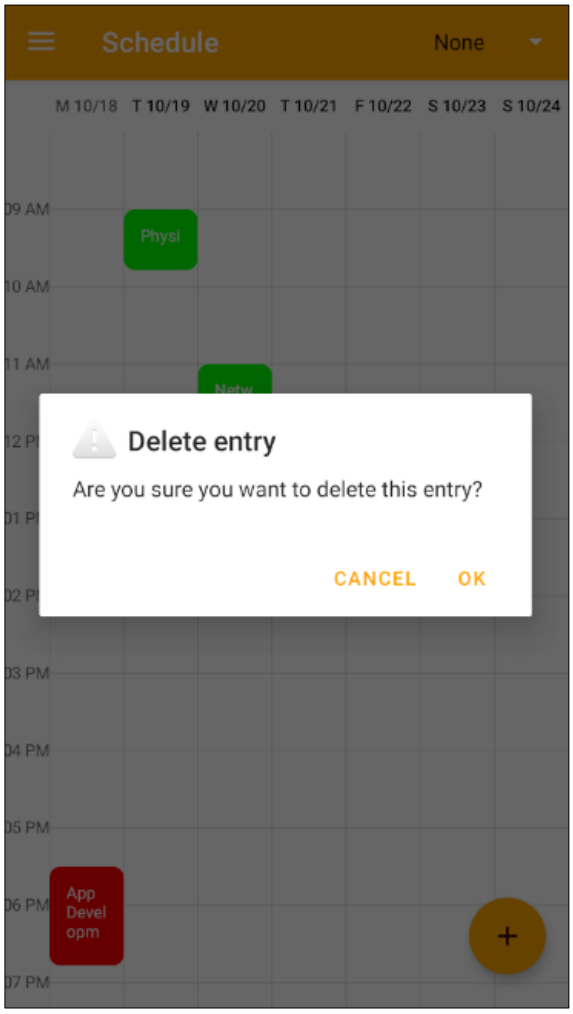

Figure 7. Screenshot of removing events

```
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 8. Code of the function for removing events

Task #4: implement app navigation through a sidebar menu

App navigation was implemented by using the *Android NavigationUI* library [5]. 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 9** and the code in **Figure 10.**

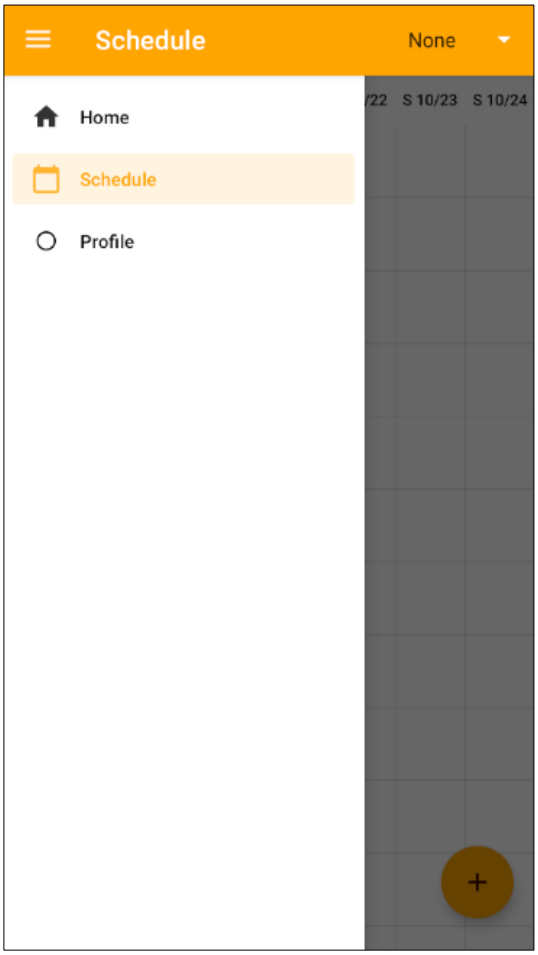

Figure 9. Screenshot of the sidebar menu

```
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 10. Code of the app navigation

Task #5: 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 11** and the code in **Figure 12**, **Figure 13** and **Figure 14**.

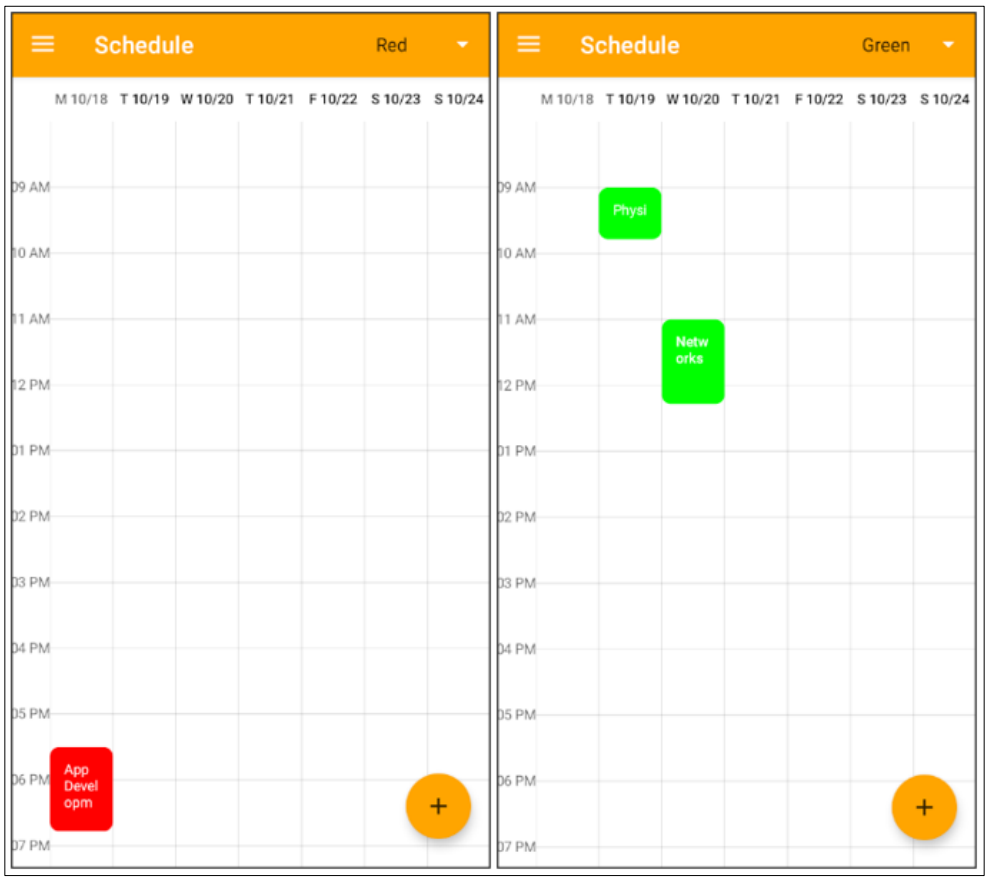

Figure 11. Screenshot of filtering events by color

```
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 12. Code of the filtering of the events

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

Figure 13. 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 14. Code of the *ScheduleFragment* part for event filtering

Task #6: 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 15** and the code snippet in **Figure 16**.

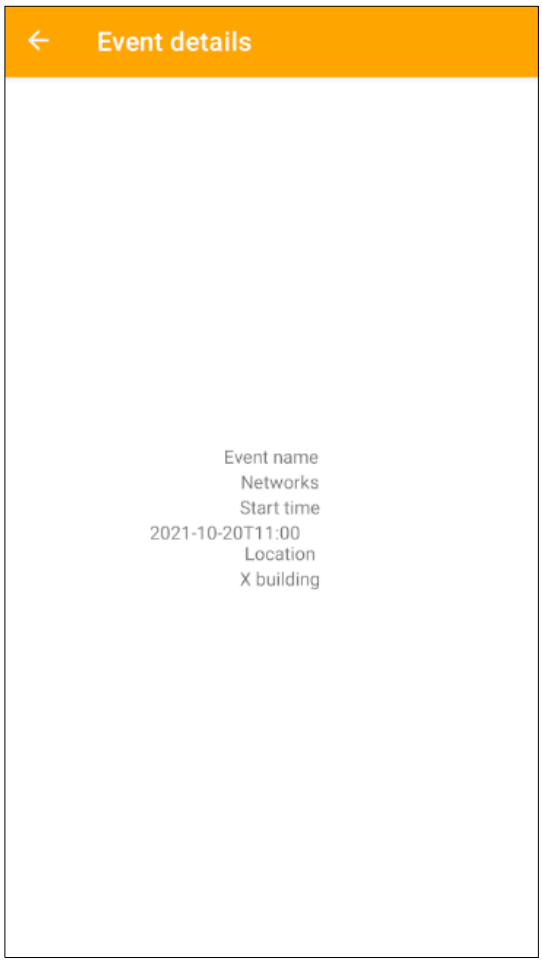

Figure 15. Screenshot of viewing an event

```
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 = args.startTime
    binding.locationText.text = args.location

    binding.lifecycleOwner = viewLifecycleOwner

    return binding.root

}
```

Figure 16. Code of *EventFragment*

Reference list

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

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

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

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

[5] „Navigation,“ [Web]. Available: https://developer.android.com/guide/navigation.