Mobile Computing
Storing Data on Android

Slides: tmb.gr/mc-dat

Source on GDocs

Overview

These slides show how to store app data on Android.

How to persist preferences and write to shared files.

How to use the *Room* database abstraction library.

How to get a *Flow* of changes to update the UI.

Prerequisites

Have some basic knowledge of writing Kotlin code.

Finish the lesson on managing state on Android.

Bring your Android device or use the emulator.

Data and file storage .html

Android uses a file system offering these options:

App-specific storage, private data in internal files.

Share preferences, simple data in key-value pairs.

Shared storage, for documents that can be shared.

Databases, for private, relational application data.

App-specific files .html

Files that other apps don't need to or shouldn't access, stored in internal & encrypted or external directories, which are (both) removed when uninstalling the app.

```
context.openFileOutput(filename,
  Context.MODE_PRIVATE
).use { it.write(fileContents.toByteArray()) }
```

```
context.openFileInput(filename).bufferedReader(
).useLines { lines -> Log.v(lines) }
```

Shared preferences

.kt|.html

SharedPreferences store key/value pairs in a file.*

```
var score = 0 // a value to be persisted

val prefs = context.getSharedPreferences(
  "prefs", Context.MODE_PRIVATE)

prefs.edit().putInt("score", score).apply()

score = prefs.getInt("score")
```

*A recent alternative is DataStore.

Shared storage .html

Shared storage saves user data that can or should be accessible to other apps and saved even if the app is uninstalled, e.g. sensor data exported to a CSV file.

To create and use files, apps interact with document providers, including external and cloud storage, via the storage access framework, by launching intents.

Creating a shared document .kt | .html

Use an *Intent* to ask the user to pick a file location.

```
val exportToCsvIntent = Intent(
  Intent.ACTION_CREATE_DOCUMENT
).apply {
  addCategory(Intent.CATEGORY_OPENABLE)
  type = "text/csv" // mime type
  putExtra(Intent.EXTRA_TITLE, "export.csv")
}
```

Writing a shared document .kt|.html

```
var l = rememberLauncherForActivityResult(
  contract = ….StartActivityForResult()) {
    result -> val uri = result.data?.data; … }

l.launch(exportToCsvIntent) … // user picks uri

app.contentResolver.openOutputStream(uri).use {
  out -> out.bufferedWriter(charset =
    Charsets.US_ASCII).use { w ->
    w.write("ID, Name, Surname\n"); … } }
```

Hands-on, 10': CSV export

Extend the code, *commit* and *push* changes.

- Update your private repository (see these slides).
- Open the *MyCSVExportApp* in your repository */03* which writes a shared document as sketched before.
- Add *Space*, ' ' as an option for separator characters.
- Change the *Toast* to also show exception messages.

Open files in a CSV app or use a tool like OpenMTP.

App architecture .png|.html

An app architecture defines the boundaries between parts and the responsibilities each part should have.

Separation of concerns is the guiding principle, e.g. when an app is split into a UI layer and a data layer.

Data should drive UI, from a single source of truth, e.g. in an "offline-first" app, from a local database.

Data layer

.png | .kt | .html

The data layer contains app data and business logic, i.e. rules on how data is created, stored, and changed.

Separating it from the UI allows the data to be shared among screens and enables testing of business logic.

Room database .png|.html

Room provides an abstraction layer over SQLite DBs, including compile-time verified SQL, annotations to generate code and streamlined database migration.

```
val dao = db.getPersonDao() // get data access
val p = dao.getPerson(id) // get person entity
dao.update(person =         // update their name
  p.copy(name = "Adele")) // copy other fields
```

Room uses KSP

.toml|.kts|.html

Add the KSP* *plugin* to the project's *build.gradle.kts*

```
id("com.google.devtools.ksp") version
"2.0.21-1.0.28" apply false
```

Make sure the first part matches the Kotlin version, which is included in the project's *libs.versions.toml*

```
kotlin = "2.0.21"
```

*Kotlin Symbol Processing, used to generate code.

Room dependencies

.kts|.html

Add these *dependencies* to the app's *build.gradle.kts*

```
val v = "2.8.1" // room_version, see releases
implementation("androidx.room:room-runtime:$v")
implementation("androidx.room:room-ktx:$v")
ksp("androidx.room:room-compiler:$v")
```

Database entities

Define a class per table, create an instance per row.

```
@Entity(tableName = "person") // *
Data class PersonEntity(
  @PrimaryKey(autoGenerate = true)
  val id: Int, // or Long
  @ColumnInfo(name = "name")
  val name: String?)
```

*For table names, consider using singular nouns.

Data access object .kt|.html

Define an interface to access a table / the database.

```
@Dao
interface PersonDao { // _Impl.kt is generated
  @Query("SELECT * from person WHERE id = :id")
  fun getPerson(id: Int): Flow<PersonEntity>
  @Insert suspend fun insert(p: PersonEntity)
  @Update suspend fun update(p: PersonEntity)
  @Delete suspend fun delete(p: PersonEntity)
}
```

Database instance

.kt|.html

Extend *RoomDatabase* to provide a (generated) DB.

```
@Database(entities = [PersonEntity::class], …)
abstract class PersonDb : RoomDatabase(
) { // _Impl.kt is generated
  abstract fun personDao(): PersonDao
  … { fun getInstance(c: Context): PersonDb {
        Room.databaseBuilder(context = c,
          PersonDb::class.java, "person_db") …
  … } } }
```

Repository pattern

.kt|.html

Decouple client code from how the data is accessed.

```
interface PersonRepo { … } // is a pattern

class LocalPersonRepo(private val dao:
PersonDao) : PersonRepo {
  override fun getPersonFlow(id: Int):
    Flow<PersonEntity?> = dao.getPerson(id)
  override suspend fun insertPerson(person:
    PersonEntity) = dao.insert(person) … }
```

App container

.kt|.html

Combine a specific repo, database and DAO instance.

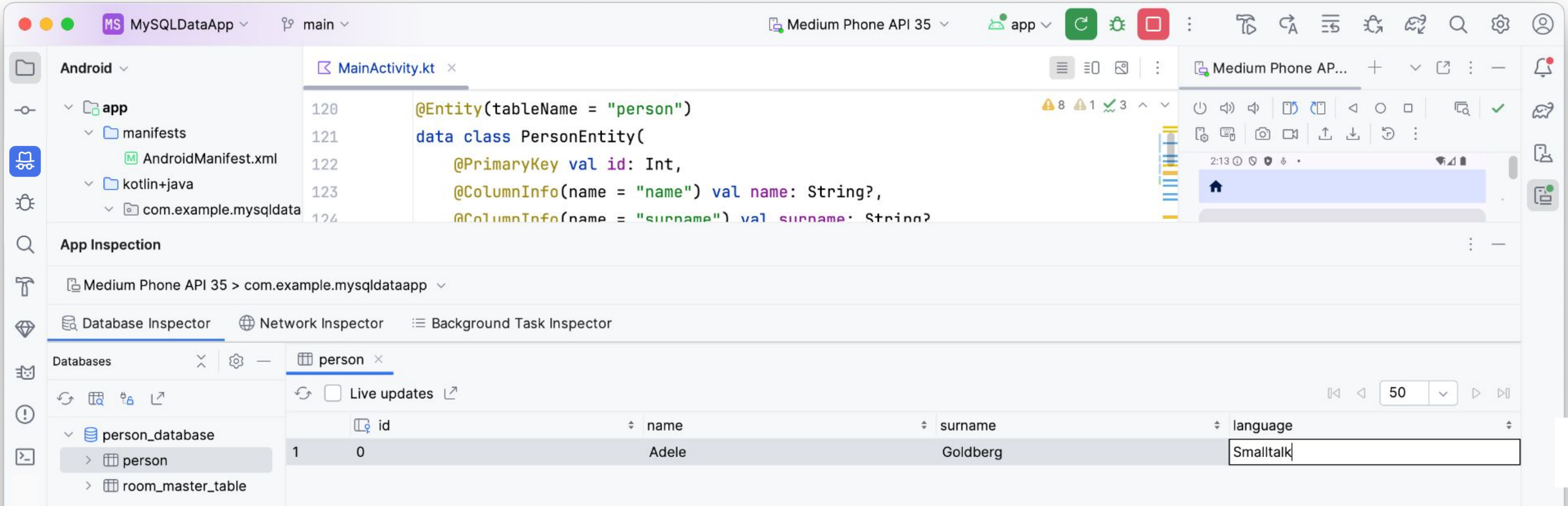
```
interface AppContainer { val personRepo: … }

class DbAppContainer(private val c: Context
) : AppContainer {
  override val personRepo: PersonRepo by lazy {
    LocalPersonRepo(PersonDb.getInstance(
      context = c).personDao())
    } }
```

Database inspector .html

Inspect, query and modify an app's MySQL database.

`View > Tool Windows > App Inspection`



```
@Entity(tableName = "person")
data class PersonEntity(
    @PrimaryKey val id: Int,
    @ColumnInfo(name = "name") val name: String?,
    @ColumnInfo(name = "surname") val surname: String?
```

person

	id	name	surname	language
1	0	Adele	Goldberg	Smalltalk

Hands-on, 10': Data layer with Room

Extend the code, *commit* and *push* changes.

- Open the *MySQLDataApp* in your repository */03* which implements a *PersonDao* as sketched before.
- Add a property *language* to the *PersonEntity* class.
- Make sure the field is added to the database table.*
- Use the database inspector to add some new rows.

*Ignore the ViewModel and UI client code for now.

UI layer

.png|.kt|.html

The user interface displays application data on screen and serves as the primary point of user interaction.

When data changes, due to user interaction or external input like a network response, the UI should update.

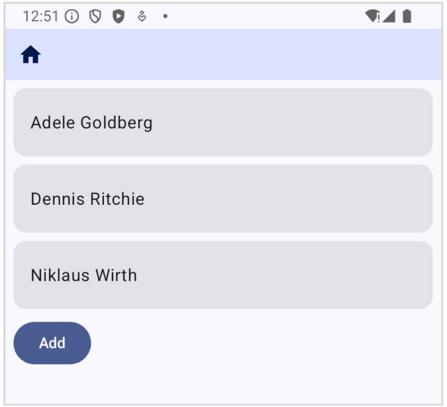
The UI layer converts application data changes to a form that the UI can present and then displays it.

Screens

.kt|.html

A typical UI for a database app offers these screens.*

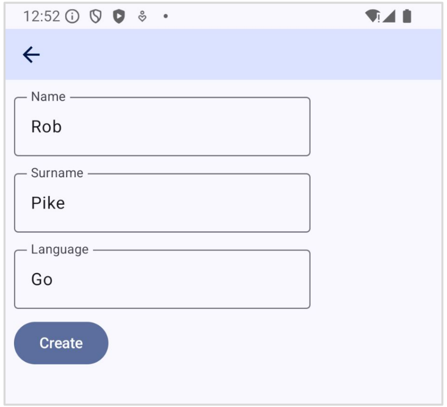
```
enum class Screen { LIST, ENTRY, DETAIL, EDIT }
```


12:51

Adele Goldberg

Dennis Ritchie

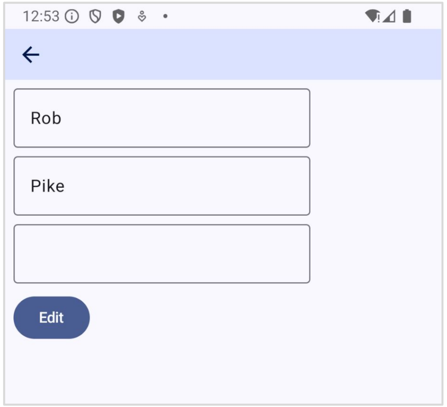
Niklaus Wirth


12:52

Name: Rob

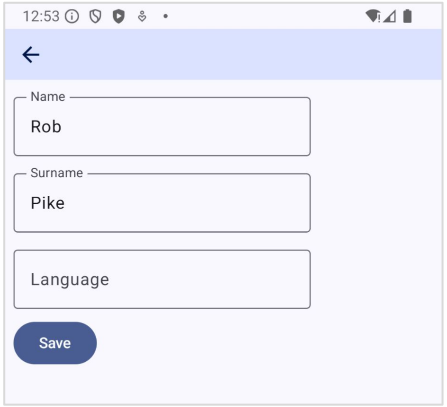
Surname: Pike

Language: Go


12:53

Rob

Pike


12:53

Name: Rob

Surname: Pike

Language

*Depending on the use case, list items show details.

Navigation

.kt|.html

Use *when* for prototypes, *NavController* for products.

```
when (screen) {
  Screen.LIST -> ListScreen(
    onAdd = { screen = Screen.ENTRY },
    onOpen = { it -> personId = it
      screen = Screen.DETAIL },
    modifier)
  Screen.ENTRY -> EntryScreen(personId, …) …
}
```

ViewModels

.kt | .html

TODO*.

```
code()
```

*TODO.

Coroutines

.kt|.html

A coroutine is a concurrency design pattern that helps to simplify asynchronous code with *suspend/launch*.

On Android, coroutines help to manage long-running tasks that would block the main thread, slow the app.

```
class …ViewModel { suspend fun createPerson() }

scope.launch { viewModel.createPerson() }
```

Flows

.kt | .html

In coroutines, a *Flow* is a type that can emit multiple values sequentially, as opposed to suspend functions that return only a single value. For example, you can receive live updates from a database or a Web API.

```
val latestNews: Flow<List<ArticleHeadline>> =
flow {
  while(true) { emit(newsApi.fetchNews()) }
}
```

Hands-on, 10': UI layer with Flows

TODO*

*Try ...

Summary

These are the basics of storing app data on Android.

Storing key/value entries using *SharedPreferences*.

Storing local data using the *Room* database library.

Propagating a single source of truth with a *Flow*.

~~Challenge: TODO~~

Work through the TODO codelab.

- Start from this BasicTODOCodelab app project.
- Add the *project files* to your private repository.
- Make sure *not* to add the 3rd-party *repository*.
- Git *commit* and *push* your code to your repo.

Done? There are more codelabs, e.g. on TODO.

Feedback or questions?

Write me on Teams or email

thomas.amberg@fhnw.ch

Thanks for your time.