FACULTY OF INFORMATION TECHNOLOGY CTU IN PRAGUE

ASSIGNMENT OF MASTER'S THESIS

Title:	Android app for meeting scheduling
Student:	Bc. David Khol
Supervisor:	Ing. Martin Půlpitel
Study Programme:	Informatics
Study Branch:	Web and Software Engineering
Department:	Department of Software Engineering
Validity:	Until the end of winter semester 2019/20

Instructions

Design and implement a native, mobile app for scheduling meetings. Create an event (duration, place, participants). With a preview of your calendar, select suitable timeslots for the meeting. Send a proposal for the meeting. Recipients will see the meeting invitation and your proposed dates. After the date is confirmed, an event is added to your calendars.

- Look for similar applications, e.g. Sunrise
- Study and use the iCal standard, alternatively other standards that come out of the analysis
- Design an application
- Use the existing backend to address the solution
- Implement synchronization with Google Calendar
- Pay close attention to the design of the user interface
- Program in a reactive manner
- Support for Android 4.4+
- Test the application

References

Will be provided by the supervisor.

Ing. Michal Valenta, Ph.D.
Head of Department

doc. RNDr. Ing. Marcel Jiřina, Ph.D.
Dean

Prague February 23, 2018

Czech Technical University in Prague
Faculty of Information Technology
Department of Software Engineering

Master's thesis

Android app for meeting scheduling

Bc. David Khol

Supervisor: Ing. Martin Půlpitel

9th May 2018

Declaration

I hereby declare that the presented thesis is my own work and that I have cited all sources of information in accordance with the Guideline for adhering to ethical principles when elaborating an academic final thesis.

I acknowledge that my thesis is subject to the rights and obligations stipulated by the Act No. 121/2000 Coll., the Copyright Act, as amended. In accordance with Article 46(6) of the Act, I hereby grant a nonexclusive authorization (license) to utilize this thesis, including any and all computer programs incorporated therein or attached thereto and all corresponding documentation (hereinafter collectively referred to as the "Work"), to any and all persons that wish to utilize the Work. Such persons are entitled to use the Work in any way (including for-profit purposes) that does not detract from its value. This authorization is not limited in terms of time, location and quantity. However, all persons that makes use of the above license shall be obliged to grant a license at least in the same scope as defined above with respect to each and every work that is created (wholly or in part) based on the Work, by modifying the Work, by combining the Work with another work, by including the Work in a collection of works or by adapting the Work (including translation), and at the same time make available the source code of such work at least in a way and scope that are comparable to the way and scope in which the source code of the Work is made available.

In Prague on 9th May 2018 .....

Czech Technical University in Prague
Faculty of Information Technology

Citation of this thesis

Khol, David. *Android app for meeting scheduling.* Master's thesis. Czech Technical University in Prague, Faculty of Information Technology, 2018.

Abstrakt

Práce zkoumá standardy iCalendar a CalDAV, které se používají k reprezentaci událostí a přístupu k plánovacím informacím ze serverů. Uvádí aktuální stav platformy Android a představuje několik knihoven, které značně usnadňují řešení problémů, s nimiž se vývojáři běžně potýkají. Jedná o standardu Reactive Streams, jeho vztah k tématu reaktivního programování a představuje jednu z jeho implementací, knihovnu RxJava.

Analyzuje implementaci stávající webové služby a navrhuje rozšíření její funkčnosti. Dále analyzuje možnosti, jak přehledně vizualizovat události v kalendáři.

Shrnuje závěrečnou podobu aplikace a popisuje, jak se Android verze liší od webové verze. Rozepisuje pokročilý algoritmus pro nalezení optimálního rozložení událostí v kalendáři.

Na závěr hovoří o testování aplikace provedeném s uživateli.

Klíčová slova Android, plánování schůzek, kalendář, reaktivní programování

Abstract

The thesis researches iCalendar and CalDAV standards which are commonly used to represent events and access scheduling information from remote servers. It looks into the current state of Android platform and introduces several libraries that address common issues which Android developers have to deal with. It goes over Reactive Streams standard, its relationship to the topic of reactive programming and introduces one of its implementations, RxJava.

It analyses the implementation of the existing web service and proposes extensions to its functionality. Furthermore, it analyses possible options how to visualize schedules in a form of a calendar view.

It introduces the final implementation of the app and describes how Android version of the app differs from the web version. It introduces an advanced algorithm to display events in a calendar in an uncluttered way.

Finally talks about the testing of the app with users.

Keywords Android, meeting scheduling, calendar, reactive programming

Contents

List of Figures

Introduction

The use of calendaring and scheduling has grown considerably in the last decade. Often times there is a situation when we want to schedule a meeting with someone but it is difficult to settle on a date that suits both parties. We don't know about their schedule and they don't know about ours.

This is especially true for people like managers and business owners. These people usually run on a tight schedule and proper calendaring and scheduling of events is of paramount importance for them.

It is not always possible to get in touch and talk with the other party in person to discuss possible dates for the meeting. As a result, we have to reach the other party via a phone call or by an email.

Phone calls have a disadvantage that we might call in an inappropriate moment when the other party cannot or do not want to pick up the call. Emails, on the other hand, do not have such disadvantage because email communication is asynchronous. After sending an email we don't have to wait for an answer and can proceed with other tasks.

But in either of the situations, we must also refer to our schedule by inspecting our calendar. This can be a difficult task to accomplish on mobile devices.

Making a phone call and inspecting a calendar at the same time is difficult, to say the least. Many people don't even know it is possible to do these two tasks concurrently. Writing an email is not quite user-friendly either. We must switch between calendar and email applications when composing the email.

For that reason we decided to make an application that would help with planning and scheduling of meetings where multiple parties are involved. The app should remove the difficulties of finding a date and time that would fit everyone. It should aid the user in defining available times by showing him an overview of his schedule. Furthermore, it should allow to define other details about the meeting, such as a location or a description, and automatically save arranged meetings into calendars of both parties.

Chapter 1

State-of-the-art

1.1 iCalendar standard

iCalendar is a data format used for representing and exchanging calendaring and scheduling information such as events, to-dos, journal entries, and free/busy information. It is independent of any calendar service or transport protocol. It is designed to only transmit calendar-based data and intentionally does not describe what to do with that data. [1, 2]

The RFC describing the standard is composed of more than hundred of pages and cannot be possibly explained in detail in this thesis. Instead, we will analyze an example to get an idea of what the standard is capable of.

The listing 1.1 shows example contents of a file in the iCalendar format. Structure of the iCalendar format is quite simple and can be read and understood without any external tools as most of the properties are named in plain English. Reading such file can be almost self-explanatory.

Every iCalendar file must start with `BEGIN:VCALENDAR` and must end with `END:VCALENDAR`. Within these two tags, we should first specify a version of the protocol. For iCalendar it is `VERSION:2.0`. `VERSION:1.0` was used to specify an old vCalendar format. Following the version property, we can include multiple components such as:

- `VEVENT` for events,
- `VTODO` for to-do items,
- `VJOURNAL` for journal entries and
- `VTIMEZONE` for time-zone information.

Multiple components of the same type can be repeated.

Each component contains a list of properties. Some of the properties are

```
BEGIN:VCALENDAR
  VERSION:2.0
  BEGIN:VEVENT
    UID:2520
    LOCATION:Ackee s.r.o.
    SUMMARY:Android Meeting
    DESCRIPTION:A recurrent event for discussing the
        latest news from the Android world.
    DTSTAMP:20141210T183838Z
    DTSTART:20141206T100000Z
    DTEND:20141206T110000Z
    ORGANIZER;CN=David Khol:mailto:david.khol@ackee.cz
    RRULE:FREQ=WEEKLY
  END:VEVENT
  BEGIN:VTODO
    UID:132456762153245
    SUMMARY:Do the dishes
    DUE:20180428T115600Z
  END:VTODO
END:VCALENDAR
```

Listing 1.1: An example of an iCalendar object. The indentation was added for the sake of clarity of the example and is not part of the standard.

specific to certain component types while others are available for all types. Furthermore, some properties are required (`SUMMARY`, `UID`, `DTSTART`), others are optional (`DESCRIPTION`, `LOCATION`, `STATUS`) and some even allow to be specified multiple times (`ATTENDEE`). The list includes properties such as:

- `SUMMARY` – defines a short summary or subject for the component
- `DESCRIPTION` – defines a more complete description of the component than that provided by the `SUMMARY` property
- `LOCATION` – defines where the event takes place
- `GEO` – as an alternative to `LOCATION`, we may define latitude and longitude of a location
- `STATUS` – varies depending on a type of the component. For events it can be one of `TENTATIVE`, `CONFIRMED` or `CANCELLED`, while for to-dos it can be one of `NEEDS-ACTION`, `COMPLETED`, `IN-PROCESS` or `CANCELLED`
- `DTSTAMP` – timestamp when the event was created
- `DTSTART` – timestamp when the event is scheduled to start
- `DTEND` – timestamp when the event is scheduled to end

- `DURATION` – we can specify duration of the event instead of `DTEND`
- `COMPLETED` – timestamp when the event actually ended
- `ATTENDEE` – defines information about a single guest of the event and can be repeated in case of multiple guests. It is composed of several subcomponents such as guest's name, participation status of the user, guest's role in the event (required, optional, non-participant) etc.
- `RDATE` – defines a list of date or datetime values for recurring components
- `RRULE` – defines a rule or repeating pattern for recurring components.

There are many more properties available to define events. An overview of all available components and list of their properties can be found in section 8.3 of RFC 5545 [2].

It is not necessary to go into details about every single property as it would be far beyond the scope of this thesis, but one should be aware of such a standard.

iCalendar allows us (with a software that supports the standard) to define and propose an event, but also react and counter-propose to other people's invitations.

1.1.1 CalDAV

iCalendar is used and supported by many services and applications and is utilised by other protocols as well. One of such protocols is *CalDAV*.

CalDAV (Calendaring Extensions to WebDAV, defined in RFC 4791 [3]), is an Internet standard allowing a client to access scheduling information on a remote server. It extends *WebDAV* specification and uses forementioned *iCalendar* format for the data.

CalDAV allows us to access and modify event information on a centralized server. This removes the need to resend a new iCalendar object containing the event details to all participants each time someone accepts the invitation. Many services are capable of syncing the data automatically.

WebDAV is an extension of the *HTTP* protocol. Its aim was to make the web into a readable and writable medium. It provides functionality to create, change and move documents on a remote server. [4]

1.2 Android ecosystem

There is no `main()` function in the Android world that starts an application. When the user starts an application, the framework looks into the metadata of the application and finds an activity that is labeled as the main one. Each android application must have at least one Activity but, depending on the complexity of the application, can have more.

To deal with scarce resources of mobile devices, Android framework uses a concept of lifecycles to control the life span of each Activity. [5]

1.2.1 Lifecycle of Android application

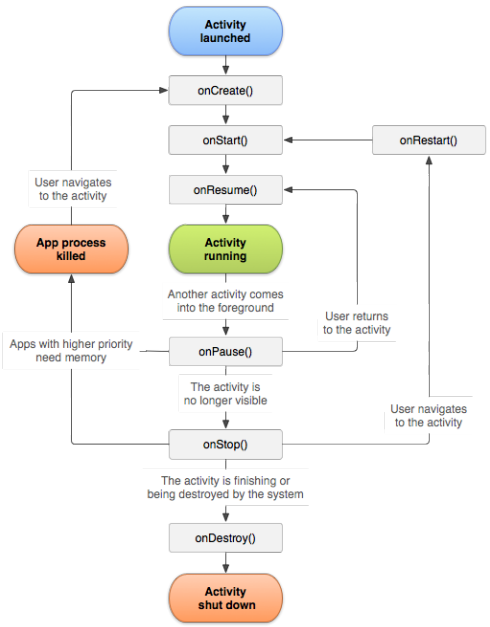


Figure 1.1: Basic lifecycle of an activity

When an Activity is first started, it goes through `Created`, `Started` and `Resumed` states and calls corresponding `onCreate()`, `onStart()` and `onResume()` callback methods. During each of these callbacks, some resources are allocated to be used by the Activity.

Later, when the Activity is finished, it goes through `Paused`, `Stopped` and `Destroyed` states and calls corresponding `onPause()`, `onStop()` and `onDestroy()` callback methods. During each of these callback, some resources are reclaimed by the system. Generally speaking, when something is initialized in `onCreate()` (resp. `onStarted()`, resp. `onResumed()`), its resources should be released in `onDestroy()` (resp. `onStopped()`, resp. `onPaused()`).

But it isn't always this simple. For example when an Activity loses focus it doesn't go through all `Paused`, `Stopped` and `Destroyed` states to get completely destroyed. Instead it just goes to the `Paused` state.

When the application goes to the `Paused` state, the Activity is still partially visible, but the user cannot interact with it anymore. This might be a good time to stop playing a video user has been watching or pause a game the user has been playing.

When the application goes to the `Stopped` state, the Activity is no longer visible as it is completely obstructed by another one. This might be a good time to release handles to graphics engine or stop listening for changes in a database.

When the application goes to the `Destroyed` state, the Activity is about to be terminated and freed from the memory completely. This is the time to close network connections and to dispose of other long-time running actions.

Imagine that a user starts our Activity, let's say a game. Our Activity goes through `Created` and `Started` to `Resumed` state. He plays the game for a while when suddenly a dialog reminder pops up on the screen. At this moment our Activity goes into `Paused` state. The user cannot interact with our game anymore and can only interact with the dialog. If the user dismisses the dialog, our Activity goes into `Resumed` state again and the user can continue playing. On the other hand, if the user interacts with the dialog in a way that it takes him to another Activity, our Activity would go to `Stopped` state. He may resolve the task in a calendar or make a phone call, finally returning to our game. Our Activity would again go through `Started` state to `Resumed` state. After some time the user decides to close the game. At this moment the game goes through `Paused` and `Stopped` and finally to `Destroyed` state, effectively ending the lifecycle of the Activity.

More thorough explanation can be found at [5, 6].

1.2.2 Persisting data across configuration changes

There are situations when an Activity can get destroyed even without an explicit action that would finish the Activity.

- One such situation arises when the device is running low on resources. The system might decide to free up some resources by destroying inactive Activities that were not used for a long time.

  When an Activity is about to get destroyed, we are given a chance to save a state of the *Activity* by overriding `onSaveInstanceState(Bundle)` method and storing volatile data of the *Activity* into the `Bundle`. The bundle is then persistently saved, all references to the *Activity* deleted, running tasks stopped and the memory reclaimed by the system.

When the user returns to the *Activity*, it is recreated by going through `Created`, `Started` and `Resumed` states as if it was its first time being run, only with a difference that a `Bundle`, with the same contents as the `Bundle` we have saved before, is passed as an argument to the `onCreate` method. Then we can retrieve the data and restore the *Activity* to the same state as before.

- Activity undergoes the same process when a *configuration change* occurs. A *Configuration change* happens when device configurations, such as display language, screen density, screen size or screen orientation change. The Activity is destroyed and recreated again so it can load appropriate resources based on the new configuration.

Persisting state this way has one big drawback. There is no way how to persist a network call, database connection, and other unfinished processes. In the following section, we will introduce a solution addressing the problem.

1.2.3 Android architectural patterns

Although it is not required to use any architectural patterns to build Android applications, using one (or even several) can help us to create a modular program that is easier to understand, maintain and also to test. Doing so, we nicely follow Separation of Concerns design principle.

"The most important principle in Software Engineering is the Separation of Concerns (SoC): The idea that a software system must be decomposed into parts that overlap in functionality as little as possible. It is so central that it appears in many different forms in the evolution of all methodologies, programming languages and best practices." [7]

There are plenty of architectural patterns that tackle various requirements of software. Probably the most commonly-known pattern is *Model-View-Controller* (MVC) which was embraced most notably by the web development community. Other architectural patterns include patterns such as *Entity–Component–System* (ECS) that follows the Composition over inheritance principle that allows greater flexibility in defining entities and is mainly used for the game development [8, 9], *Presentation–Abstraction–Control* (PAC) that further extends MVC pattern by connecting components in a hierarchical fashion [10], *Event-bus pattern* that builds upon the publish-subscribe pattern and enables messages to be delivered between components without requiring the components to register themselves to others [11] and many more [12, 13, 14].

Due to the structure and limitations imposed by the design of Android framework itself (e.g. Activities and Fragments, the two main building blocks of Android applications, are instantiated by the framework, not by ourselves) some of the patterns are not feasible to be used to make Android applications. Although it is possible to build an application based upon MVC or Event bus,

it might not be the best fit. In the Android development world, mainly *Model-View-Presenter* (MVP) and *Model-View-ViewModel* (MVVM) patterns arose in popularity.

1.2.3.1 MVC vs MVP vs MVVM

All three patterns separate the application into three main components – a *Model*, a *View* and either a *Controller*, a *Presenter* or a *ViewModel* depending on the pattern. They all share some similarities:

The *View* is typically responsible for displaying a GUI to the user, i.e. text-field, buttons, styles, etc.

The *Model* is typically responsible for storing and retrieving data and domain logic, i.e. domain objects, database interaction, etc.

The *Controller*, the *Presenter* and the *ViewModel* are responsible for separation of the *View* and the *Model* and act as mediators between the two.

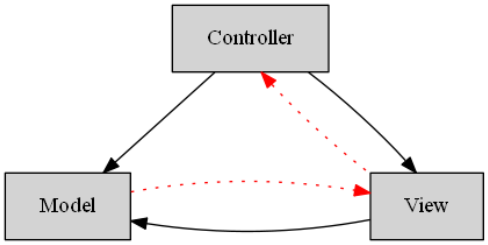


Figure 1.2: MVC pattern.

In the **MVC** pattern, the *View* is responsible for displaying data received from the *Controller* and monitors the *Model* for changes and diplays updated model. There is a *many-to-one* relationship between the *View* and the *Controller* – a single *Controller* may select different *Views* based on the operation being executed. The *View* and the *Model* interact with each other using an *Observer pattern*. The *Controller* is responsible for processing incoming requests from the user. It changes the *Model* and updates the *View*.

The **MVP** pattern is similar to the MVC pattern in which a *Controller* is replaced by a *Presenter*. There is a *one-to-one* relationship between the *View* and the *Presenter* and two-way communication between them. The *View* should not process any data and should contain minimal amount of logic as possible. Its sole purpose is to delegate the user's input to the *Presenter* and display data received from the *Presenter*. The *View* but does not know anything about the *Model*. The *Presenter* retrieves data from the *Model*, transforms them and updates the *View* through an interface.

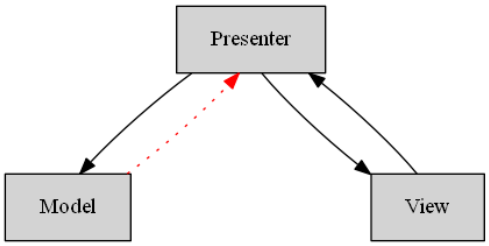


Figure 1.3: MVP pattern.

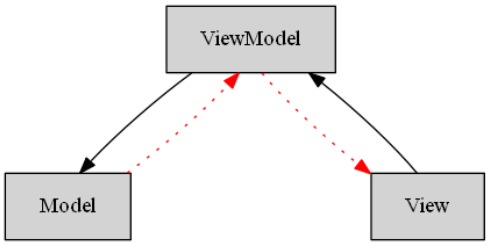


Figure 1.4: MVVM pattern.

The **MVVM** pattern, instead of using a *Controller* or a *Presenter*, has a *ViewModel*. This time there is a *many-to-one* relationship between the *View* and the *ViewModel* – several *Views* can be mapped to a single *ViewModel*. We can either set up a two-way data binding between them or we can prepare observable data in the *ViewModel* that *Views* can subscribe to. Either way, the *ViewModel* does not communicate with *Views* directly, but utilises some form of an *Observable pattern* instead.

Android's *Activities* and *Fragments* are instantiated by the framework and as a result their testing is generally more involved than testing of other classes. For that reason they should be kept as simple as possible. In both *MVP* and *MVVM* patterns, *Activities* and *Fragments* represent *View* components.

1.2.3.2 Repository pattern

Repository pattern adds an extra layer of indirection to our *MVVM* (resp. *MVP*) architecture by adding a `Repository` between the `ViewModel` (resp. `Presenter`) and the `Model`, as depicted in the figure 1.5.

Imagine an application with a **login screen** where the user must either register or log in before accessing application's contents, a **contents screen** where he can browse the contents and a **profile screen** where he can view and edit information about his account profile.

The **login screen** (resp. **profile screen**) would be composed of a LoginActivity (resp. ProfileActivity) as the *View*, a LoginViewModel (resp. Pro-

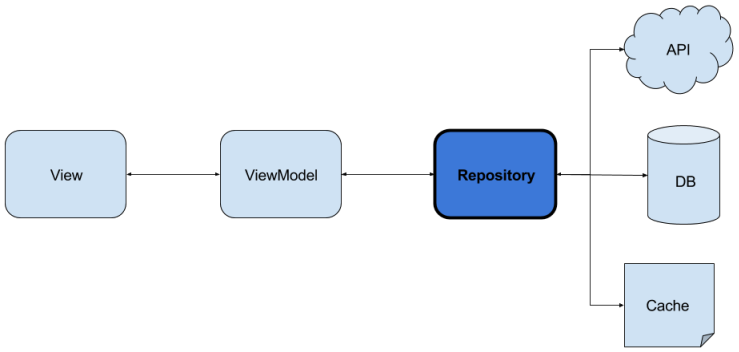


Figure 1.5: Repository pattern schema for MVVM.

fileViewModel) as the *ViewModel*, and backend server and local database as the *Model*.

In the standard MVVM implementation the *LoginViewModel* would perform a network call and after receiving the response from the server, it would save the information about the logged in user into the local database.

On the **profile screen** we display information about the user that we retrieved from the local database. When the user updates his profile, we perform a network call to update the information on the server and store the information back into the local database.

Instead of accessing the Model layer directly (performing network calls and storing the information into local database) in the *LoginViewModel*, we may delegate this logic to a Repository. In the ViewModel we just tell the repository to log the user in.

This works great on Android platform because repositories are not tied to a lifecycle of a single activity. When a configuration change occurs, *Activities* get recreated, but *Repositories* survive. This way we don't have to restart network calls or perform the same call from multiple ViewModels. We can just perform the call once and observe the result in both ViewModels.

1.2.4 Android libraries

Building an application using only the API provided by the Android framework can be a frustrating and tiresome task sometimes. Luckily there are tons of libraries that facilitate the development.

In the following subsections, we will describe core libraries used in the implementation part of the thesis that alter the structure of the code. We will not go into details about libraries that are used by virtually every application (such as Android Support libraries) or libraries that only add some non-critical utilities (such as error reporting after an app crash) or libraries that are used

just to add more functionality to other libraries (such as serializing adapters for Retrofit).

1.2.4.1 RxJava

RxJava allows us to apply reactive programming paradigm to our code and react to and propagate changes between components of the app. It is used in a majority of classes throughout the implementation part and integrates well with many other libraries, including libraries introduced below.

How does RxJava work and what is its relationship to reactive programming is a very deep topic and is thoroughly explained in a separate section 1.3. Those who are not familiar with reactive programming paradigm might want to read it first in order to fully understand how RxJava interacts with other libraries.

1.2.4.2 Retrofit

Before explaining how *Retrofit* works, we should describe how one performs a network call using plain Android SDK. It is not a simple task. Since Android 3.0 we have to perform network operations on a thread other than the main UI thread. Not doing so results in a `NetworkOnMainThreadException`. In plain Android code, we would use `AsyncTask`s to offload network calls onto another thread. We would use `HttpsUrlConnection` to handle a network connection and notify the user about the progress through the `AsyncTask`. After retrieving all data from the network, we must then convert received `InputStream` into a more usable format such as a `String` or a custom domain-specific model. On top of that, we should also handle cases when our activity is destroyed and recreated due to a configuration change. We could cancel the request and restart it after activity recreation or just put the network related logic into a separate `Headless fragment` that survives configuration changes. By implementing one simple network call this way, we might end up writing hundreds of lines of code. More concrete description of the process with code examples can be found at [15].

Thankfully, there are libraries that solve a lot of mentioned problems for us. Namely, *Retrofit* helps us to decouple network calls from the rest of the application by defining a simple interface. *Retrofit* is a type-safe HTTP client for Android and Java. [16] We don't have to deal with `HttpUrlConnection` and transform received `InputStream` data ourselves.

Listing 1.2 presents an example of a simple interface defining two network calls. One creates a meeting and the other retrieves a meeting from a server. Retrofit uses *REST* (Representational State Transfer) as means of communication with a server.

We must annotate each interface method with either `@GET`, `@POST`, `@PUT`, `@DELETE`, `@PATCH`, `@HEAD` or `@OPTIONS` to define which *REST* method should be

```
interface ApiDescription {

  @POST("meetings")
  fun createMeeting(@Body meeting: Meeting): Completable

  @GET("meetings/{meetingId}")
  fun meetingDetail(@Path("meetingId") meetingId: Int):
      Single<Meeting>

}
```

Listing 1.2: Retrofit interface.

used to perform the network call and an address where the request should be sent to. Depending on the *REST* method used, we can add annotated parameters to methods' signatures:

- `@Path` specifies variable parts of the request's url
- `@Body` defines a body of a *POST* method
- `@Query` adds an optional parameter after '?' sign
- `@Header` adds a HTTP header to the request
- `@Url` defines a custom dynamic url for the request
- `@Field` adds the parameter to a request body

By default, Retrofit can only deserialize HTTP bodies into a `ResponseBody` type and it can only accept a `RequestBody` type for `@Body`-annotated parameters. We can add converters to add support for custom types as well. Moreover we can also add custom adapters. One such adapter allows us to use RxJava's observable types as return types of each method.

Listing 1.3 shows the code for performing a network call using combination of *Retrofit* and *RxJava*. Note that the whole setup required less than 20 lines of code that is much more simple than a code we would have to write using `AsyncTask`s and `HttpsUrlConnection`s.

1.2.4.3 Anko

Many of resource files in Android are written in XML, including, but not limited to, styles, strings, menu contents, vector drawables, colors, Android manifest, etc.

UI in Android is usually defined in XML as well. When we want to display a UI for our application, we can pass an XML file to a `Layout Inflater` which inflates all views specified in the file and creates a view hierarchy for us. We can

```
val api: ApiDescription = Retrofit.Builder()
    .baseUrl("https://planner.ack.ee")
    .addConverterFactory(MoshiConverterFactory.create())
    .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
    .build()
    .create(ApiDescription::class.java)

api.meetingDetail(378, "xeKdIef")
    .subscribeOn(Schedulers.io())
    .observeOn(AndroidSchedulers.mainThread())
    .subscribe { meeting: Meeting ->
      // do something with the meeting
    }
```

Listing 1.3: Performing a network call using retrofit and RxJava.

then access views in the hierarchy and set their attributes programmatically. 1.5 shows an example of a primitive layout written using XML.

```
<LinearLayout
  android:orientation="vertical"
  android:layout_width="match_parent"
  android:layout_height="match_parent"
  >
  <EditText
    android:id="@+id/name"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    />
  <Button
    android:text="Say hello"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:onClick="onButtonClick"
    />
</LinearLayout>

// in a code where we use the xml above
lateinit var name: EditText = findViewById(R.id.name)

fun onButtonClick(v: View) {
    Toast.makeText(act, "Hello, ${name.text}!",
        Toast.LENGTH_SHORT).show()
}
```

Listing 1.4: Layout defined in XML file.

There are some drawbacks using XML approach:

- No type safety. When we access a view in the hierarchy, we must cast it to correct type by ourselves. We will not be notified about wrong

typecasts until the runtime.

- We cannot put any custom logic into XML, even if it is just a logic for just showing and hiding of content.
- Minimal code reusability.
- XML must be parsed before displaying its contents, creating an unnecessary overhead.

Although it is possible to create UI programmatically, it is not a common practice, mainly because it looks chaotic and is hard to maintain. Listing 1.5 defines the same layout as in the previous example but is written using pure Kotlin.

```
val layout = LinearLayout(ctx)
layout.orientation = LinearLayout.VERTICAL
val name = EditText(ctx)
val button = Button(ctx)
button.text = "Say Hello"
button.setOnClickListener {
    Toast.makeText(ctx, "Hello, ${name.text}!",
        Toast.LENGTH_SHORT).show()
}
layout.addView(name)
layout.addView(button)
```

Listing 1.5: Layout built programmatically

Thanks to Kotlin's extension functions, we have an ability to define custom DSLs (domain-specific languages). Anko defines a DSL for writing UI layouts using Kotlin in a concise way without a need for parsing of XML files. Moreover, since it is written in Kotlin, we can put custom UI-related logic directly into layouts. 1.6 shows Anko equivalent of the examples mentioned before.

```
verticalLayout {
    val name = editText()
    button("Say Hello") {
        onClick { toast("Hello, ${name.text}!") }
    }
}
```

Listing 1.6: Layout defined using Anko

1.2.4.4 Android Architecture Components

Android Architecture Components [17] library is relatively new compared to other mentioned libraries. There are several components in the library:

LiveData is a lifecycle-aware observable data holder class.

ViewModel stores and manages UI-related data in a lifecycle conscious way.

Room provides an abstraction layer over SQLite.

Paging helps to gradually load information as needed from a data source.

LifeCycle helps to perform actions in response to a change in the lifecycle status of another component.

As mentioned in the previous section about *MVVM* 1.2.3.1, *ViewModel* uses *Observable pattern* to notify other components about its changes. *LiveData* implements the pattern specifically tailored for *Android* because it is aware of the lifecycle state of its observed view. Although these two Components work well together, we don't need *LiveData* component because we are using RxJava already.

In addition to the *ViewModel* component, we will also use *Room* component. *Room* provides an abstraction layer over *SQLite* that is included in the Android SDK. *SQLite* is a relatively small relational database that, unlike many other database systems, does not use client-server architecture but is embedded into a program directly. [18] Room allows us to abstract away from low-level SQL queries if we want to, but still supports them if needed. *Room* removes a lot of boilerplate required to convert SQL queries to data objects, gives us a compile-time checking of SQL queries and adds an option to reactively observe changes in the database. [19, 20]

1.3 Reactive programming

Reactive programming is about the processing of asynchronous streams of data items, where applications react to the data items as they occur. A stream of data is essentially a sequence of data items occurring over time though. This model is more memory efficient because the data are processed as streams, as compared to iterating over the in-memory data.

In the Reactive Programming model, there is a Publisher and a Subscriber. The Publisher publishes a stream of data, to which the Subscriber is asynchronously subscribed.

The model also provides a mechanism to introduce higher-order functions to operate on the stream by means of Processors. Processors transform the data stream without any need for changing the Publisher or the Subscriber. The Processor (or a chain of Processors) sit between the Publisher and the

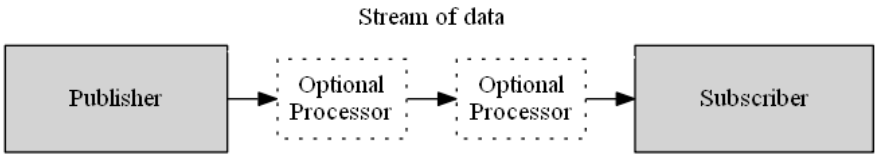


Figure 1.6: Publisher-Subscriber-Processor relation.

Subscriber to transform one stream of data to another. The Publisher and the Subscriber are independent of the transformation that happens to the stream of data. [21]

1.3.1 Reactive Streams

We should note that there is a difference between the terms *Reactive programming* and *Reactive Streams*. *Reactive programming* is a programming paradigm, just like object-oriented programming, procedural programming or functional programming. Simply said, it is programming with asynchronous data streams.

On the other hand, *Reactive Streams* is a specification. It gives us a common API for *Reactive programming*. One does not have to follow *Reactive Streams* specification to do reactive programming, although it is becoming a widely-accepted standard (it has also been added to Java 9 as Flow API).

> "*Reactive Streams* is an initiative to provide a standard for asynchronous stream processing with non-blocking back pressure." [22]

There is a multitude of implementations of Reactive Streams [23]. Some of implementations include Akka Streams [24], Reactor [25], RxJava [26], MongoDB [27] or Spring [28].

In this thesis we will talk about and use primarily *RxJava* because it has been widely accepted as the reactive library amongst Android developers. RxJava is a part of a group called *Reactive Extensions*[29], also known as *ReactiveX* or just *RX*.

There are many other implementations [30] of *Reactive Extensions* for many popular programming languages such as `C++`, `C#`, `Python`, `PHP`, `Javascript`, `Swift` etc. If you wanted to use another language than `Java`, most of the logic and operators used would remain the same and the only difference would be the syntax.

1.3.2 Reactive Streams API

The API consists of the following four components that are required to be provided by Reactive Streams implementations:

```
interface Publisher<T> {
  fun subscribe(s: Subscriber<in T>)
}

interface Subscriber<T> {
  fun onSubscribe(s: Subscription)
  fun onNext(t: T)
  fun onError(t: Throwable)
  fun onComplete()
}

interface Subscription {
  fun request(n: Long)
  fun cancel()
}

interface Processor<T, R> : Subscriber<T>, Publisher<R>
```

Listing 1.7: Reactive Streams interfaces

Upon first glance these interfaces may look deceptively simple. But to be Reactive Streams specification compliant, any implementation of the interfaces must follow list of about 40 rules. [31]

Figure 1.7: Publisher-Subscriber-Processor relation with backpressure.

A *Publisher* is a producer of a potentially unlimited number of items, publishing them according to the demand received from its *Subscriber*(s).

A *Subscriber* is a consumer of items produced by *Publisher*. To start receiving items from the *Publisher*, it must first subscribe to it via a call to `Publisher.subscribe(Subscriber)`.

Upon the subscription the *Publisher* creates and passes a new *Subscription* to the *Subscriber* via a call to `Subscriber.onSubscribe(Subscription)`. Through this *Subscription* the *Subscriber* may then request items from the *Publisher* via calls to `Subscription.request(Long)`.

The *Publisher* may starts emitting items into the stream, according to the demand from its *Subscriber*, using `Subscriber.onNext(T)`. When the *Publisher* is out of items to emit, it must signal `Subscriber.onComplete()`. If an error occured during production of items, the *Publisher* must signal `Subscriber.onError(Throwable)`. If the *Subscriber* calls `Subscription.cancel()`, the subscription should be considered cancelled and *Publisher* should stop signalling the *Subscriber*.

A *Processor* takes on a role of both *Publisher* and *Subscriber*. As men-

tioned before, *Processor*s transform the data stream without the need for changing the *Publisher* or the *Subscriber*. All the *Subscription* logic remains the same, but *Processor* is on the both ends of the communication. It keeps one *Subscription* for its *Publisher* it is receiving items from and another *Subscription*(s) for its *Subscriber*(s) it is emitting items to.

It should be noted that multiple *Subscriber*s may be subscribed to a single *Publisher*. In such case each *Subscriber* receives its own *Subscription* that mediates the communication between the two.

1.3.3 Backpressure

Imagine a situation where we didn't have a *Subscription* through which *Subscriber*s would request more items. *Publisher*s would just push their data into *Subscriber*s when they would want.

When *Publishers* are producing at a much faster rate than the rate at which the data items are consumed by the *Subscriber*s, *Subscriber*s must keep a buffer where they would store unprocessed items and remove them as they are being processed. The size of the buffer where the unprocessed items are being buffered might be, and usually is, limited. If such system would run for an extended amount of time, this buffer could grow and the system would eventually run out of memory leading to a crash.

Introducing a *Subscription* to mediate the communication between *Publisher* and *Subscriber* solves this problem by never allowing more items to be emitted than a *Subscriber* can buffer.

The *Reactive Streams* API does not provide any strategies to deal with backpressure, but one could implement them by oneself. There are several strategies how to deal with backpressure:

- Error – throw a `MissingBackpressureException` when the buffer overflows, effectively terminating the stream.
- Buffer – we can buffer items as long as we have enough memory. When we exhaust all available memory `OutOfMemoryException` is thrown.
- Drop – Uses a fixed 1-item sized buffer. When more items are received, drop them until the buffered item is removed from the buffer and processed.
- Latest – Uses a fixed 1-item sized buffer. Add the item to the buffer (if empty) or replace the item currently in the buffer when more items are received.

Instead of dealing with backpressure with strategies mentioned above, we can resolve the issue using combination of one or more *operator*s, such as `debounce`, `throttle` or `sample`. *Operator*s are explained in the next section.

More thorough examples about dealing with backpressure can be found at [32].

1.3.4 RxJava

Cardinality doesn't matter in the Reactive Streams specifications. A stream can contain a single element, ten elements or no elements at all. In RxJava we have few extra options that are not part of the Reactive Streams specifications, but depending on the situation, they might be more expressive. See figure 1.8 for overview of RxJava types.

Type	Protocol
Observable	`onSubscribe onNext* (onComplete | onError)?`
Flowable	`onSubscribe onNext* (onComplete | onError)?`
Single	`onSubscribe (onSuccess | onError)?`
Completable	`onSubscribe (onComplete | onError)?`
Maybe	`onSubscribe (onSuccess | onComplete | onError)?`

Figure 1.8: Five basic types in RxJava

Observable and *Flowable* are the same from the protocol viewpoint as both of them can contain any number of elements but they work differently internally. *Flowable* is the implementation of Reactive streams specification, with asynchrony, backpressure and all that entails. *Observable* is a lightweight counterpart of *Flowable* that is non-backpressured. You can think of it as a *Flowable* that requested unlimited items. *Observable* has lower overhead than *Flowable*. In situations where backpressure is not a problem, it generally performs better.

Single, *Completable* and *Maybe* are *Observables* that have a limitation imposed on the number of items they can emit. *Single* must complete with exactly one item emitted otherwise signal an error. That is why *Single* does not have `onNext` handler because the item is delivered as a part of `onSuccess` handler. *Completable* does not emit any items at all, it just signals if it was completed successfully or with an error. *Maybe* is conceptually a union of *Single* and *Completable*. It can complete without emitting any items, emit one item or signal an error. All of *Single*, *Completable* and *Maybe* are intrinsically non-backpressured, because they may contain at most 1 element.

In many situations, we know how many items the publisher will emit and possible time intervals between each emission. Based on this information, it might be better to use more constrained type than Flowable. Not because we don't have to care about backpressure (and make our code tiny bit faster), but because of the semantics. Later when we combine different publishers together, it might help us decide which operators to use.

When we observe a sensor in our phone such as gyroscope or monitor movement of a finger on a screen, we don't know how many and how fast items will be emitted. There might be so much data that we might not be able to process every single item as fast as we are receiving them, especially if we are doing some computation-intensive tasks on every item. This is a good candidate for *Flowable*.

When we observe clicks of a button, we still don't know how many items will be emitted, but this time we are more confident about the intensity of emissions. Do we need backpressure in this case? In most situations no, but whether *Flowable* is more appropriate depends on whether we can keep up. This is a good candidate for *Observable*.

When we do an API call to get details about a user, we know that there will be exactly one item in the stream. But don't get confused. Just because the user we requested didn't exist or because the server rejected our authorization or because we don't have internet access, it doesn't mean there can be 0 elements. In all these situations, an Error signal should be delivered instead. This is a good candidate for *Single*.

When we do an API call to update information about a user, we don't expect any items in the stream. We just want to know if the update was successful or not, in such case we would get an exception through onError signal. This is a good candidate for *Completable*.

When we want to retrieve user's avatar picture from the database, we might get the bitmap data, when the picture exists, or nothing, when it doesn't exist, what is perfectly fine. User might not set his profile picture and thus returning nothing is a valid option too. This is a good candidate for *Maybe*.

1.3.5 Operators

If you have some experience with functional programming such as using Java Streams introduced in Java 8 or even just functional extensions on collections in Kotlin, you will have a much easier time understanding RxJava operators. They use very similar naming and syntax to chain operators together. It is not uncommon to be able to change the receiving type and get the same results.

In code example 1.8 we create a list, a stream and an observable, all of them initialized with the same list of names. Then all three receivers are run through chain of operators, each of them having different implementations. If you would run this code, you would get the same output for each receiver `forEach()` call.

```
val names = listOf("David Khol", "Martin
    Pulpitel", "Dominik Vesely", "Marek Janca",
    "David Bilik")

val list: List<String> = names
val stream: Stream<String> = names.stream()
val observable: Observable<String> =
    Observable.fromIterable(names)

list    .map { it.split(" ")[0] }
        .filter { it.startsWith("D") }
        .sorted()
        .distinct()
        .forEach { println(it) }

stream  .map { it.split(" ")[0] }
        .filter { it.startsWith("D") }
        .sorted()
        .distinct()
        .forEach { println(it) }

observable.map { it.split(" ")[0] }
        .filter { it.startsWith("D") }
        .sorted()
        .distinct()
        .forEach { println(it) }
```

Listing (1.8) Comparison of Kotlin's functional extensions, Java Streams and RxJava Observables.

```
David
Dominik
David
Dominik
David
Dominik
```

Listing (1.9) Output

1.3.6 Operator types

There are many built-in operators at our disposal in RxJava. These operators may help us to:

- create a new *Observable* (create, defer, empty, just, timer, etc.),
- transform items emitted by an *Observable* (map, flatMap, buffer, groupBy, etc.),
- filter items emitted by an *Observable* (filter, debounce, first, sample, etc.),
- combine multiple *Observable*s (zip, combineLatest, merge, etc.),
- handle and resolve error signals (catch, retry),
- do conditional operations (all, contains, skipUntil, etc.),
- do arithmetical operations (average, count, max, sum, etc.),
- do aggregating operations (concat, reduce),

- deal with backpressure (buffer, sample, debounce, window),
- split an observable for use by multiple *Subscriber*s (connect, publish, replay, etc.),
- convert an *Observable* to another type (toSingle, toMaybe, toObservable) and
- other operations (subscribe, subscribeOn, observeOn, delay, timeout, etc.)

A full list of RxJava's operators can be found at [33].

1.3.7 Operator representation

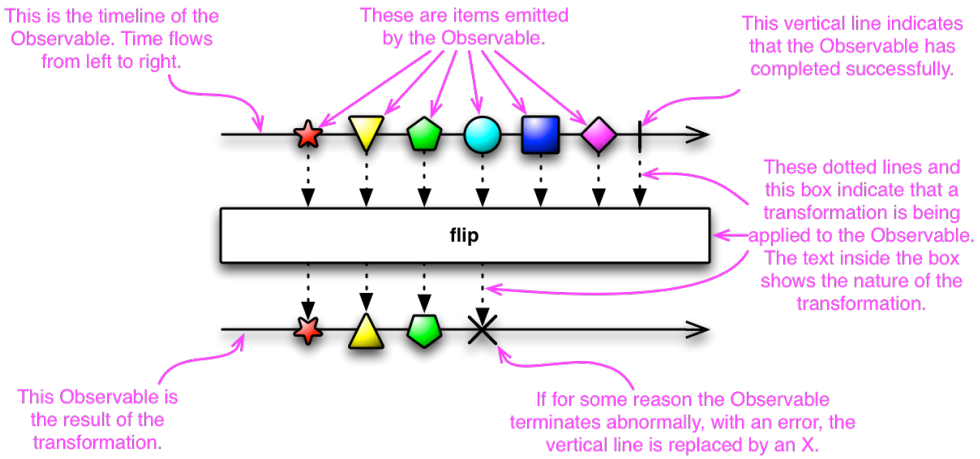


Figure 1.10: Marble diagram explanation.

Operators are often visualised using *Marble Diagrams*. In the figure 1.10 is an example of Marble Diagram describing a fictional operator *flip*. In every Marble Diagram, the time flows from the left to the right. On the top we have a horizontal line(s) representing an input Observable(s) and its (their) emissions. In the middle we have a box describing function of the operator. On the bottom we have a horizontal line representing an output Observable and its emissions. Colourful pictures on a horizontal line represent *onNext* signals, a vertical line represents *onComplete* signals and a cross represents an *onError* signal.

The *Map* operator 1.11 applies a custom transformation to each item received from the source *Observable* and returns an *Observable* that emits the results of the transformation.

The *FlatMap* operator 1.12 applies a custom transformation to each item received from the source *Observable*, where that transformation returns an

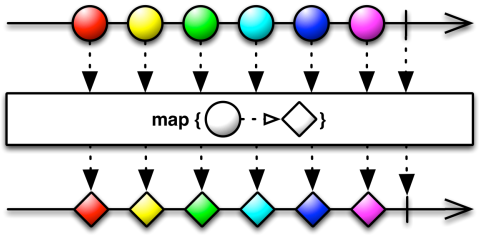


Figure 1.11: The *Map* operator.

Observable that itself emits items. *FlatMap* then merges the emissions of these resulting *Observable*s, emitting these merged results as its own sequence. Note that FlatMap merges the emissions of these *Observables*, so that they may interleave.

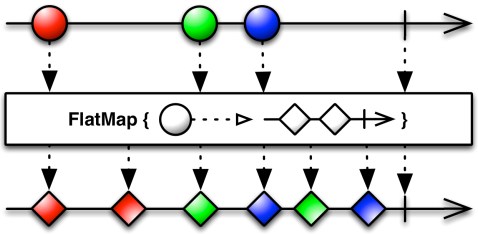


Figure 1.12: The *FlatMap* operator.

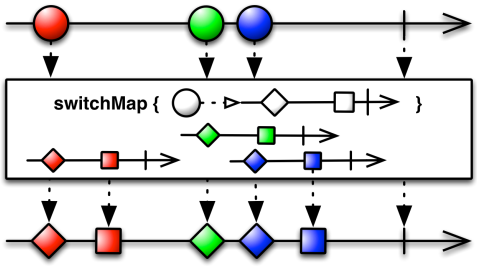


Figure 1.13: The *SwitchMap* operator.

The *SwitchMap* operator 1.13 works similarly as the *FlatMap* operator and under certain circumstances it even emits the same items, given the same input. The difference is that once a new item is received from the source *Observable*, all previous inner *Observable*s' emission are ignored, meaning that at any given point only one inner *Observable* is emitting items. It produces the same emissions as *FlatMap* when inner *Observable*s don't overlap with their emissions.

Imagine a user searching for some information on the internet. As he types in his query, we might want to give him suggestions based on the current query. With every change to the query (with every character he types) a network call to retrieve suggestions is initiated, producing an *Observable*. But as he writes more, these suggestions we just have requested become out-of-date. That itself wouldn't be such a big problem, because we initiated another request to retrieve up-to-date suggestions, right? No. The real problem is that these out-of-date suggestions may arrive later than the up-to-date suggestions due to network latency or any other reason. *SwitchMap* automatically unsubscribes from those out-of-date suggestions when a new query is received, giving a chance only to the newest suggestions.

The *Debounce* operator 1.14 takes a time duration as an argument. When

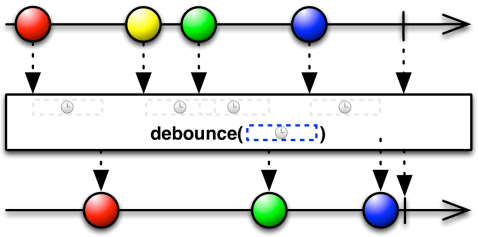


Figure 1.14: The *Debounce* operator.

an item is received from the source *Observable*, it starts a timer. If during this period no other item is received, the last received item is emitted. On the other hand, if another item is received during this period, the timer restarts and again awaits whether another item arrives or not. This is one of ways how to deal with backpressure without actually introducing any backpressure strategies.

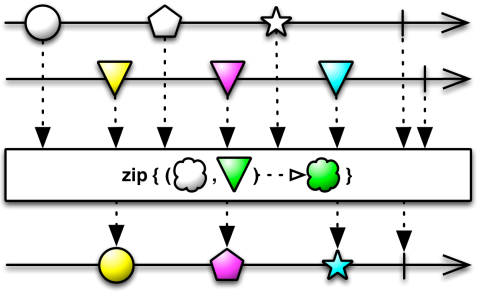


Figure 1.15: *Zip* operator.

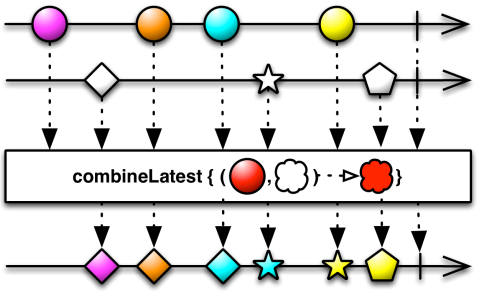


Figure 1.16: *CombineLatest* operator.

Zip 1.15 and *CombineLatest* 1.16 operators operate on two *Observables* at once. Whenever *Zip* receives an emission from an input Observable, it stores that item into a buffer associated with that *Observable*. Whenever both buffers contains more than zero items, the *Zip* operator takes one item from both buffers, combines them using provided transformation function and emits the result; repeating until one of the buffers is not empty.

Whenever *CombineLatest* receives an emission from an input Observable, it notes which Observable emitted the value and also stores the value. Every time a new value is received, latest emission from each input Observable is combined into a single item using provided transformation function and the result is emitted.

The *Zip* operator is useful when we need to combine two streams of items that have a one-to-one mapping and a strict order of their emissions. The *CombineLatest* operator on the other hand, is useful when we need to combine two streams of items that does not have a one-to-one mapping and a strict order of their emissions.

1.3.8 Using operators

Let's analyze a more complicated example. Let's say we have a list of names and want to print surnames of people whose first name starts with letter D. We also want to group these surnames by first names of those people.

```
list
    .filter {
        it.startsWith("D")
    }
    .map {
        it.split(" ")
    }
    .groupBy({
        it[0]
    }, {
        it[1]
    })
    .map {
        "${it.key}:
            ${it.value.joinToString()}"
    }
    .forEach { println(it) }
```

Listing (1.10) Kotlin's functional extensions

```
> filter
> filter
> filter
> filter
> filter
> map
> map
> map
> groupBy
> groupBy
> groupBy
> map
> map
David: Khol, Bilik
Dominik: Vesely
```

Listing (1.11) Output

```
observable
    .filter {
        it.startsWith("D")
    }
    .map {
        it.split(" ")
    }
    .groupBy({
        it[0]
    }, {
        it[1]
    })
    .flatMapSingle { group ->
        group.toList().map {
            group.key to it
        }
    }
    .map {
        "${it.first}:
            ${it.second.joinToString()}"
    }
    .forEach { println(it) }
```

Listing (1.12) RxJava's Observable

```
> filter
> map
> groupBy
> flatMapSingle
> filter
> filter
> map
> groupBy
> flatMapSingle
> filter
> filter
> map
> groupBy
> nested map
> map
David: Khol, Bilik
> nested map
> map
Dominik: Vesely
```

Listing (1.13) Output

When we run Kotlin example 1.11 and RxJava example 1.12, in both cases the final `forEach()` calls print identical output, as expected. But when we try to debug the flow (by adding a `println()` call inside of each operator), we get interesting outputs.
Kotlin's operators get called one after each other in the order we defined them.

1. `filter` gets called five times for each person, filtering out 2 people whose name doesn't start with 'D'.

2. `map` gets called three times for each person starting with 'D' and splits people's names into arrays of two items – first name and surname.

3. `groupBy` gets called three time for each person starting with 'D' and groups people based on their first name and keeps their surname.

4. `map` gets called twice for each unique first name starting with 'D' and joins the first name and surnames back together.

5. `forEach` prints surnames for each unique first name starting with 'D'.

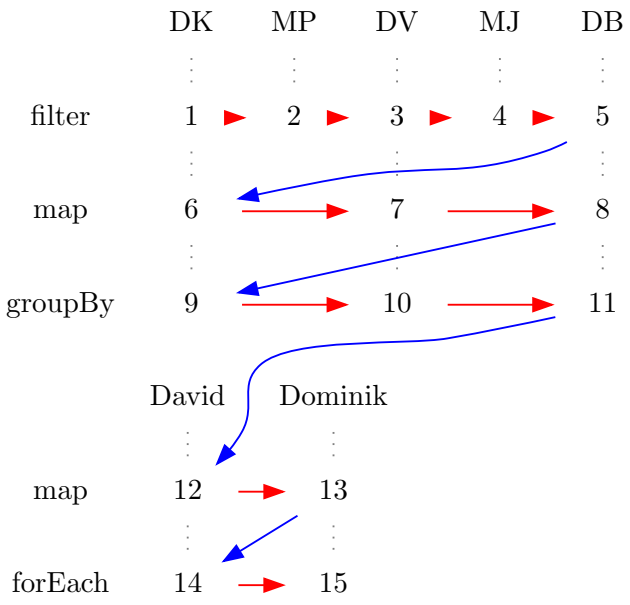


Figure 1.19: Items before operators.

See figure 1.19 for a graphical representation. Red lines represent transition to the next item while blue lines represent transition to the next operator. We always process all items before moving on to the next operator.

But when we look at RxJava's output, operators get called in seemingly random order. What is going on here?

Only one item is emitted into the stream at a time. The first person goes through the `filter`, `map`, `groupBy` and finally ends up being caught by the `flatMap` operator. Then the next person is emitted, but is filtered out immediately by `filter`. Then the next person is emitted and goes through the previously mentioned operators again, and so on.

When there are no more items to be emitted into the stream, a `Complete` signal is emitted and propagated through `filter`, `map` and `groupBy` to `flatMap` operator.

The reason why not all operators were processed immediately is because of the `toList()` operator. `toList()` does not emit any items until it receives a Complete signal. At that point, it sends a single item – a list of items it has accumulated during its lifetime and completes.

When `flatMap` receives a `Complete` signal, it distributes the signal to all groups (each group is actually a standalone Observable) it has created. These groups result into two emissions – one for 'David' and one for 'Dominik'. Stream then continues to `map` these two items and print them to a output.

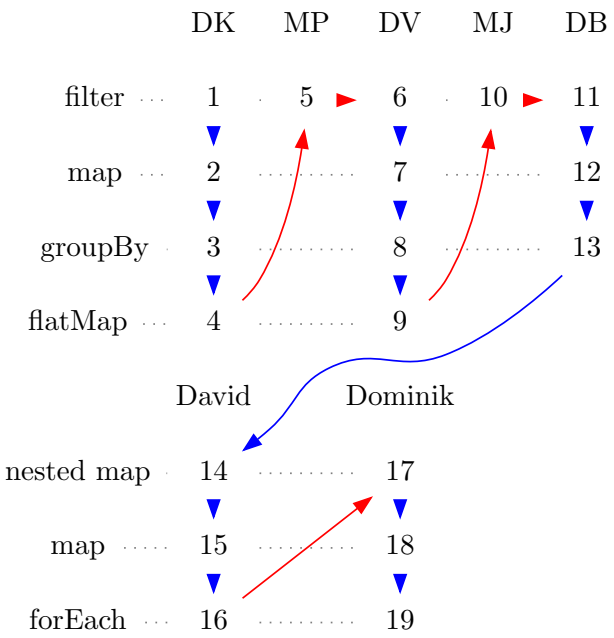


Figure 1.20: Operators before items.

See figure 1.20 for a graphical representation. Same as before, red lines represent transition to the next item while blue lines represent transition to

the next operator. We pass an item through all operators until it gets caught by some blocking operator before moving on to the next item.

1.3.8.1 Subscription

If we removed the `forEach` call from both examples 1.11 and 1.12, what would happen? Kotlin version would print everything up to the last `> map`, ignoring two final lines. RxJava version would not print anyting at all. Why? Because inside of the Observable's `forEach` call is actually hidden `subscribe()` call that is needed to start the observable stream.

We already explained the subscription logic in a previous section 1.3.2. An *Observable* must not emit any items until it receives a `Subscription.request(N)` call from a *Subscriber*. But there will not be any requests without a *Subscriber*.

What happens when we call `subscribe()` on our observable stream is that the *Subscriber* subscribes to its *Observable* and 'starts' it up. In our case there are several *Processor*s, each acting both as an *Observer* and a *Subscriber*. Each of these *Processor*s upon being subscribed to subscribe to the previous *Transformer* in the operator chain. Finally, the subscription chain reaches the first *Observable* which starts emitting items into the stream.

In the previous examples if we would replace Observables with Streams, most of the things said would still be true – outputs would be exactly the same, figure 1.20 would look similar too (the flatMap logic would be handled differently, but the vertical to horizontal processing would remain quite the same) and subscription works similarly too (instead of subscribing to, Streams are 'started' by being collected or reduced [34]).

1.3.8.2 Schedulers

One of the features that RxJava has, but Java Streams does not, is an ability to switch the execution of a stream between processing threads.

Although the order of operations depicted in 1.20 is not wrong, it is not absolutely right either. It is true only because we are executing all operators on the same thread. By default, RxJava is single-threaded which implies that all operators are executed on the same thread on which `subscribe()` method is called.

To switch between the threads, we can use `subscribeOn()` and `observeOn()` operators.

- The `subscribeOn()` operator designates which thread the Observable will begin operating on, no matter at what point in the chain of operators that operator is called.
- The `observeOn()` operator, on the other hand, affects the thread that the *Observable* will use for operators following this operator. For this

reason, we may call `observeOn()` multiple times at various points during the chain of operators in order to change on which threads certain of those operators operate.

In RxJava we have several options how to switch the execution:

- Schedulers.newThread() – Switches the execution to a new thread every time it is invoked. Should be used with caution because spawning too many threads can degrade performance of the system.
- Schedulers.io() – Switches the execution to a new thread from a thread-pool or creates a new thread if no thread is available in the pool. This is more
- Schedulers.computation() – Similar to *io()* in a sense that it is backed by a thread-pool, but the pool is limited to the number of processors in the device.
- AndroidSchedulers.mainThread() – A special scheduler available only on Android that switches the execution to the Android's main Looper. This is very important as most of the UI related methods must be called from the main Looper's thread.

More about the topic of schedulers available from [35, 36].

1.3.8.3 Cold vs. Hot observables

Depending on the nature of an Observable, it might be *hot* or *cold*. A *hot Observable* may begin emitting items as soon as it is created, and so any observer who later subscribes to that *Observable* may start observing the sequence somewhere in the middle. Such *Observable*s are often subjects to backpressure. A *cold Observable*, on the other hand, waits until an observer subscribes to it before it begins to emit items. Such an observer is guaranteed to see the whole sequence from the beginning. [37]

It might seem like a small implementation detail but not knowing the difference might cause a lot of confusion when debugging, especially to an unexperienced programmer.

Chapter 2

Analysis and design

2.1 Web version of Ackee Planner

Ackee Planner already exists as a web service available at [38]. There are two possible use cases for using the service:

- A user wants to arrange a meeting with someone else.
- A user received an invitation to a meeting from someone else.

To better understand the functionality of Ackee Planner, let's analyze flows through the app from the perspective of a user who wants to arrange a meeting and another user who received an invitation.

2.1.1 Arranging a meeting

First let's analyze a flow through the web version of Ackee Planner from the perspective of a user who wants to arrange a meeting.

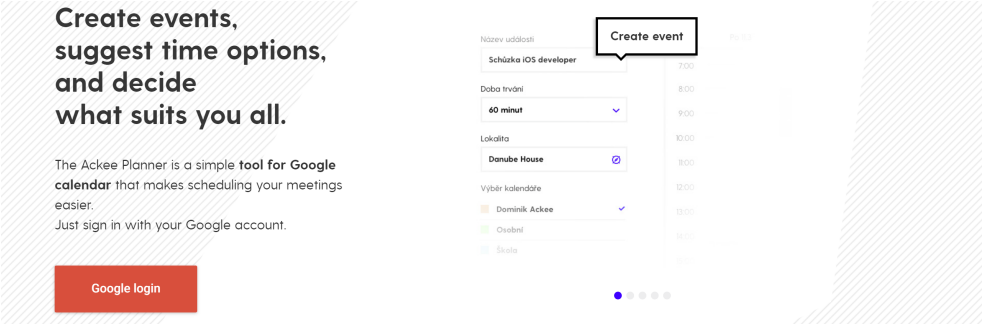


Figure 2.1: A login screen.

The user visits the application's website and is faced with a login screen 2.1 on which he has to log in. When he logs in for the first time, a Google's

standard permission dialog appears. He must grant permissions to backend for it to be able to access and manage calendars associated with his Google Account. If he declines to grant the permissions, we don't let him in and show the login screen again. If he accepts, proceed to the main screen.

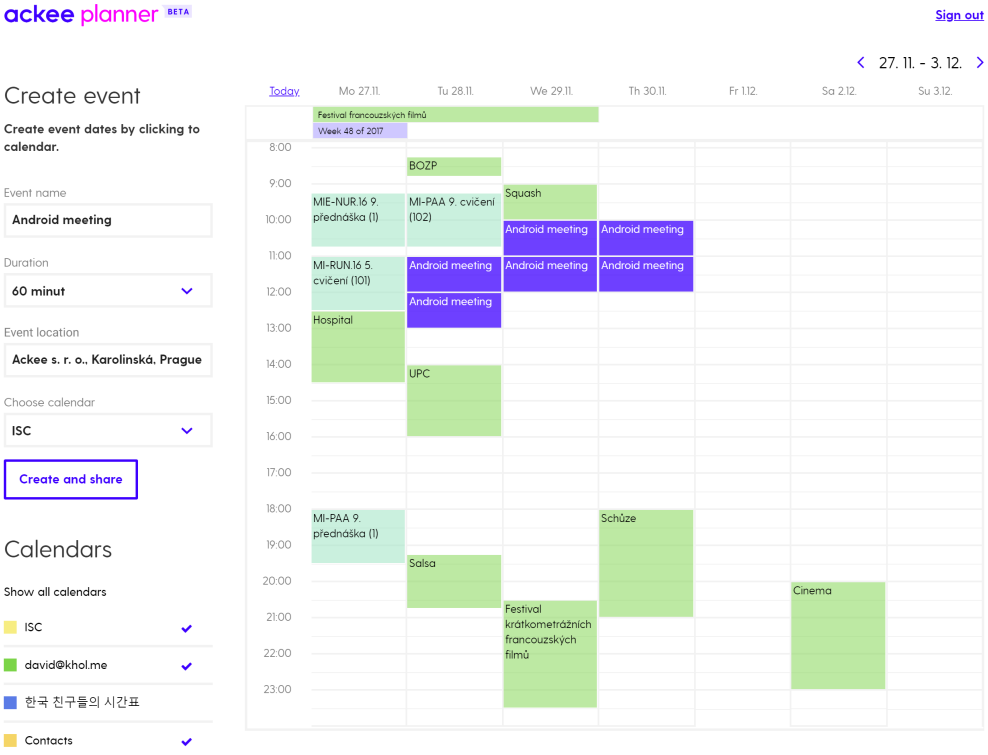


Figure 2.2: The main screen.

On the main screen 2.2, the user is presented with a large calendar view which he can click on to define available time-slots. He can also see other events in his calendars to make his decision easier. When he is not interested in events from some of his calendars, he can toggle the visibility of all events from specified calendars. Here he also defines a title and duration of the meeting and a location where the event will take place. When typing the location's name, suggestions are displayed below the input box. Finally, the user must choose to which calendar the event should be saved after being accepted by guests.

After all details of the meeting were specified, he may click on 'Create and share' button and continue to the next screen. In case he clicks on the button before specifying all details, an error message appears.

On the last screen 2.3 the user is presented with a URL address of the meeting created on the previous screen. The user can copy the link and send it to whomever and however he wants or he can write email addresses of

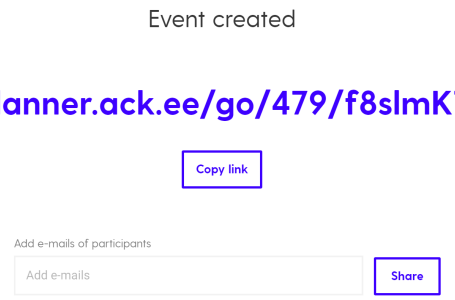


Figure 2.3: A confirmation screen.

participants and the system will send an email to them containing an invitation message and the URL address.

2.1.2 Accepting an invitation

Now let's analyze a flow through the web version from the perspective of a user who received an invitation to a meeting.

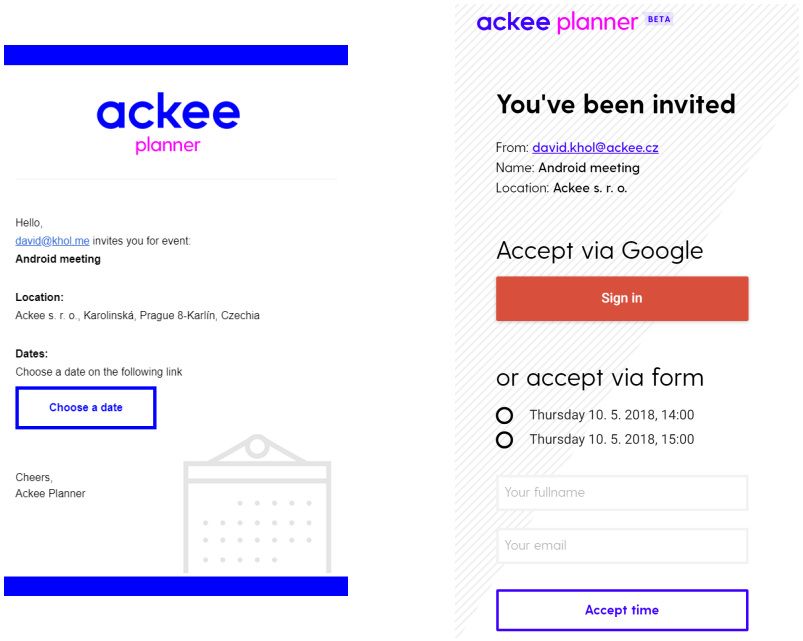


Figure 2.4: An invitation email and an invitation screen without logging in.

In most of the cases, he receives an invitation through an email. When he clicks on a button in the email, he is redirected to the invitation screen. In cases that he receives a direct link to the meeting, he is redirected to the invitation screen as well.

On the invitation screen, he has an option to log in using Google account or to accept the invitation as an anonymous user and only to specify his name and optionally his email address. For people who don't have a Google account or don't want to give permissions to access their calendars, most commonly due to privacy concerns, the anonymous option is added.

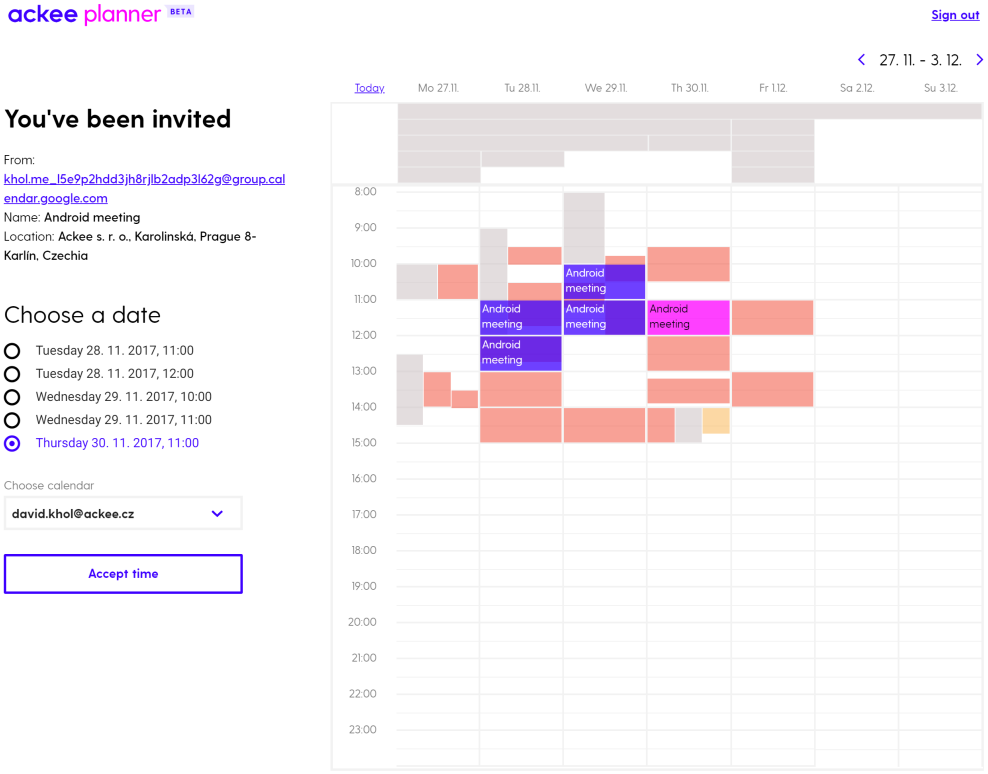


Figure 2.5: The main screen.

After logging in, the user is faced with the main screen. It has a similar interface to the main screen of the meeting-arrangement flow. It contains a large calendar view, where events proposed by the host are displayed. Details about the meeting and list of time-slots are placed to a side of the calendar view. The user can select an event by clicking either on an event in the calendar view or in the list. He has then an option to choose the calendar which the event will be saved to. The clicking on the 'Accept time' button takes the user to a confirmation screen where he is presented with a confirmation screen showing a review of what date he has chosen.

2.2 Design of Android version of Ackee Planner

When I started to design the Android version I had just a general idea about what is Ackee Planner meant for but didn't know details of the service. Unin-

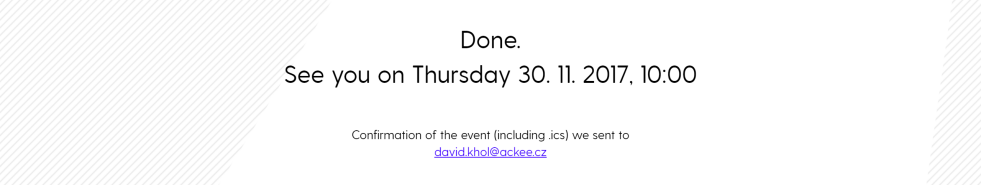


Figure 2.6: The confirmation screen.

fluenced by the existing implementation I tried to think of features I would like to see in the service. I analyzed the web version and existing backend only after a considerable time spent by brainstorming and sketching the mobile version. An overview of all the sketches is shown in the figure 2.7.

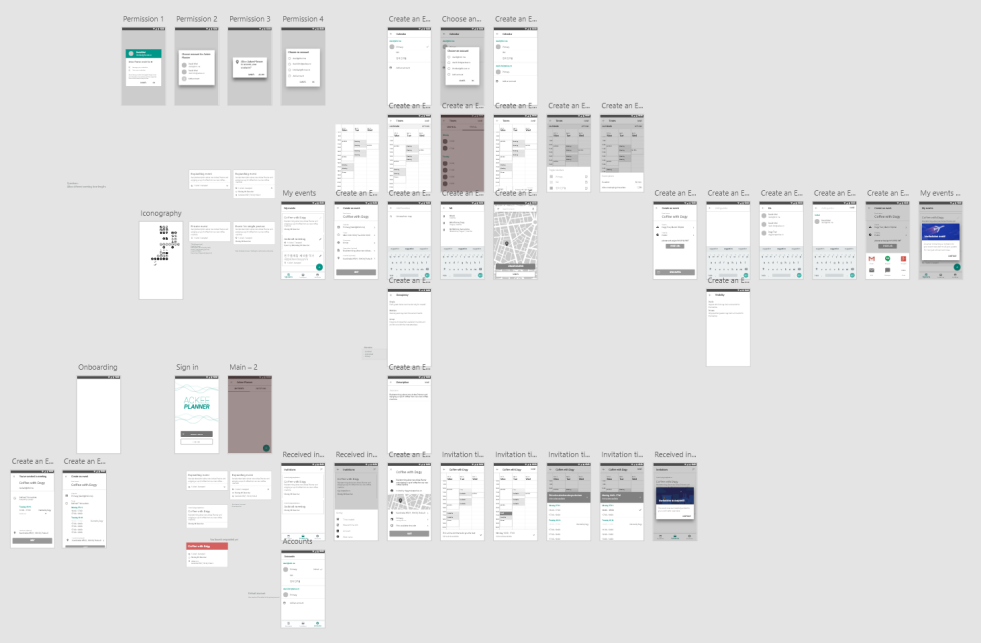


Figure 2.7: Sketches of the Android version of Ackee Planner.

Following are some of the functional requirements I came up with.

The central entity of the whole app is a *Meeting* and it should be possible to define at least the following attributes:

- *Title* – A short text summarizing the meeting.
- *Description* – An optional description that can span multiple sentences.
- *Location* – An optional location where the meeting will take place.
- *Times* – A set of potential time-slots for other people to choose from.

- *Duration* – How long the meeting will be.
- *Visibility* – Whether anyone (Public) or only authorized users (Private) may see the details of a meeting.
- *Guests* – A list of people invited to the meeting.
- *Occupancy* – How to manage time-slots in case of multiple guests. Either allow only one person per time-slot (Single), allow several people to claim the same time-slot (Multiple) or force all participants to agree on the same time-slot (Group).
- *Calendar* – Each attendee may choose what calendar the meeting should be saved to.
- *Reminders* – Each attendee may set up reminders for the event.

The user should be:

- able to list through the meetings he has created in the past
- able to view details about previously created meetings
- able to edit details of meetings that didn't happen yet
- notified when he receives an invitation from someone else
- able to list through previous invitations
- able to select meeting's location from a map
- able to search for a location by its name or address
- able to define guests by their email addresses or just by name
- able to cancel a meeting
- able to open the application directly from an invitation email
- able to see all events from all of his account's calendars at the same time
- able to change between accounts
- alerted when a time-slot he has defined is conflicting with another event
- able to intuitively define time-slots
- able to intuitively choose from available time-slots
- able to change the visibility of calendars when previewing time-slots

After analyzing the backend it became clear that some of the ideas are not feasible because the backend lacks certain functions. Thus it was difficult to implement every requirement idea. Some examples include:

- It is not possible to cancel or otherwise modify an already created meeting.
- The backend does not provide an endpoint for retrieval of all previously created events of the user.
- A user is notified about an invitation only via email, there is no support for native notifications.
- Meetings don't support description attribute.
- Meetings are always one-to-one. It is not possible to allow multiple people to claim the same time-slot.
- Meetings are always public to anyone who has a link to the meeting. Malicious users can claim all available time-slots as an anonymous user as there is no mechanism to prevent anonymous users from claiming multiple time-slots.

2.3 Calendar layouts

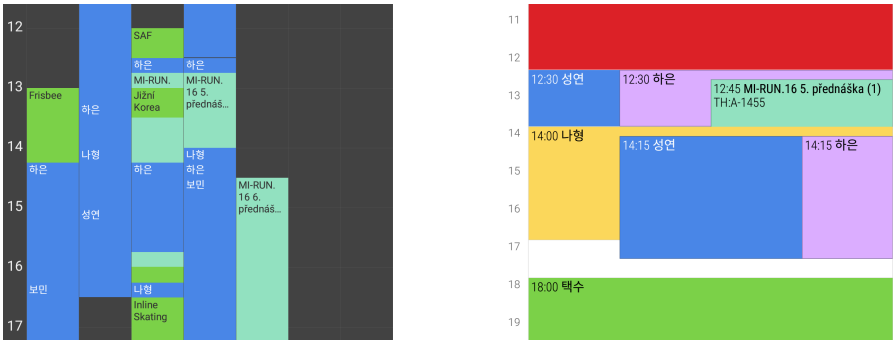
Defining potential time-slots of a meeting for other users to choose from is the central activity of the application. It is important that this action is as user-friendly as possible. It is important to give the user a quick overview of his schedule and enable him to find and choose an empty time-slots in his schedule effortlesly.

I downloaded 15 popular calendar applications from Google Play Store to analyze how various applications cope with the problem. Based on observations I defined these four tiers of how we can handle situations where two or more events overlap in time:

- **Tier 1** – Drawing events over other events.
- **Tier 2** – Drawing events next to other events in a cascading manner. Later events are always displayed to the right of previous events.
- **Tier 3** – Drawing events next to other events in columns. Later events are positioned to the left when the previous event in that column already ended. Otherwise create a new column with the event positioned to the right side.
- **Tier 4** – Drawing events in such manner that no event is drawn over other events and each event claims as much space possible without restricting other events.

A list of analyzed applications and notes on how does each application handle the layout is described in a figure 2.13.

Tier 1 – ovelapping events



(a) Drawing over other events.

(b) Drawing over other events.

Figure 2.8: Ovelapping events in Simple Calendar and Business Calendar apps.

Figure 2.8 shows two examples of how some applications handle overlapping events. The left version draws each event in full width and let following events be drawn over previous ones. This may confuse the user because it is not clear when does an event end. When two events start at the same time, one of them will be completely blocked by the other one. The right version is more sophisticated, but still suffers from the problem that we are not sure when an event ends. Moreover, one of the events is not displayed at all.

Tier 2 – cascading events

Figure 2.9: Cascading events in a web version of Ackee planner.

Figure 2.9 shows two screenshots of the web version of Ackee Planner. Pictures show the situation before and after we add an event that is overlapping two separate groups of events. Two events at the top get positioned to the left

of the new event; four bottom events get positioned to the right. Although the two groups originally didn't overlap, they now share constraints through the new event. Note that this situation could be resolved in much better way if the new event was just placed to the same side of both groups.

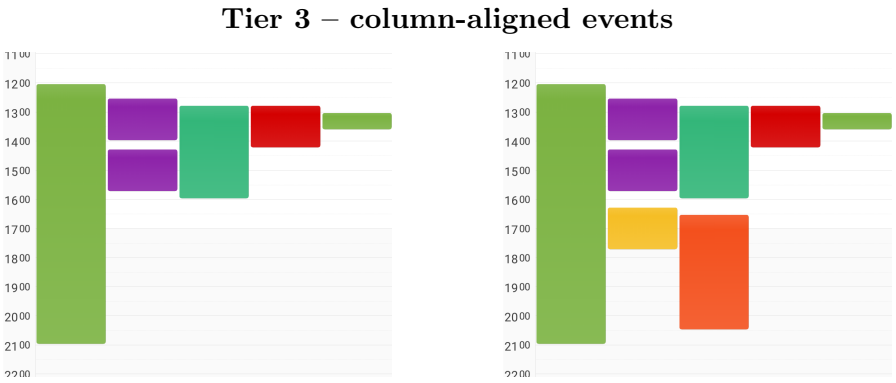


Figure 2.10: DigiCal's column-aligned algorithm.

Figure 2.10 presents column-aligned approach. This approach aligns all horizontal space of events to columns and may horizontally constrain events even if there is empty space next to them where they could be expanded into. Even though there is enough space available next to the orange event, its width is limited to the width of a single column.

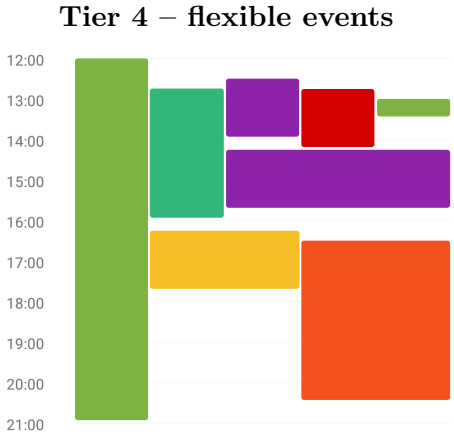


Figure 2.11: Google Calendar's flexible width algorithm.

In figure 2.11, we can see Google Calendar's flexible algorithm. In contrast to the column-aligned approach, this algoritm does not unnecessarily constrain events and uses all available empty space next to them. It also comes up with a better space management. It analyzes where an event should be positioned

in relation to other events to provide it with more space, when feasible. We will analyze how to implement a flexible algorithm in the next chapter.

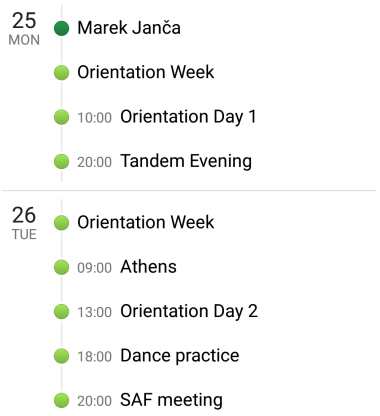


Figure 2.12: Calendar agenda.

As a result of the analysis, I realized that classic calendar applications and scheduling applications are inherently different:

When we are notified about event's date and time, we don't care that much if it collides with other events because there is nothing we can do about it. Think of an event about which we have no say in the matter such as movie theatre projection. We cannot change the running time and our only option is to drop the event. Although it is useful to see colliding events at a glance, it is not critical. We are just 'consuming' events from other sources.

On the other hand, when we are in charge of deciding when an event will be held, it is important to see the context. It would be foolish to offer other people a date we cannot participate in. We are only 'producing' events for others.

Many of the analyzed applications didn't even support calendar view and only provided agenda style list of events, such as the one depicted in figure 2.12. These applications were not designed to provide an easy way to schedule a meeting, but rather focus on a linear representation of events that is always easy to follow.

Tier	Name (package)	Note
0	**Cal** *(com.anydo.cal)*	Agenda only.
0	**TimeTree** *(works.jubilee.timetree)*	Agenda only.
0	**TimeBlocks** *(com.hellowo.day2life)*	Agenda only.
0	**Haroo** *(com.yunasoft.awesomecal)*	Agenda only.
1	**Calendar** *(com.simplemobiletools.calendar)*	Drawing over each other.
1	**Business calendar** *(netgenius.bizcal)*	Drawing over each other in unpredictable manner.
1	**Business calendar** *(com.appgenix.bizcal)*	Looks like column approach, but sometimes draws over each other and can completely hide some events.
3	**New Calendar** *(info.kfsoft.calendar)*	Agenda only.
3	**Calendar** *(com.boxer.calendar)*	Column approach.
3	**Calendar+** *(com.joshy21.vera.free.calendarplus)*	Column approach.
3	**aCalendar** *(org.withouthat.acalendar)*	Optimized column approach.
3	**Easy Calendar** *(free.calendar.schedule.reminder.todo.agenda.note)*	Column approach.
3	**WAVE Calendar** *(com.esites.ecal)*	Column approach.
4	**Calendar** *(com.google.android.calendar)*	More sophisticated algorithm.

Figure 2.13: List of analyzed Android applications

Chapter 3

Realisation

3.1 Android

We have an API endpoint where we can authenticate the user using Google account or let him proceed as an anonymous user. When the user is logged in with a Google account, we can retrieve his calendars and events in his calendars. Furthermore we can create a new meeting, invite other people to the meeting, retrieve details about a meeting and accept invitations to other people's meetings.

To log the user in using a Google account, we must first implement OAuth 2.0 authentication. OAuth 2.0 is the industry-standard protocol for authorization. [39] Simply said, OAuth allows us to access resources on behalf of the user. How to set up OAuth authentication and how does OAuth work is beyond the scope of this thesis.

We can leverage the fact that on Android we can retrieve user's events locally. This approach has several benefits compared to retrieving the data from the endpoint:

- We don't need internet access to retrieve events. When the user loses network connection for a while, everything will continue to work the same. It can also save some mobile data.
- We can retrieve all events from all calendars from all user's *accounts* at once. With the API endpoint, we are always limited to displaying events from only one of the user's accounts.
- We can display user's events even if he decides not to log in using his Google account. The user may decide to proceed to accept an invitation

anonymously and thus not give us the access to his calendars.

3.1.1 Permissions

There is a downside to this approach though – we must ask the user for permission to access his calendars. In our case, this is not a problem because the user has to give us the permission anyway if when he wants to authenticate with our endpoint.

All permissions are irrelevant when the user is just accepting an invitation because the user may proceed as an anonymous user and not allow us to access to his calendar. But to create a meeting, we need to authenticate the user by logging him in.

To authenticate the user, we need his email address. There are two ways how to retrieve it. Either we can use the `GET_ACCOUNTS` permission and ask for his account with modal dialog, or use the `READ_CALENDAR` permission and retrieve the user's accounts from the *calendar content provider*.

The downside of the `GET_ACCOUNTS` permission is that it is grouped together with `READ_CONTACTS`. [40] Whenever we ask for the `GET_ACCOUNTS` permission, system presents the user with a dialog asking him for an access to his contacts. If the user would not have granted the access to all of his contacts (mostly for privacy reasons), we could not log him in.

We used the `READ_CALENDAR` approach mainly because in our case there are virtually no drawbacks of using the `READ_CALENDAR` permission.

3.1.2 Deep Links

Deep Links on Android allows us to open our app when the user clicks on a URL. Normally, following a URL redirects the user to a web browser. Using *Deep Links* we can hook into the process by defining certain attributes in the metadata of our application. When we define a deep link to our application and the user clicks on a URL matching our metadata, instead of redirecting him to the web browser we can open a custom activity based on the followed URL.

This technique is used to open the app when the user click on a link in his email. Instead of showing the web-based form, we take him directly to the invitation screen within our app.

3.2 Calendar

As a part of Ackee Planner, there is a screen where the user defines a list of time-slots for guests to choose from. One could argue that this is the

fundamental action of the application and that it deserves the attention the most.

We can represent times in a **textual form** and let the user edit times and dates by using input fields and/or native Android dialogs. This solution is very easy to implement but also very **user-unfriendly**. It is difficult for the users to imagine the time spans in their heads quickly. Furthermore it is difficult to communicate the fact that their events are overlapping with each other in their calendar.

Another option is to show times in a **graphical form** and let the user edit times and dates by gestures such as clicking and swiping. This solution is usually more **user-friendly** but can be difficult to implement. We can present more information that would not otherwise fit on the screen or would be too confusing to understand.

In the previous chapter, I researched similar applications and came up with several options how to present calendar in a graphical form. 2.3

- Let events be drawn full-width and allow them to be drawn over each other. This is not really useful. When they are drawn over each other, it might become unclear when does an event end, because it is covered by other events.

- Cascading approach. This may create a cascade of events where each event gets smaller every time a new event is added. This solution is currently used in the web version of Ackee Planner.

- Column aprroach. This doesn't suffer from the cascading problem as the previous option, but still causes wasteful or unnatural layouts.

- Flexible algorithm. Not many applications go this far to implement a calendar view. Actually only one of the analyzed applications used this approach.

3.2.1 Flexible calendar layout algorithm

Before introducing the actual algorithm, let's define terminology that will be used throughout the explanation.

- Two events **overlap in time** if there is a moment in time where both events happen concurently.

- Two events **share constraints** when their widths are tied together. When one event's width changes, the other event's widths should be updated too. When two events **overlap in time**, they always share constraints.

The opposite statement does not always hold. It is possible for two events to **share constraints** even though they do not **overlap in time**. The reason will be explained later in the algorithm.

In order to develop a flexible algorithm, following rules must be satisfied:

1. Two or more events overlapping in time should not be drawn over each other.
2. Two or more events overlapping in time should divide the avaiable width space equally.
3. Events should claim as much avaliable width space as possible for themselves without limiting other events.
4. Prefer such layouts where combined width of all events is maximized (without breaking the previous rules).

The algoritm accepts a list of events as its input. Each event defines a starting time and an ending time.

```
A (start=0, end=3)
B (start=1, end=2)
C (start=1, end=2)
D (start=2, end=4)
```

Listing (3.1) Sample input

The algorithm returns the input list of events and defines two additional attributes to each event – its width and its horizontal offset.

```
A (start=0, end=3, width=0.33, offset=0.0)
B (start=1, end=2, width=0.33, offset=0.33)
C (start=1, end=2, width=0.33, offset=0.67)
D (start=2, end=4, width=0.67, offset=0.33)
```

Listing (3.2) Sample output

We can break the algorithm down into 5 steps:

1. building **Timeline Steps** from input *events*,
2. building **Concurrency Groups** from *Timeline Steps*,
3. building an **Event Graph** from *Timeline Steps*,
4. calculating **widths** from *Concurrency Groups* and
5. calculating **offsets** from *widths* and the *Event Graph*.

3.2.1.1 Building a Timeline

A **Timeline** is a series of additions and removals of events based on their starts and ends. Each event is included in the timeline exactly twice, once for its start and once for its end.

- For each input event create two items containing `EventId`, `Time` and `Action` attributes – once for its addition `(event.id, event.start, ADD)` and once for its removal `(event.id, event.end, REMOVE)`.
- Sort all the items by their `Time` attribute.

Successive items from the `Timeline` with the same `Action` are merged together into **Timeline Steps** that contain all IDs of the merged items.

```
A (start=0, end=3)
B (start=1, end=2)
C (start=1, end=2)
D (start=2, end=4)

Timeline    Timeline Steps
+0A
+1B
+1C         +ABC
-2B
-2C         -BC
+2D         +D
-3A
-4D         -AD
```

Listing (3.3) An input of 4 events, their Timeline and their Timeline Steps

Listing 3.3 shows an example of simple input and its `Timeline` and `Timeline Steps`.

Now we have an oscilating list of `Timeline Steps` where odd-indexed groups represent additions of events and even-indexed groups represent removals of events.

3.2.1.2 Building Concurrency Groups

Concurrency Groups are sets of events where all the events in the group *share constraints* with each other. We can greatly simplify next steps of the algorithm if we don't include groups that are subsets of other groups. Keep only the maximal `Groups`.

1. Create an empty buffer.

2. Retrieve next set of events from the `Timeline` and add the events to the buffer. Record a set of events currently in the buffer.
3. Retrieve next set of events from the `Timeline` and remove the events from the buffer.
4. Keep repeating steps 2 and 3 until the `Timeline` is empty.

```
Timeline Steps  Concurrency Groups
+ABC             ABC
-BC
+D               AD
-AD
```

Listing (3.4) `Timeline Step`s and `Concurrency Group`s

Listing 3.4 show an example of Timeline Steps and their transformation to Concurrency Groups.

Now we have all the groups of events where each group defines a maximum set of events that each *share constrains* with others.

3.2.1.3 Calculating widths

We can compute `Width`s using the `Concurrency Groups` from previous step.

When an event's width has been set, we call it *resolved*. Vice versa, when an event's width has not been set yet, we call the event *unresolved*.

We assign a number between 0 to 1 to each `Concurrency Group` called *Available space*. This number tells us how much width we can split between the events of the `Group`.

Space Constraint is a ratio between remaining *available space* and number of unresolved events in the `Group`. Smaller the constraint is, bigger the priority the `Group` has to resolve the sizes in the `Group`.

The algorithm for calculating widths is following:

1. Set *Available space* of all groups to 1.
2. Recompute space constraint of all groups containing any unresolved events.
3. Find the group with the smallest space constraint.
4. Set the width of each unresolved event in the group to the group's Space constraint.

5. Repeat steps 2, 3 and 4 until all events have been resolved.

It might be difficult to imagine what does the algorithm actually do. Listing 3.5 shows the process of the algorithm using the data from the previous step. At the beginning we know nothing about width of any of the events. We just know there are two `Concurrency Group`s `ABC` and `AD`. We set their `Available space` to 1.0 and recompute space constraints. The `ABC` has the smallest space constraint so we resolve it first. We set width of all `A`, `B` and `C` to 0.33 because none of them has been resolved yet. Next we repeat the process – recompute available space and space constraints. This time `group AD` already contains a resolved event `A` whose width has to be removed from the `group`'s available space. This time, the `AD` has the smallest space constraint so we resolve it now. We set width of `D` to 0.67 and leave the `A` as is, because was already resolved before.

```
        A width = ?
        B width = ?
        C width = ?
        D width = ?
-----------------------------------------------------
Concurrency Groups  Available space  Space constraint
ABC                 1.0              1.0 / 3 = 0.33
AD                  1.0              1.0 / 2 = 0.5
-----------------------------------------------------
Set width to all unresolved events in ABC group.
        A width = 0.33
        B width = 0.33
        C width = 0.33
        D width = ?
-----------------------------------------------------
ABC                 0                resolved
AD                  0.67             0.67 / 1 = 0.67
-----------------------------------------------------
Set width to all unresolved events in AD group.
        A width = 0.33
        B width = 0.33
        C width = 0.33
        D width = 0.67
```

Listing (3.5) Calculating widths

Note that we cannot simply sort `Group`s by their space constraints and skip step 2 of the iteration, because the space constraints of `Group`s change every iteration. When we resolve a `Group`, space constraints of other `Group`s may change.

3.2.1.4 Building an Event Graph

Building an *Event Graph* (*Graph* from now on) is the most complicated step of the algorithm but also the most important. Using the *Graph* we can find out where an event should be positioned in relation to other events.

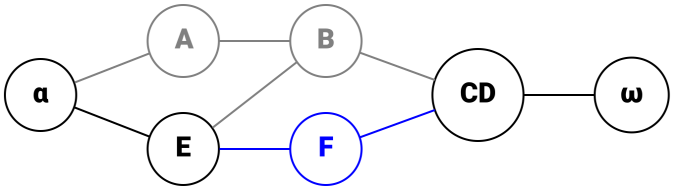


Figure 3.6: Example of a Graph.

A *Graph* consists of interconnected `Group`s. A `Group` is a set of one or more events that overlap in time. The order of events within a `Group` does not matter. Figure 3.6 shows an example of a *Graph* that is composed of several `Group`s. The leftmost and the rightmost `Group`s do not contain any events and serve as sentinels. Throughout the algoritm, they are referred to as $\alpha$and ω. No `Group` can be connected to the left side of α, resp. to the right side of ω. You can think of them as infinitely long events that never finish and are always in the active *state*. When we choose any `Group` in the *Graph* and traverse it to the left (resp. right), we will always reach α(resp. ω).

`Group`s can be connected to other `Group`s from the left and the right sides. When two `Group`s are connected, it means that they *share constraints* (and so do all of their events). Because of the shared constraints, they must be positioned next to each other.

- This relation is **reflexive**. When one `Group` is to the left (resp. right) side of another, the other `Group` must be to the right (resp. left) side of the first one. Mutliple `Group`s can be connected to a single `Group` and single `Group` can be connected to multiple `Group`s.

- This relation is also partially **transitive**. See figure 3.6. Although groups `A` and `CD` are not directly connected to each other, they *share constraints* through the group `B` which lies in between of them. As a result, a `Group` is constrained to all other `Group`s that we can reach by traversing other `Group`s only to the right or only to the left of the original `Group`. `Group E` is directly constrained to α, `B` and `F` and transitively constrainted to `CD` and ω, but is not constrained in any way to the group `A`.

A `Group` can be either in an **active** or in a **finished state**. The state tells us whether all the events in this group are over or still active at the moment

when we make changes in the graph. When we add a `Group` to the graph, its state is set to `active`. When events in the group finish, we set the state to `finished`. Once set to `finished`, it can never be changed to `active` again.

A **bottom line** of the graph is a list of `Group`s in the *Graph* starting with α and ending with ω where each `Group` is the previous `Group`'s last following `Group`. In the example of the figure 3.6, bottom line is [α – `E` – `F` – `CD` – ω].

A **space in the bottom line** (or just a *space*) is one or more connected events in the bottom line that are in *finished* state. In the figure 3.6, we have just added a group `F` (highlighted in blue) between groups `E` and `CD`. Before the addition the bottom line was [α – `E` – `B` – `CD` – ω]. Group `B` was in a finished state and acted as a *space in the bottom line*. When we add a new group of events into the *Graph*, we always connect it to two *active* `Group`s that were separated by a *space*.

Each traversal across the fully-built *Graph* from α to ω represents a `transitively-constrained Group` (`TC Group` from now on). In our example are three such `TC Group`s:

- ABCD – composed of [α – `A` – `B` – `CD` – ω]
- BCDE – composed of [α – `E` – `B` – `CD` – ω]
- CDEF – composed of [α – `E` – `F` – `CD` – ω]

These groups closely match `Concurrency Groups` from the previous step. That is no coincidence. Many of the *Graph*s built this way have exactly the same `TC Group` as their `Concurrency Groups`. As a matter of fact, the `Concurrency Groups` from the previous step are the perfect minimum of `TC Group`s we strive for.

By adding a `Group` into the *Graph* we may create a new `TC Group` that is not included in `Concurrency Group`s. To achieve a perfect layout, we should not create any new `TC Group`s.

As it turns out, there are cases where it is simply not possible to organize `Group`s in the *Graph* perfectly. Sometimes we just have no other choice than to create a new `TC Group` that is not included in the `Concurrency Group`s. In such cases, widths from the previous step are invalid and have to be recomputed with all the `TC Group`s!

The algorithm for building a *Graph* is following:

1. Create two empty `Group`s α and ω that will be used as left and right sentinels of the graph.

2. Create an empty `Group` and add it in between α and ω. Set its state to finished. α and ω must be part of the same graph, but cannot be connected to each other directly, because the algorithm for finding an empty *space* would fail.

3. Find a *space in the bottom line* of the graph. It is guaranteed that there is at least one empty *space* in the bottom line.

4. Retrieve next set of events from the timeline. We want to add them to the graph. Create a new `Group` in an `active` state with these events. Connect the `Group` to two `Group`s separated by a *space in the bottom line*.

5. Retrieve next set of events from the timeline. We want to set these events to the *finished* state. Find all `Group`s that they are included in. If a `Group` is composed only of the retrieved events, we can set the `Group`'s state to finished. If the `Group` contains both *active* and *finished* events, split it into two and connect them together – one that cointains only *active* events and one that contains only *finished* events.

6. Repeat steps 3, 4 and 5 until the timeline is empty.

During the step 3, when there are multiple *spaces*, we may attempt to reorganize the `Group`s in the *Graph*. If it is possible to reorganize the *Graph* without removing or adding new `TC Groups` in such a way that two *spaces* would appear right next to each other, we can combine those *spaces* into a single one.

If we leave two or more spaces in the bottom line, next step will introduce a new `TC Group` meaning that we add unnecessary constraints that we try to avoid.

In some cases it is possible to **rotate** a group of interconnected `Group`s. The *rotation* causes all connections within the group of interconnected `Group`s to be inverted – if an `A` was to the left side of a `B`, after rotation the `B` would be to the left side of `A`. *Rotating* a single *Group* has no effect.

This step was excluded from the final implementation of the algorithm because it was not possible to find such *rotation* in every case and caused unstable results. After closer analysis of the Google Calendar, I found out that its algorithm also sometimes produces suboptimal results.

Figure 3.7 shows an example of full process of building a *Graph*. Blue nodes represent `Group`s added in that step and red nodes represent `Group`s finished in that step. In the second last step of the algorithm it is impossible to reorganize the *Graph* in a way that would merge two *spaces* into one. The last step then adds a group `H` that is transitively-constrained to a group `F`. This introduces a new `TC Group` that was not included in the `Concurrency Group`s. As a result, the algorithm cannot achieve the perfect layout.

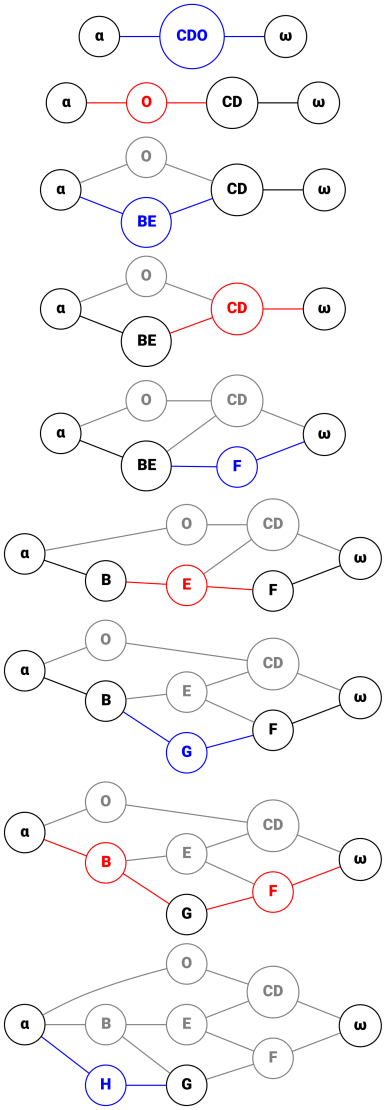


Figure 3.7: Process of building a Graph.

3.2.1.5 Calculating offsets

To calculate events' offsets we need the *Graph* and events' widths.

1. Calculate widths of each `Group` in the *Graph*. The width of a `Group` is simply the sum of widths of its events.

2. Calculate the offset for each `Group`. Offsets can be calculated by performing a depth-first traversal through the *Graph* and can be nicely expressed as a recursive function.

We start the traversal from the α `Group` and traverse the *Graph* to the right. The offset of a `Group` is the sum of widths of the previously traversed `Group`s. If we expand a `Group` more than once, we keep the maximal calculated offset.

3. Calculate the offset for each event. Offset of an event is equal to the offset of its enclosing `Group` plus sum of widths of previously calculated events.

3.2.2 Calendar View

There is no built-in `View` for displaying a calendar in the Android SDK. It is not very surprising because built-in `View`s and `ViewGroup`s are usually meant for simple, specific tasks such as displaying a label, a button, a slider or grouping other `View`s together. Calendars can be presented in many various forms and there isn't a single representation that could be applied to every calendar scenario.

There are some 3rd-party lfibraries available that give support for a calendar `View`. In our case, none of the libraries could be used without changing their code because we had a special requirement that none of them supported – we need to be able to display events in multiple layers – one layer for events already in the user's calendar and another layer for time-slots of the meeting.

There was no other option than to create our own `View`. There were two approaches we could choose from to implement the `View`:

- Draw everything into a bitmap and display the bitmap.
- Use a combination of built-in `View`s and `ViewGroup`s.

Many of the previously mentioned libraries used the first approach. It might be easier to organize the code using this approach but, in my opinion, it is limiting. We have to implement all gestures by ourselves – scrolling up and down, swiping left and right, zooming in and out, clicking etc.

The other approach offers a built-in support for scrolling, swiping, clicking and also gives many other benefits. Examples include:

- Free animations – we can nicely animate the changes in the `View` with just a few lines of code.
- Native effects – we can use native effects such as a shadow effect behind elevated items or a ripple effect when clicking on events.
- Accessibility – a visually impaired user can still use the `View` through use of accesibility tools.

- View caching – the framework caches `Views`' appearances. When a part of the view changes, it does not have to redraw everything, resulting in a potentially increased performance.

Implementation of any of the examples above would be a very time-consuming task with uncertain results. The drawback of this approach is that we have to understand a convoluted logic behind laying out, measurement and drawing of custom `Views` in order to implement the `View` correctly.

3.2.2.1 UI Automator Viewer

Android Device Monitor is a set of tools in the Android SDK, not well-known among developers, that is very useful for profiling, debugging and performance analysis of applications. One of these tools is *UI Automator Viewer* that comes handy specifically when analyzing view hierarchies of applications.

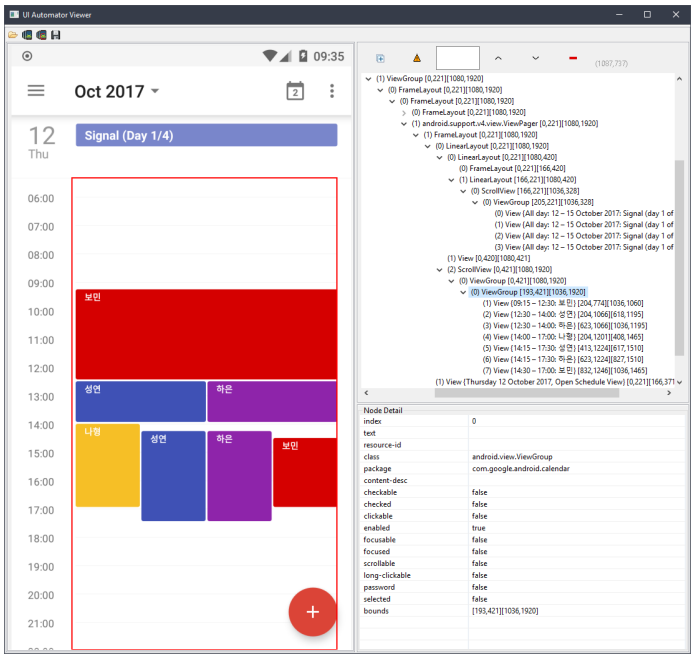


Figure 3.8: UI automator viewer interface

Starting with Android Studio 3.0, many of the tools are deprecated in favor of built-in tools in Android Studio itself. Nevertheless, the *Layout Inspector* tool (a replacement for the *UI Automator View*) still needs some more refinement (boundaries are displayed for all views by default and boundaries for a selected view are not properly highlighted in some cases) and, in my opinion, it is better to use the deprecated version at this moment.

Both tools retrieve the currently displayed view hierarchy on the device using ADB (Android Debug Bridge). We can inspect what *ViewGroups* were

used to build the hierarchy including some of their attributes and IDs. It can give us useful clues on how to implement some functionality of other applications.

This tool was of tremendous help when building complicated layouts by myself. Using the tool I got an insight to how the calendar view can be implemented by analyzing view hierarchies of other applications.

More information about *Hierarchy Viewer* can be found at [41] and information about *Layout Inspector* at [42].

3.3 Android version of Ackee Planner

Android version of Ackee planner should have a flow similar to the web version.

There are two use cases same as for the Web version of Ackee Planner and two more added:

- A user wants to arrange a meeting with someone else.
- A user received an invitation to a meeting from someone else.
- A user wants to review his previously created meetings.
- A user wants to respond to an invitation he postponed responding to before.

Due to space restrictions on screens of mobile devices, it was necessary to split some screens into smaller parts. Some of the original functions were enhanced.

The user is not required to log in before using the app. Instead of requiring all permissions from the user at the start of the app, he may use it until reaching a point where the permission is necessary to proceed.

Granting of permissions is deferred to later stages. This follows Android guidelines about permissions. Before SDK 23 (Android 6.0 Lollipop), users had to accept all permissions of an application before installing it to their devices. Users could not opt out from a particular permission if they wanted to use the application. Nowadays users can revoke permissions selectively and we have to check for permissions at runtime.

3.3.1 Arranging a meeting

Let's analyze a flow through the Android version of Ackee Planner from a perspective of a user who wants to arrange a meeting. First of all, by any means, he needs to have the application installed on his device.

When he first opens the application, he is presented with an empty screen and a message encouraging him to click on a Floating Action Button (FAB) to create a new meeting. Notice that, in contrast to the web version, there is

no log in required at this point. Clicking on the FAB takes him to another screen where he can define details about the meeting.

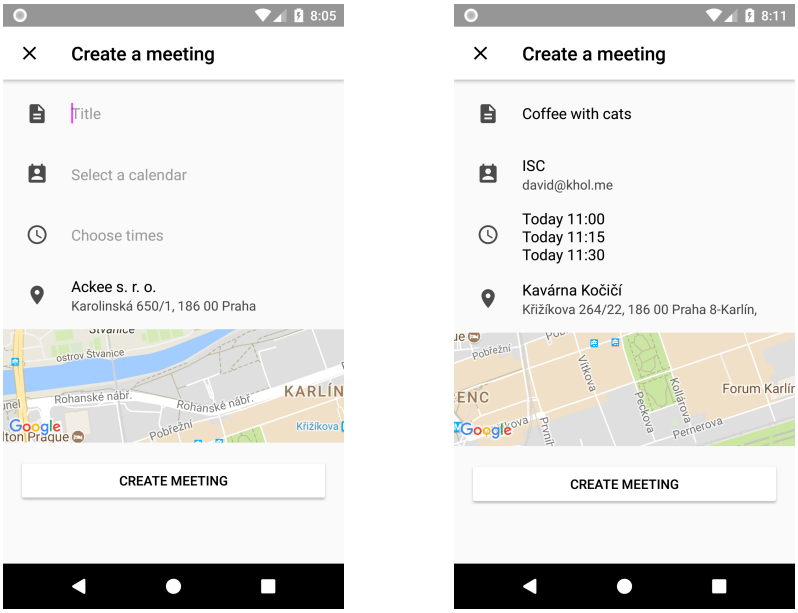


Figure 3.9: Empty and filled-in meeting creation screens.

On the *meeting creation screen* the user must specify details about the meeting – including its title, potential time-slots, its location and what calendar the event should be stored into when accepted. Furthermore a map of currently defined location is displayed.

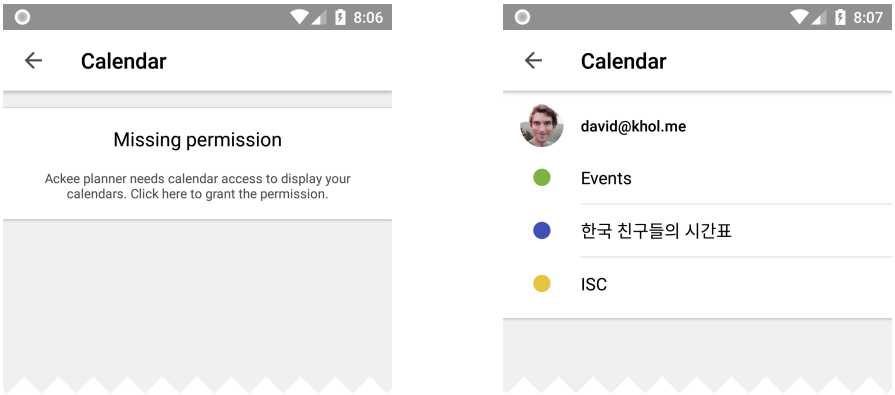


Figure 3.10: A calendar selection screen before and after granting permissions.

On a *calendar selection screen* the user is faced with permissions for the first time. If the user hasn't provided Calendar permission before, a message will be shown asking him to grant the permission.

The permission is used to display *all* calendars from *all* of his accounts. This is a big difference in comparison with the web version. In the web version the user is limited to be logged in using only one of his accounts. In the Android version, he can switch between accounts effortlesly without losing progress he made so far.

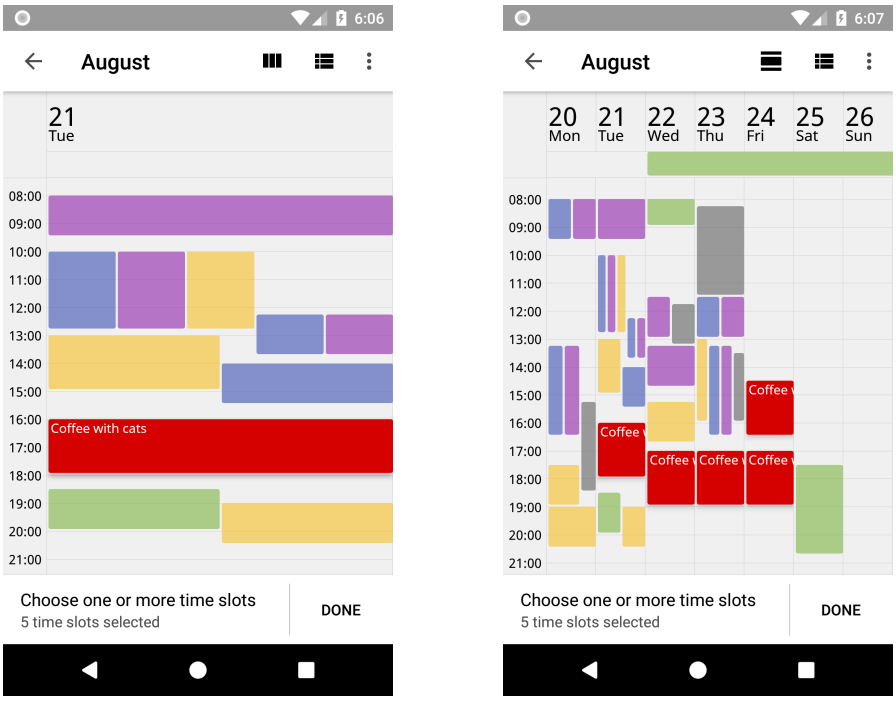


Figure 3.11: Daily and weekly views of time-slot selection screens.

On a *time-slot selection screen* the user must select at least one time-slot to proceed. He can define time-slots by clicking on the calendar. He can scroll horizontally to move between days (or weeks) and vertically to change diplayed time period of the day. Moreover, he may use a pinch gesture to zoom in, to increase precision when defining time-slots, or out, to have a better overview of the schedule.

Similarly to the web version of Ackee Planner, the user may toggle the visibility of his calendars in case that he is uninterested in its events. If the calendar view is too cluttered or hard to navigate, he can display textual agenda of time-slots defined so far.

On a *location selection screen* the user can define where the meeting will take place. As he writes the location's description, suggestions based on current query appear below, giving the user an option to select an address without writing the whole address by himself. In addition, next time he opens the location screen, previously entered addresses will be shown without the need of writing anything at all. Recent locations can be cleared by swipe gesture or

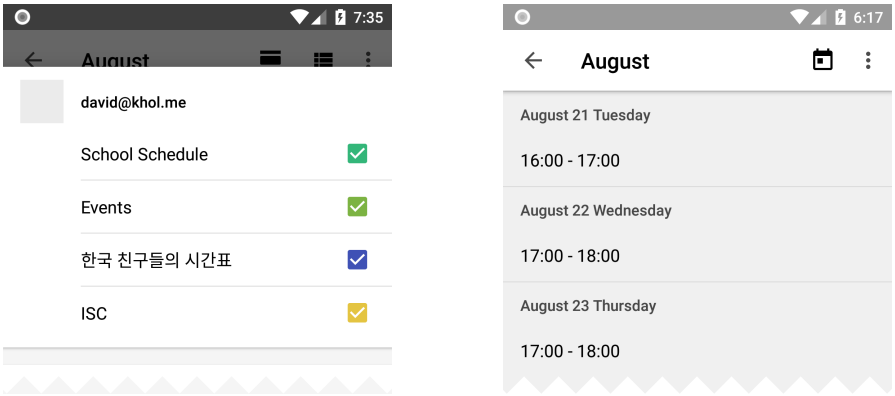


Figure 3.12: A dialog where the user can change visibility of his calendars on the left. Textual representation of currently defined time-slots on the right.

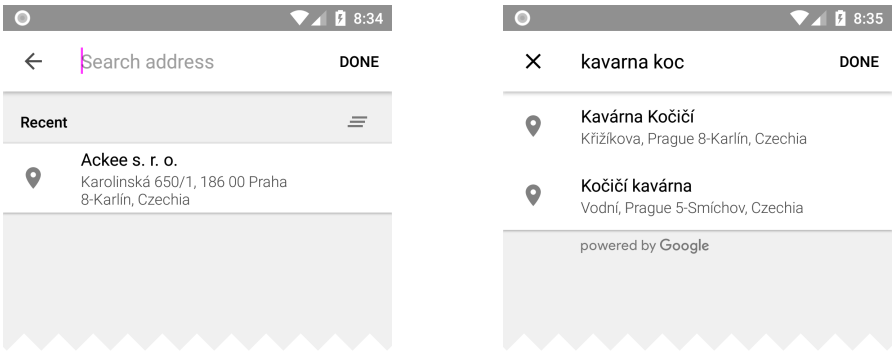


Figure 3.13: A location selection screen.

by clicking the icon above them.

Originally *the location selection screen* featured a full screen map view in which the user could zoom and pan around to find a desired location. It proved to not be very useful as it was just easier to find the location by entering an address or name. In the end, the fullscreen map was scrapped in favour of a small map preview on the *meeting creation screen*.

On the following *sharing screen* the user can send invitations to the just created meeting. Similarly to the web version, the user can copy the link and send it to whomever and however he wants. A native modal dialog will appear upon clicking on the url, where he can choose an application of his liking to share the meeting.

On a *guest selection screen* the user is asked for a permission to access the user's contacts in a similar way as on the *calendar selection screen* 3.10. This permission is not critical and the user may proceed without granting it. In such case he must type in full email addresses. If he grants the permission, he can choose to write either names of people or their email addresses and

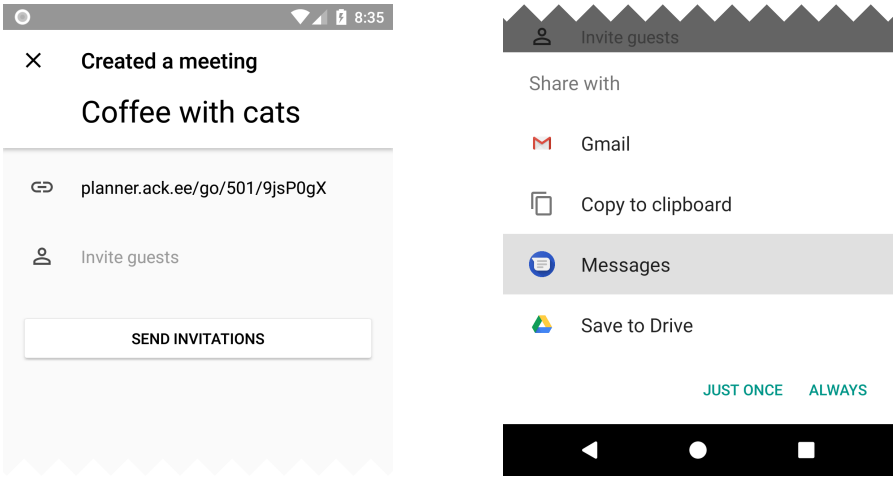


Figure 3.14: A sharing screen.

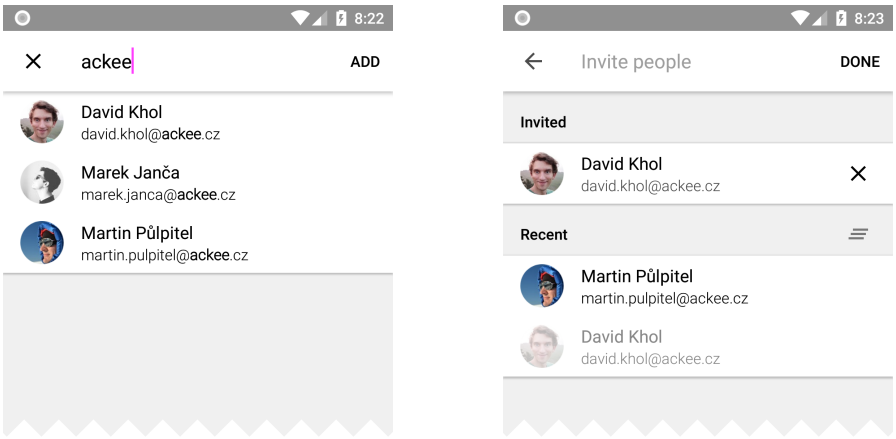


Figure 3.15: A guests selection screen.

suggestions matching the query will appear below. Similarly to the location screen, when the user creates another meeting, previously contacted people are remembered and shown first.

When the user is finished with sending invitations or navigates back, he is redirected back to the main screen.

3.3.2 Accepting an invitation

Let's analyze a flow through the Android version of Ackee Planner from a perspective of a user who received an invitation to a meeting.

Most of the time he receives the invitation via email. When the user is on a mobile device and clicks on the button in the email, an invitation screen within Ackee Planner automatically opens.

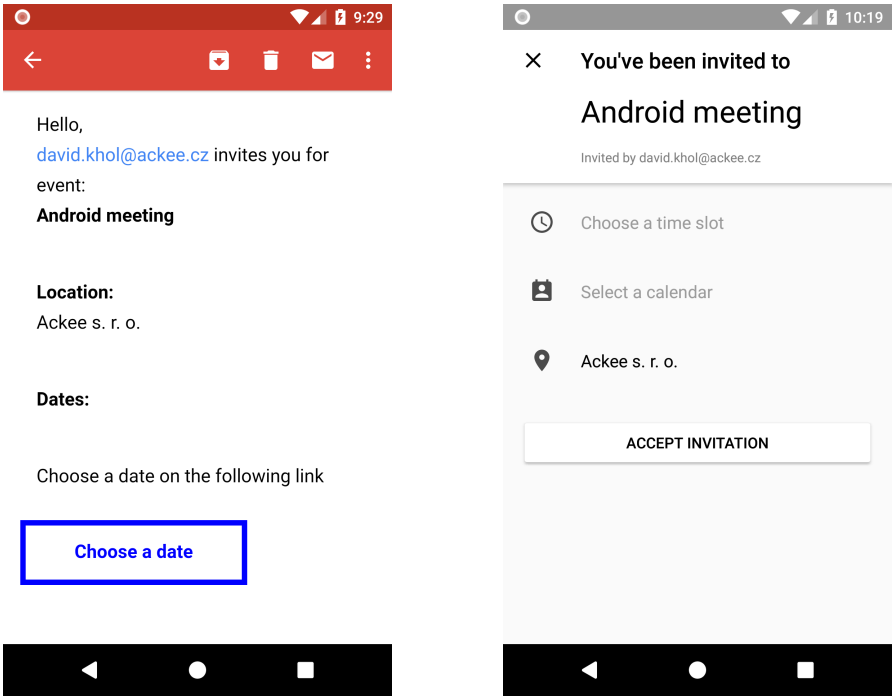


Figure 3.16: An invitation in Gmail app and within Ackee Planner.

He may then proceed to choose a time-slot that was proposed by an owner of the event. Most of the functionality remains the same as before – the user can toggle between calendar view and textual list, can toggle between displaying a single day or a week of events and can change visibility of his calendars.

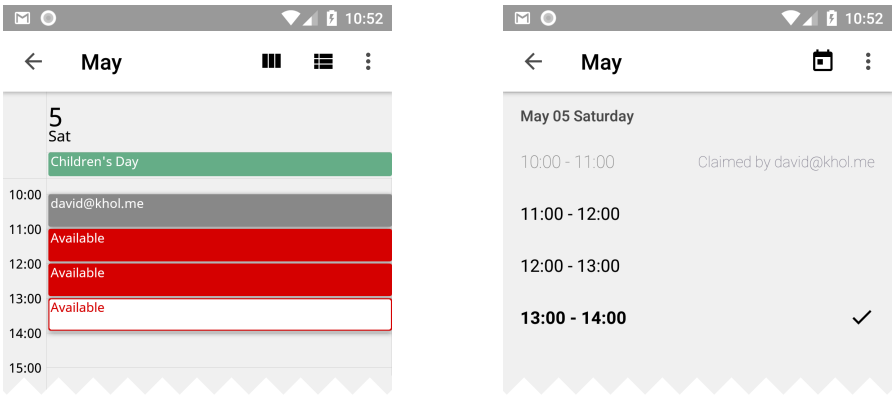


Figure 3.17: A sharing screen.

Because multiple people can claim a time-slot for themselves, we must disallow the user to choose a time-slot already claimed by someone else. In

both situations, claimed time-slots are grayed out and contain a name of the claimant. Clicking on a claimed event does nothing. Clicking on an unclaimed time-slot highlights the current selection.

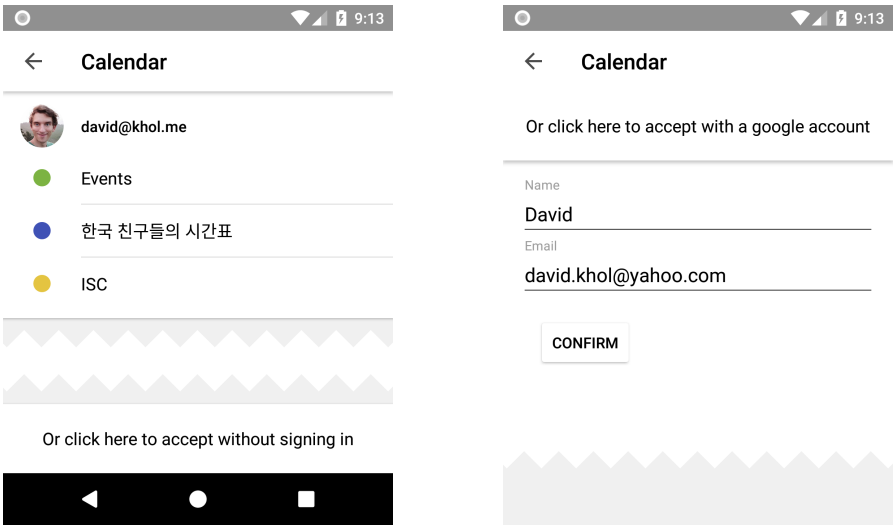


Figure 3.18: A calendar screen.

Choosing a calendar which the event should be saved to works similarly to the *calendar selection screen* from previous section 3.10.

But this time there is an extra option to accept the invitation without signing in. Clicking on the button below user's calendars replaces contents of the screen with two input fields. User may just write down his name and his email address. This information will be used to accept the meeting. The owner of the meeting will be notified just the same, but because we didn't receive permissions to write into the user's calendar, we cannot add the event to his calendar. A file in a *iCalendar* format containing information about the event is sent to the user instead.

3.3.3 Overview

Previous two flows were almost identical to the web version as they tried to match the flow as closely as possible. Both cases described a use-and-forget scenario. Once the user completes his task, he doesn't need the application any more because all further actions take place outside of the app.

The functionality of the main screen gives the user a reason to keep the app installed. On the main screen, he can list through and review previously created meetings. He may not change any details of those meetings, but he may send out more invitations if he wishes to.

Another reason for the user to keep the app installed is that he may choose not to respond to an invitation immediately, but postpone it for later. When

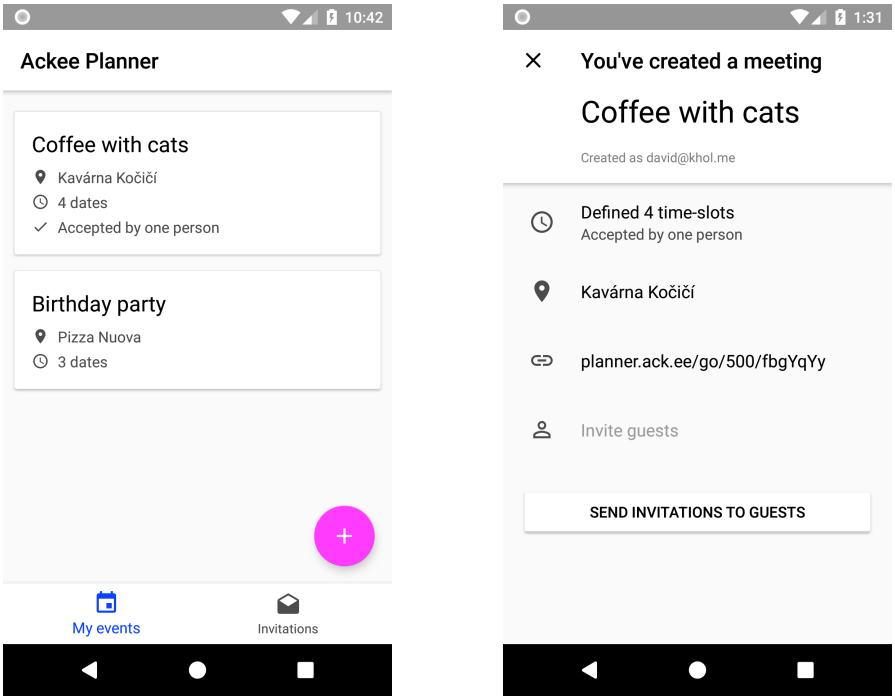


Figure 3.19: Main screen with two previously defined meetings and a detail screen of one such meeting.

he does so, he can just open the app later, click on the meeting card and fill in the details. Accepted invitations are marked with a check icon. When he opens a detail of an already accepted invitation, the submit button is unavailable because he should not be able to accept the same event twice.

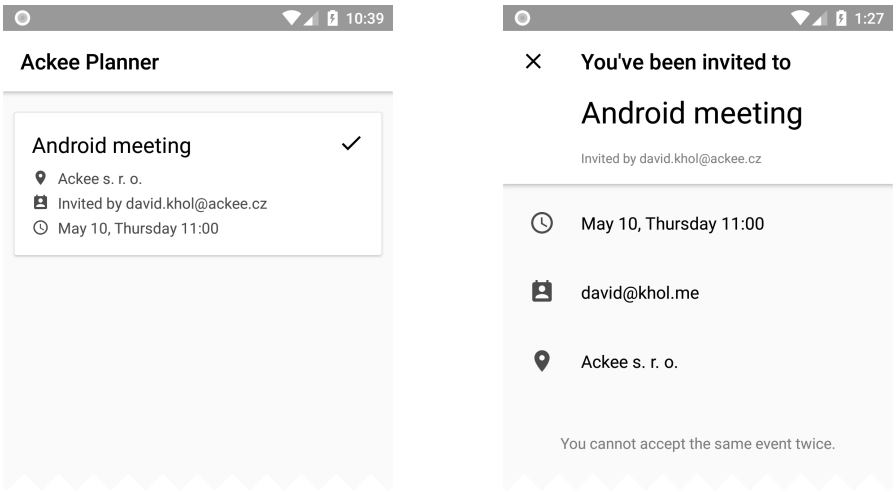


Figure 3.20: Main screen with an overview of previously received invitations and detail screen of an accepted invitation.

Chapter 4

Testing

"Elaborate usability tests are a waste of resources. The best results come from testing no more than 5 users and running as many small tests as you can afford." [43]

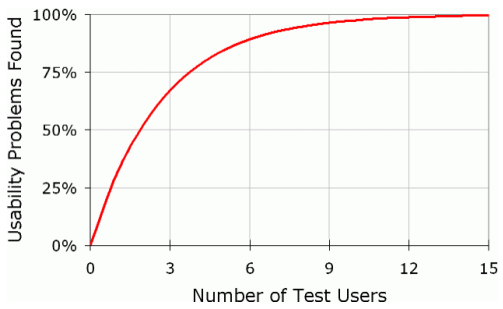


Figure 4.1: Diminishing returns of user testing.

Figure 4.1 shows diminishing returns of user testing. Testing with only two people is usually enough to find 50% of usability problems and testing with five users results in discovering of 85% of the problems.

I have prepared following testing scenarios to evaluate the usability and comprehensibility of the app.

The first tested user was not very proficient in using a smartphone. During the testing, she said she has never read an email on her phone and that she doesn't use any digital calendar services and uses a physical calendar instead.

The other tested user was familiar with smartphones but not with Android in particular. There was a small misunderstanging about navigation that was unrelated to the Ackee Planner.

4.1 First task

Because most of the people get to know about the app for the first time when they receive an invitation, first test is focused on accepting of an invitation. For every questioned user I have created a fictional meeting called 'Coffee with cats' for which I have specified multiple dates during several days. Then I have sent the invitation via an email, told them to read the contents of the email and let them proceed however they want.
Here is a list of observed issues:

- Invitation email and the app are not translated to Czech language.
- After being redirected from the email for the very first time the app crashed.
- She said it makes more sense to choose calendar first and then to choose a time-slot. In the current implementation the order is opposite.
- When she opened the screen where she should select a time-slot, she thought she is in charge of defining time-slots and didn't realize she should select one. She tried to define her own time-slots by clicking on empty spaces which, as expected, did nothing.
- I accidentaly showed her that she scroll horizontally to switch between days. I am not sure if she would realize it by herself. It is possible that she would think that events she saw were the only ones available.
- She tried to scroll about 10 days to see if there are any more events available. She didn't know there is an option to display full week at once and another option to display list of events in a textual form.
- In general, she didn't understand how permissions work. She successfully granted permission to calendar when a full-screen permission dialog showed up. Later during the testing it turned out that she thought she had given the permission to the host of the meeting. She didn't realize that the owner of the app was given the permission instead.
- I created a new meeting again and told her to accept the invitation without granting any permissions. It turned out that the button on the bottom of the screen 3.18 is too subtle to be noticed. When I showed her the other screen, she didn't understand what are the 'Name' and 'Email' fields for. She didn't complete the task.
- The other user didn't notice that he can change displayed day by scrolling.
- When he was supposed to choose one of his calendars he clicked on the desired calendar twice. The first click initiated a sign in process but the

second click cancelled it. Although the app shows a message when the sign in process is cancelled, it would be better if such situations didn't happen at all.

4.2 Second task

Second task was to create and share a meeting. User was asked to define its title, time-slots, calendar and location and send an invitation to another person.

Here is a list of observed issues:

- She succesfully wrote the title, logged in and defined time-slots, but struggled with changing of the location.
- On the guest screen she ignored an inlined permission (similar style to figure 3.10). She was confused why she has to write full email address to send an invitation to another person. She didn't understand we cannot show her suggestions unless she grants the permission.
- She didn't know she can send an invitation to multiple people.
- He though he has to attend all time-slots he had defined in the calendar View. He didn't realize that guest will accept only one of the time-slots.
- He asked if he can change the duration of the event.
- When he granted a permission to access his contacts an on-screen keyboard didn't appear. He had to click in the input field again to start writing names.

Most of the issues discovered during the testing were related to UX design of the app. During the testing we encountered two crashes that were inspected and fixed. How to resolve UX issues will be consulted and addressed with a help of professional UX designers.

Conclusion

I have familiarized myself with iCalendar protocol, learned how it can be used and understood its capabilities. In the end, iCalendar protocol was not implemented in the Android version of the app because invitations to meetings are already sent out by the backend.

During the research phase I have identified several shortcomings of the existing backend implementation. Some of the shortcomings are simple to address, such as a missing description of meetings, but some require more work, such as adding new endpoints to download all events from a specific user.

I have spent a lot of time analyzing what is the best way to display events in a calendar view and tried to implement an advanced algorithm myself. I concluded that an ultimate algorithm to lay views out perfectly does not exists because it turned out there are situations where a trade-off has to be made.

Finally, I have performed user testing with several people. I have acquired a lot of feedback as the result of the testing.

Overall, in my opinion, the app is a success. I have incorporated features that were not originally part of the assignment as it was supposed to only mimic the functionality of the web version. But the development of the app is not over yet, mainly due to limitations imposed by backend and other things beyond my expertise. Here is a list of things that could be developed next or further improved:

- Add a support for *Android Instant Apps*. Such apps allow users to run an application instantly, without prior installation. A user would be able to click on a link in an invitation email and the app would automatically open even if he didn't install it before. Current solution requires the user to download the application from the Google Play store in advance. *Ackee Planner* is a perfect candidate for such functionality. Most users will usually spend only a few minutes in the application.

Rather than undergoing a hassle of installing the app on their phones, many users may decide just to fulfill the task on a desktop instead.

Unfortunately, adding the support is not a trivial task because it requires a lot of changes to a code structure and a very small package size. Moreover, existing web service would need to be altered as well because of the way how navigation within Instant Apps works. Heavy refactoring of the code would be inevitable.

- At the time when the backend API was designed, its only consumer was the web version. A mobile version was not taken into consideration. The current backend is designed purely for the web version and many useful functions for the Android version are not provided.

  It was not planned to retrieve a list of meetings the user has been invited to, neither to retrieve a list of meetings the user has created. Currently, the application stores information about meetings without any help of the backend. But when a user reinstalls the application or changes devices, this information is not retained. Moreover it is impossible to retrieve a list of users we have invited to a specific meeting before. When the user tries to send an invitation to someone who has been already invited, the backend returns HTTP code 500 – Internal Server Error.

- Due to the restricted screen space on mobile devices, the app tried to be as simple as possible without any unnecessary verbose explanations of its functionality. The user testing pointed out situations in which the app does not accurately communicate its intent. The feedback of the user testing will be consulted and addressed with a help of professional UX designers.

- The app's visual design is plain and simple. It lacks any colors and visually interesting elements. My attempt to add colors to the application ended up looking ridiculous and so I decided to let a professional graphic designer create the design.

Bibliography

[1] iCalendar.org - iCalendar Resources, Specifications and Tools. Available from: https://tools.ietf.org/html/rfc5545

[2] RFC 5545 - Internet Calendaring and Scheduling Core Object Specification (iCalendar). Available from: https://icalendar.org/

[3] RFC 4791 - Calendaring Extensions to WebDAV (CalDAV). Available from: https://tools.ietf.org/html/rfc4791

[4] WebDAV protocol explained. Available from: http://nuanceimaging.custhelp.com/app/answers/detail/a_id/12230/~/webdav-protocol-explained

[5] The Activity Lifecycle — Android Developers. Available from: https://developer.android.com/guide/components/activities/activity-lifecycle.html

[6] Activity — Android Developers. Available from: https://developer.android.com/reference/android/app/Activity.html

[7] Separation of Concerns — Effective Software Design. Available from: https://effectivesoftwaredesign.com/2012/02/05/separation-of-concerns/

[8] Understanding Component-Entity-Systems - General and Gameplay Programming - GameDev.net. Available from: https://www.gamedev.net/articles/programming/general-and-gameplay-programming/understanding-component-entity-systems-r3013/

[9] Component · Decoupling Patterns · Game Programming Patterns. Available from: http://gameprogrammingpatterns.com/component.html

[10] Presentation-Abstraction-Control. Available from: http://www.dossier-andreas.net/software_architecture/pac.html

[11] Typical EventBus Design Patterns — ThoughtWorkshop. Available from: http://timnew.me/blog/2014/12/06/typical-eventbus-design-patterns/

[12] Richards, M. *Software Architecture Patterns*. O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472, 2015. Available from: http://www.oreilly.com/programming/free/files/software-architecture-patterns.pdf

[13] 10 Common Software Architectural Patterns in a nutshell. Available from: https://towardsdatascience.com/10-common-software-architectural-patterns-in-a-nutshell-a0b47a1e9013

[14] Chapter 3: Architectural Patterns and Styles. Available from: https://msdn.microsoft.com/en-us/library/ee658117.aspx

[15] Connect to the network — Android Developers. Available from: https://developer.android.com/training/basics/network-ops/connecting

[16] Retrofit. Available from: https://square.github.io/retrofit/

[17] Android Architecture Components — Android Developers. Available from: https://developer.android.com/topic/libraries/architecture/

[18] About SQLite. Available from: https://www.sqlite.org/about.html

[19] Save data using SQLite — Android Developers. Available from: https://developer.android.com/training/data-storage/sqlite

[20] Save data in a local database using Room — Android Developers. Available from: https://developer.android.com/training/data-storage/room/

[21] Reactive Programming with JDK 9 Flow API — Oracle Community. Available from: https://community.oracle.com/docs/DOC-1006738

[22] Reactive Streams. Available from: http://www.reactive-streams.org/

[23] Reactive Streams implementations. Available from: http://www.reactive-streams.org/announce-1.0.0#implementations

[24] Akka — Akka. Available from: https://akka.io/

[25] Project Reactor. Available from: https://projectreactor.io/

[26] ReactiveX/RxJava: RxJava – Reactive Extensions for the JVM – a library for composing asynchronous and event-based programs using observable sequences for the Java VM. Available from: https://github.com/ReactiveX/RxJava

[27] MongoDB Java Driver. Available from: http://mongodb.github.io/mongo-java-driver-reactivestreams/

[28] 23. Web Reactive Framework. Available from: https://docs.spring.io/spring-framework/docs/5.0.0.M1/spring-framework-reference/html/web-reactive.html

[29] ReactiveX. Available from: http://reactivex.io/

[30] ReactiveX - Languages. Available from: http://reactivex.io/languages.html#languages

[31] Reactive Streams Specification for the JVM. Available from: https://github.com/reactive-streams/reactive-streams-jvm

[32] Backpressure (2.0) · ReactiveX/RxJava Wiki. Available from: https://github.com/ReactiveX/RxJava/wiki/Backpressure-(2.0)

[33] ReactiveX - Operators. Available from: http://reactivex.io/documentation/operators.html

[34] Reduction (The JavaTM Tutorials ¿ Collections ¿ Aggregate Operations). Available from: https://docs.oracle.com/javase/tutorial/collections/streams/reduction.html

[35] ReactiveX - Scheduler. Available from: http://reactivex.io/documentation/scheduler.html

[36] Schedulers in RxJava — Baeldung. Available from: http://www.baeldung.com/rxjava-schedulers

[37] ReactiveX - Observable. Available from: http://reactivex.io/documentation/observable.html

[38] Ackee Planner. Available from: https://planner.ack.ee/

[39] OAuth 2.0 – OAuth. Available from: https://oauth.net/2/

[40] Permissions Overview — Android Developers. Available from: https://developer.android.com/guide/topics/permissions/overview#permission-groups

[41] Profile Your Layout with Hierarchy Viewer — Android Developers. Available from: https://developer.android.com/studio/profile/hierarchy-viewer

[42] Debug Your Layout with Layout Inspector — Android Developers. Available from: https://developer.android.com/studio/debug/layout-inspector

[43] Nielsen, J.; Landauer, T. K. A mathematical model of the finding of usability problems. 1993: pp. 206–213, doi:https://doi.org/10.1145/169059.169166.

Appendix A

List of abbreviations

ADB Android Debug Bridge

API Application Programming Interface

DSL Domain-Specific Language

HTTP Hypertext Transfer Protocol

JSON JavaScript Object Notation

OS Operating System

REST Representational State Transfer

SDK Software Development Kit

SQL Structured Query Language

UI User Interface

UX User Experience

URL Uniform Resource Locator

XML Extensible Markup Language

Appendix B

Contents of enclosed CD

- `readme.txt` ..... the file with CD contents description
- `apk` ..... the directory with executable android package
- `src` ..... the directory of source codes
  - `ackee-planner` ..... implementation sources
  - `thesis` ..... the directory of LaTeX source codes of the thesis
- `text` ..... the thesis text directory
  - `DP_Khol_David_2018.pdf` ..... the thesis text in PDF format