Contents

1 Modern Android code

This chapter illustrates the various steps that we took to migrate the Dicio Android application to new and modern technologies. The original codebase dated back to 2019 and did not respect good Android app design standards even from back then. The migration took just ≈ 80 hours to complete.

We made these changes: we converted the codebase from Java to Kotlin (section 1.1), we migrated the UI from the imperative Android Views to the declarative Jetpack Compose and picked up the MVVM architecture (section 1.2), we introduced dependency injection to better separate components and provide state to composables (section 1.3), we ditched SharedPreferences in favour of DataStore (section 1.4), we applied a few build system overhauls (section 1.5). The last section, 1.6, explains the implementation of a particularly complicated component: the Speech To Text button.

1.1 From Java to Kotlin

The only programming language available for Android development was Java up until 2017, when Google announced plans to also support Kotlin [1]. Kotlin really took off within the developer community only after 2019, when Google adopted a Kotlin-first approach [2].

Since the first commit to the Dicio codebase dates back to 2019, and Kotlin was not widespread yet, back then we naturally chose Java as the programming language for the project. In more recent years, though, the advantages of Kotlin over Java have become apparent [3]:

- Kotlin avoids `NullPointerException`s since nullable types are embedded into the type system. For example `Type?` can take values of type `Type`, or the value `null`. To turn a `Type?` into a `Type` the programmer needs to explicitly assert non-nullity (e.g. `obj!!`) or use a safe call (e.g. `obj?.doSomething()`, which calls `doSomething()` only if `obj != null`). Instead Java has no built-in null safety, which often leads to crashes, although `@Nullable` and `@NonNull` annotations can help.

- Kotlin has a more powerful function system than Java, with inline lambdas, propereties (which automaitcally create getters and setters), operator overloading, extension functions.

- Kotlin also has various shortands for creating singleton classes, data classes, union types.

- Kotlin compiles to JVM bytecode just like Java, and an effort was made to make Java and Kotlin almost fully interoperable, allowing to migrate apps one file at a time.

- Kotlin does not have checked exceptions, which is kind of a disadvantage, but speeds up development.

Therefore the first step in migrating Dicio to modern techologies was to convert all of the code from Java to Kotlin. Fortunately this could be done mostly automatically thanks to the tools provided by Android Studio, leaving only a few errors and broken references to solve.

Furthermore, the old Dicio codebase used a Reactive Programming framework named RxJava3 [6] to achieve concurrency. This was needed, for example, to analyze the user input and perform network requests without blocking the UI thread. During the migration phase it made sense to switch to Kotlin *coroutines* [5] instead, which are built into the language itself. Coroutines implement the `async/await` paradigm at the language level, making them more flexible and easy to reason about.

[1] Celebrating 5 years of Kotlin on Android, https://android-developers.googleblog.com/2022/08/celebrating-5-years-of-kotlin-on-android.html

[2] Android's Kotlin-first approach, https://developer.android.com/kotlin/first

[3] Kotlin comparison to Java, https://kotlinlang.org/docs/comparison-to-java.html

For example, here is some Java code which uses RxJava3. Its purpose is to run `skill.processInput()` in the background, and after it has finished call either `generateOutput()` or `onError()` on the main thread.

```java
Single.fromCallable(() -> {
    skill.processInput();
    return skill;
})
.subscribeOn(Schedulers.io())
.observeOn(AndroidSchedulers.mainThread())
.subscribe(this::generateOutput, this::onError);
```

Here is the equivalent Kotlin code that uses coroutines. It uses standard coding structures (e.g. `try-catch`), and makes it more explicit what is running on the UI thread.

```kotlin
val scope = CoroutineScope(Dispatchers.IO)
/* ... */
scope.launch {
    try {
        skill.processInput()
        activity.runOnUiThread { generateOutput(skill) }
    } catch (throwable: Throwable) {
        activity.runOnUiThread { onError(throwable) }
    }
}
```

1.2 From Android Views to Jetpack Compose

The traditional Android development employed an *imperative* approach to UI. The initial view tree to show on the screen was instantiated based on the structure stored in an XML file, and this tree was mutated throughout the app lifecycle via Java code. For example, to obtain a reference to a view on the screen, the Java function would have been `findViewById()`, in a similar fashion to `document.findElementById()` in HTML+JS. This setup, however, had many problems: it required developers to always work on multiple files, it made it difficult to reuse code, it was open to programmers using wrong view IDs causing crashes at runtime, and most importantly it required to keep the view tree in sync with the application state manually.

Therefore in 2019 Google introduced Jetpack Compose [3], a *declarative* toolkit for UI inspired by ReactJS and Flutter, and made it stable in 2021. In the declarative approach, the UI is a *pure function* of the application state (a so-called *`@Composable` function*), making the application state the single source of truth. Whenever the state changes, the function is called to obtain the new UI, i.e. a *recomposition* happens. While this process may seem really slow and wasteful, frameworks employ heavy optimizations to only recompose the parts of the UI corresponding to what changed in the state, making the performance on par with the imperative approach. `@Composable` functions are written entirely in Kotlin and within their body they just call other @Composable functions to build the UI as desired, allowing to easily split the code in many reusable components. The most basic components, like buttons or text areas, are usually provided by libraries, for example the Material Design 3 library.

Here is an example app with a counter that can be increased by pressing on a button (no styling is included to keep the code short):

```xml
<!-- This code would be inside res/layout/main_activity.xml -->
<LinearLayout>
    <Button
        android:id="@+id/button_id"
        android:text="Click here" />
    <TextView
        android:id="@+id/text_id" />
</LinearLayout>
```

```
// This code would be inside MainActivity's onCreate
setContent(R.layout.main_activity)
Button button = findViewById<Button>(R.id.button_id)
TextView text = findViewById<TextView>(R.id.text_id)
button.setOnClickListener(v -> {
    // assume counter is a field of the MainActivity class
    counter += 1;
    text.setText("Counter is: " + counter);
});
text.setText("Counter is: " + counter);
```

The equivalent Jetpack Compose code is much shorter and lies in only one file. Moreover, there is no manual call to `text.setText()` every time the counter changes, which the developer might forget to include.

```
@Composable
fun MainScreenWithState() {
    var counter by rememberSaveable { mutableIntStateOf(0) }
    MainScreen(counter, { counter += 1 })
}

@Composable
fun MainScreen(counter: Int, increaseCounter: () -> Unit) {
    Column {
        Button(onClick = increaseCounter) {
            Text("Click here")
        }
        Text("Counter is: " + counter)
    }
}
```

The Dicio app was originally written using the legacy Android Views and so had to be migrated to Jetpack Compose. This migration step was the one which took the longest, since it required rethinking how to handle state throughout the whole app, as the previous code had no separation between UI and business logic. For the rewritten code, however, we picked the MVVM (Model-View-Viewmodel) architecture, encouraged by Jetpack Compose's good `ViewModel` support.

Here is an example of the same `MainScreen`, but controlled by a `ViewModel`.

```
class MainScreenViewModel : ViewModel() {
    private val _counterState = MutableStateFlow(0)
    val counterState: StateFlow<Int> = _counterState.asStateFlow()

    fun increaseCounter() {
        _counterState.update { counter ->
            return@update counter + 1
        }
    }
}

@Composable
fun MainScreenWithState() {
    val viewModel: MainScreenViewModel = viewModel()
    val counter by viewModel.counterState.collectAsState()
    MainScreen(counter, viewModel::increaseCounter) // see above
}
```

This design separates the logic that increases the counter from the UI, and makes future changes much easier (some examples of future changes may be to allow resetting the counter, to control the counter via a notification, . . .). One notable thing is the use of `MutableStateFlow`, which makes it possible for the UI layer to listen to changes (via the `collectAsState()`), and also handles atomic `update` operations. Moreover, Kotlin flows support mapping, filtering and other operations on their items via coroutines, which would make it easy to e.g. create a second screen that only shows even counter values, while making sure that recompositions only happen when the even number changes.

1.3 Dependency Injection with Hilt

Dependency Injection (DI) is a design pattern used in object-oriented programming that allows an object to receive its dependencies from an external source rather than creating them itself. This promotes more modular code, eliminates boilerplate factory classes, and allows replacing real instances of services with fake ones at test time.

Dicio extensively uses the Hilt library [8] for dependency injection. Injectable classes are declared by adding the `@Inject` annotation to their constructor. Each injectable class (and a few other components, e.g. `ViewModels`, activities and fragments) can in turn have some injectable fields that will be automatically built and provided by Hilt upon instantiation. It is possible to make an injectable class a singleton with the `@Singleton` annotation, which is often useful for app-wide services that need to be accessed from multiple places.

For example, the home screen uses a Hilt view model, which depends on `SkillHandler` and `SkillEvaluator`, which in turn uses `SkillHandler` too. Moreover `SkillEvaluator` is also used in `MainActivity`. And the graph of dependencies can get even worse than this, as the codebase grows, but Hilt keeps things tidy, as the following code shows:

```kotlin
@AndroidEntryPoint // <- Hilt needs an entry point to be able to fulfill @Injects
class MainActivity : BaseActivity() {
    // automatically obtains the singleton SkillEvaluator when MainActivity is setup
    @Inject lateinit var skillEvaluator: SkillEvaluator
}

@Composable
fun HomeScreen() {
    // automatically obtains a Hilt view model for use in a @Composable
    val viewModel: HomeScreenViewModel = hiltViewModel()
}

@HiltViewModel // <- makes sure the lifecycle of the view model is handled correctly
class HomeScreenViewModel @Inject constructor(
    val skillHandler: SkillHandler,
    val skillEvaluator: SkillEvaluator,
) { /* ... */ }

@Singleton // <- this is a singleton, so Hilt will instantiate this class only once
class SkillEvaluator @Inject constructor(
    private val skillHandler: SkillHandler,
) { /* ... */ }

@Singleton
class SkillHandler @Inject constructor() { /* ... */ }
```

1.4 From SharedPreferences to Protobuf-backed DataStore

`SharedPreferences` were the standard way to store user settings in Android apps up until recently, when Google started suggesting `DataStore` [2] instead.

`SharedPreferences` were basically just an XML file on disk with key-value pairs, where the keys were strings, and the values could be any Java built-in type plus string sets. The Android ecosystem provided ways to easily read and write those values, but there were a few shortcomings, such as blocking I/O, contrived setup to listen to changes, hardcoded string keys, and no built-in way to encode enum values.

`DataStore` solves these problems by providing an API based on Kotlin coroutines, that allows getting values asynchronously and makes it simple to listen to changes. In order to make the stored values type-safe, and eliminate the need for hardcoded string keys, `DataStore` allows storing the data on disk as a `.pb` Protobuf-encoded file. Each Protobuf-encoded file corresponds to a `.proto` file processed at compilation time, which contains type and enum definintions and describes the binary representation of data on disk.

For example, this might be a `.proto` file for user settings in an app like Dicio:

```
message UserSettings {
    Theme theme = 1;
    bool show_speech_to_text_button = 2;
    map<string, bool> enabled_skills = 3;
}
enum Theme { THEME_LIGHT = 0; THEME_DARK = 1; /* ... */ }
```

Then, in the app code, the settings can be read by just accessing the `.data` field of a `DataStore` object. Listening to changes, e.g. inside a `@Composable`, boils down to manipulating the Kotlin flow:

```
val dataStore: DataStore<UserSettings> = DataStoreFactory.create(/* ... */)

@Composable
fun SttButtonIfEnabledInSettings() {
    val showSttButton = dataStore.data // this is a Kotlin flow
        .map { settings -> settings.showSpeechToTextButton }
        .collectAsState(initial = true) // value to use while settings are being loaded

    if (showSttButton) {
        Button(onClick = /* ... */) {
            Text("Speech to text button")
        }
    }
}
```

1.5 The Gradle build system

Android apps are setup and built using Gradle [4], a build system based on the JVM. Keeping Gradle build files up-to-date requires some maintenance, since very often there are new features that need to be adopted and old functionalities become deprecated.

For example, the recently introduced Version Catalogs provide a way to keep the versions of all dependencies in a single `.toml` file, which helps avoid version clashes and forgetting library updates. Therefore the dependencies of Dicio are now listed in a Version Catalog, for example:

```
# in gradle/libs.versions.toml
[versions]
datastore = "1.1.1"
[libraries]
androidx-datastore = { module = "androidx.datastore:datastore", version.ref = "datastore" }

// in app/build.gradle.kts
dependencies {
    implementation(libs.androidx.datastore)
}
```

A major overhaul of Gradle happened when the suggested language of build files changed from Groovy DSL to Kotlin DSL (where DSL stands for "Domain Specific Language"). Groovy DSL is a dynamically typed language with a rather lenient syntax, making build scripts easy to read but hard to maintain: for example, variables can be assigned without `=`, and functions can be called without `(...)`, making assignments and function calls easy to confuse. Kotlin DSL, on the other hand, is statically typed and uses the concise but strict Kotlin syntax rules. Therefore all Dicio build files were migrated from Groovy DSL (`.gradle` files) to Kotlin DSL (`.gradle.kts` files). For example, these snippets perform the same actions, but the first is in Groovy DSL and the second in Kotlin DSL:

```
minifyEnabled false
proguardFiles getDefaultProguardFile('proguard-android-optimize.txt'), 'proguard-rules.pro'

isMinifyEnabled = false
proguardFiles(getDefaultProguardFile("proguard-android-optimize.txt"), "proguard-rules.pro")
```

In order to understand what the user says, Dicio has a list of possible sentences in various languages, as described in **??** TODO ADD REF. This list needs to be directly accessible from Kotlin code for maximum performance, but it would be cumbersome for community translators to translate Kotlin files directly. Therefore the Kotlin lists are generated at compile-time based on some `.yaml` files stored under `app/src/main/sentences`. This is done by a Gradle plugin we created, `sentences-compiler-plugin`, that properly handles build task dependencies and only triggers recompilation when something changes. The plugin uses the KotlinPoet library [7] to generate correct Kotlin code.

1.6 Speech To Text button

One of the most complicated components of the Dicio app is the Speech To Text (STT) button at the bottom of the home screen. It needs to handle various operations: downloading, unzipping and loading the STT model while showing progress to the user; listening to the user speech and providing that information to other app components; handling language changes; acquiring the microphone permission; reporting errors. The button icon and label need to reflect the current state of the STT input device, so the user knows what is going on and what to expect when clicking on the button. Clicks on the button need to be perform different actions based on the current state, e.g. initiating the model download or starting to listen.

To perform STT we chose Vosk [1], a project that provides small STT models that can be run locally with good results, along with the libraries needed to actually run the models. Vosk models need to be downloaded, unzipped and loaded before they can perform STT.

In order to handle the STT button complexity correctly, we designed the state machine shown in fig. 1.1, which we implemented for the Vosk STT input device [4]. Bold arrows indicate user clicks on the button, while normal arrows indicate automatic processes (e.g. the download finishes or the STT input device detects silence and stops listening). Error states are greyed out so that they create less confusion. The *NoMicrophonePermission* state is handled separately in the UI layer, and is thus not included here.

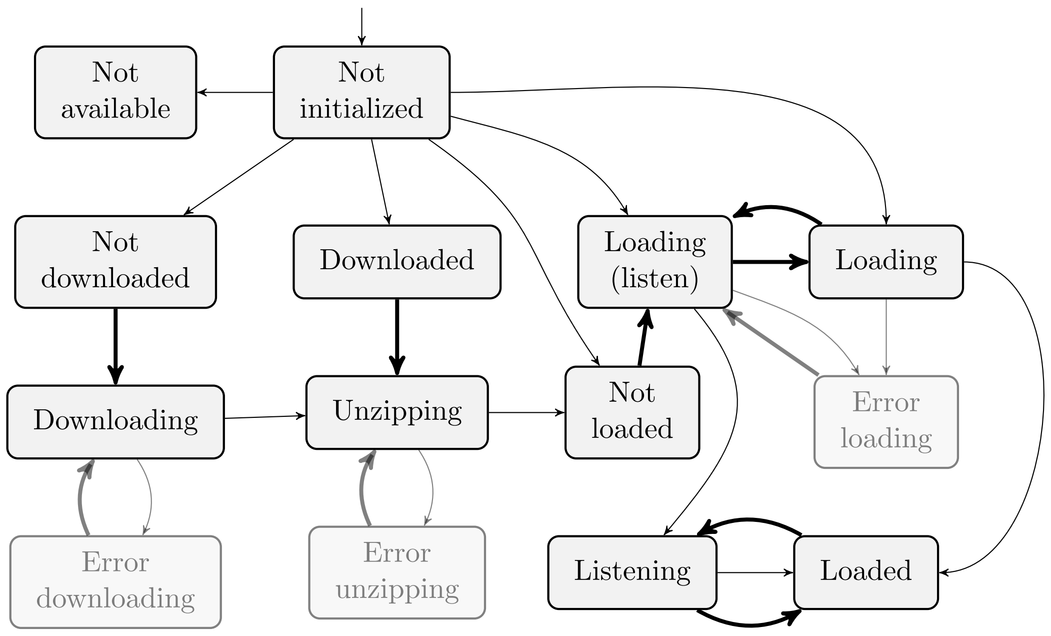

Figure 1.1: Vosk Speech To Text input device state machine

A few notes about the non-obvious state transitions:

- When the app is started, the STT input device is created with *NotInitialized* as the initial state. As soon as the app language is known, the state is set to:
  - *NotAvailable* if the STT input device does not support the current language

[4]app/src/main/kotlin/org/stypox/dicio/io/input/vosk/VoskInputDevice.kt

  - *NotDownloaded* if the previously downloaded model was not in the current language
  - *Downloaded* if a `.zip` is found on disk
  - *NotLoaded* or *Loading* if the model is found on disk (the actual state and the value of `Loading.thenStartListening` are chosen depending on the way the app was started and on user settings)
  - *NotDownloaded* otherwise

- The *Downloading* and *Unzipping* states hold information about the progress of the operation. When downloading finishes, unzipping starts directly without manual intervention.

- *Loading* comes in two possible states: with `thenStartListening` either true or false. When it is true (*Loading (listen)*), after loading has finished, the STT input device will immediately start listening. If the user clicks on the STT button while in the *Loading* state, `thenStartListening` will alternate between true and false.

- *Loaded* and *Listening* states own the loaded STT model themselves, so that even if the state changes in an unexpected way, the STT model is garbage collected along with the state object and does not create memory leaks.

- While in the *Listening* state, the STT input device reports user utterances and other events to a callback provided by the last caller of the `onClick` method (i.e. the method that the UI layer must call whenever the button is clicked). This is to allow using the STT input device from multiple places in the app at the same time, i.e. in the main screen as an input to the assistant, and in the Speech To Text service pop-up window.

Figure 1.2 shows what the button looks like for all of the states. On the first row: *NoMicrophonePermission*, *NotInitialized*, *NotAvailable*, *NotDownloaded*. On the second row: *Downloading*, *ErrorDownloading*, *Downloaded*, *Unzipping*. On the third row: *ErrorUnzipping*, *NotLoaded*, *Loading* with `thenStartListening` set to true, *Loading* with `thenStartListening` set to false, *ErrorLoading*, *Loaded*, *Listening*,

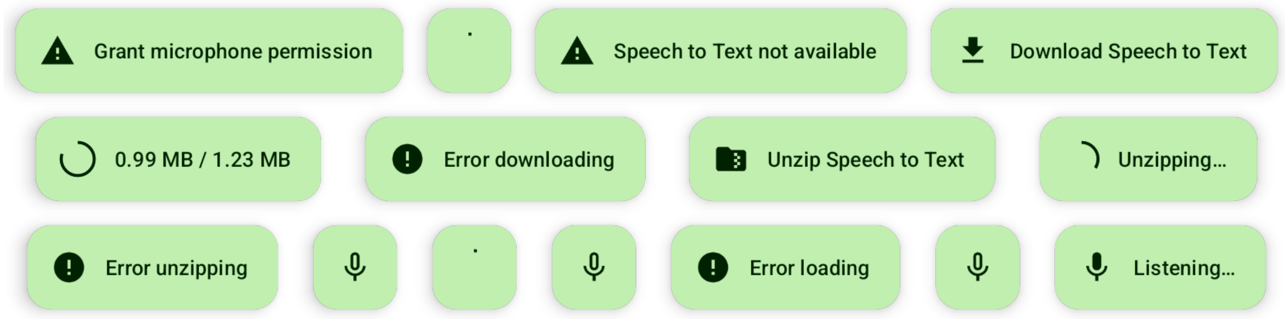

Figure 1.2: Speech To Text button previews

Bibliography

[1] Alphacephei. Vosk speech recognition toolkit. `https://alphacephei.com/vosk/`, 2019. Accessed: 2024-06-19.

[2] Google. DataStore. `https://developer.android.com/topic/libraries/architecture/datastore`, 2012. Accessed: 2024-06-18.

[3] Google. Jetpack compose. `https://developer.android.com/develop/ui/compose`, 2019. Accessed: 2024-06-14.

[4] Gradle Inc. Gradle build tool. `https://gradle.org/`, 2008. Accessed: 2024-06-19.

[5] JetBrains. Kotlin coroutines. `https://kotlinlang.org/docs/coroutines-overview.html`, 2017. Accessed: 2024-06-14.

[6] Microsoft. ReactiveX RxJava: Reactive Extensions for the JVM. `https://github.com/ReactiveX/RxJava`, 2011. Accessed: 2024-06-14.

[7] Square. KotlinPoet. `https://square.github.io/kotlinpoet/`, 2012. Accessed: 2024-06-19.

[8] Square and Google. Dagger hilt. `https://dagger.dev/hilt/`, 2012. Accessed: 2024-06-18.