Data Layer

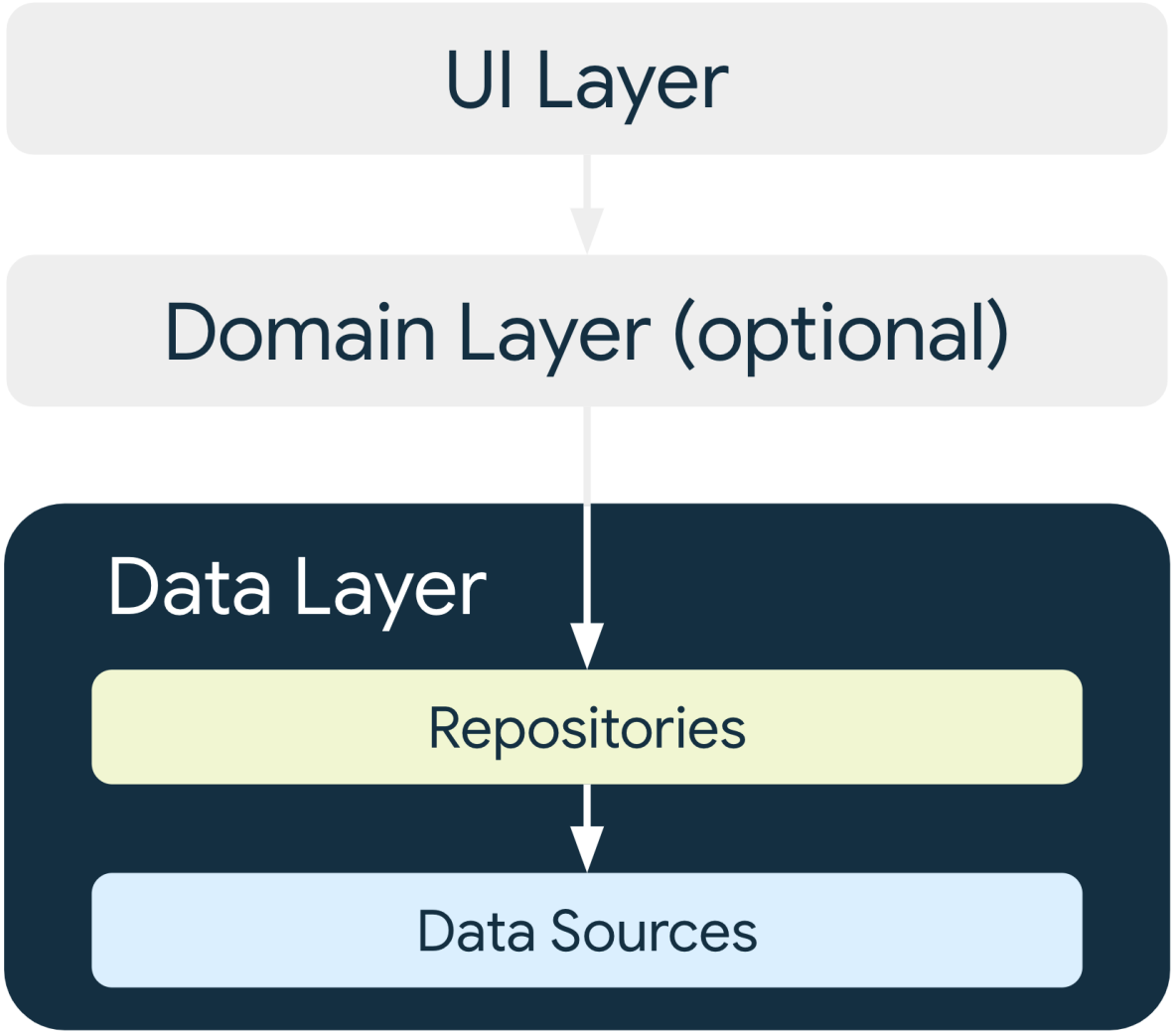
Repository pattern, Data Store

prof. dr Bratislav Predić
dipl. inž. Nevena Tufegdžić

Razvoj mobilnih aplikacija i servisa
Računarstvo i informatika

2024.

Data Layer



prof. dr Bratislav Predić
dipl. inž. Nevena Tufegdžić

Repository pattern

- Design pattern koji olakšava organizaciju podataka
- Sloj između logike aplikacije i samih izvora podataka
  - Izvor podataka: baza podataka, HTTP server, itd.
- Apstrakcija izvora podataka
- *Separation of concerns*

Repository class

- Repository pattern se u Kotlinu postiže pomoću Repository klasa
- Repository klasa sadrži funkcije za preuzimanje podataka iz izvora podataka
- Centralno mesto za unošenje promena u perzistenciju
- Apstrakcija izvora podataka

Repository - pravila

- Repository može komunicirati sa više data source klasa, ali jedna data source klasa može komunicirati isključivo sa jednim izvorom podataka
- Ostali slojevi aplikacije nikada ne komuniciraju sa data source klasama – ovo se vrši posredno preko Repository
- Immutability principle – svi podaci koje Repository otvara ka višim slojevima aplikacije moraju biti immutable (slično ViewModel-u)

Repository - pravila

- Data source klase pružaju pristup CRUD operacijama
- Repository treba da pruži interfejs ka ovim operacijama, kao i način da ostatak aplikacije biva obavešten o promenama u podacima (Flow)
  - Flow je tip koji emituje više vrednosti sekvencijalno (StateFlow nasleđuje Flow)
- Dependency Injection – pristup data source klasama

Repository - primer

Dependency Injection

```
class ExampleRepository(
    private val exampleRemoteDataSource: ExampleRemoteDataSource, // network
    private val exampleLocalDataSource: ExampleLocalDataSource // database
) {

    val data: Flow<Example> = ...

    suspend fun modifyData(example: Example) { ... }
}
```

Threading

- Bitna stavka kod Repository pattern-a jeste multithreading
- Potrebno je omogućiti konkurentni pristup podacima, tako da sve ostane konzistentno (read i write)
- Ovo se može postići pomoću više niti, ali Kotlin nudi bolje rešenje u obliku korutina (coroutine)

Korutina

- Korutina je projektni obrazac za asinhrone pozive funkcija
- U jednoj niti može se izvršavati više korutina, zahvaljujući suspension mehanizmu
  - Suspension mehanizam dozvoljava neblokirajuće operacije – nit u kojoj se suspendable funkcija izvršava ne biva blokirana usled izvršenja te funkcije
- Korutina se može pauzirati u jednoj niti, a zatim nastaviti u drugoj koja je tada slobodna
- Integrisane u Jetpack biblioteku – dobra kombinacija sa Jetpack Compose bibliotekom

Korutina - primer

```
fun main() = runBlocking { // this: CoroutineScope
    launch { // launch a new coroutine and continue
        delay(1000L) // non-blocking delay for 1 second
        println("World!") // print after delay
    }
    println("Hello") // main coroutine continues while a previous one
is delayed
}
```

- Rezultat:

  Hello

  World

Korutina

- **launch** – coroutine builder – služi za instanciranje nove korutine, koja će se izvršavati konkurentno sa ostatkom koda
- **delay** – suspending funkcija – suspenduje (pauzira) korutinu na određeno vreme
  - Suspenzija korutine ne blokira nit u kojoj je lansirana, tako da druge korutine u datoj niti mogu da se izvršavaju
- **runBlocking** – još jedan coroutine builder – otvara CoroutineScope, i samo u okviru njega je moguće lansirati korutine
  - Pozivanje launch van CoroutineScope-a daje compile error

Suspend funkcije

- Funkcije koje u sebi mogu pozivati suspending funkcije (npr. delay)
- Ključna reč suspend

```
suspend fun doWorld() {
    delay(1000L)
    println("World!")
}
```

Suspend - primer

```
fun main() = runBlocking { // this: CoroutineScope
    launch { doWorld() }
    println("Hello")
}

// suspending function
suspend fun doWorld() {
    delay(1000L)
    println("World!")
}
```

- Rezultat:

  Hello

  World

Launch i await

- **Launch** kreira i pokreće korutinu, ali ova korutina ne vraća rezultat
- Async je način kreiranja i pokretanja korutine koja vraća rezultat
- Ako je korutina pokrenuta sa async, onda se nad njom može pozvati await kako bi se sačekali rezultati

Async - primer

```
suspend fun fetchTwoDocs() =
    coroutineScope {
        val deferredOne = async { fetchDoc(1) }
        val deferredTwo = async { fetchDoc(2) }
        deferredOne.await()
        deferredTwo.await()
    }
```

Korutine - smernice

- Korutine idu uz ViewModel-e – nije dobro lansirati ih iz aktivnosti
  - Koristiti korutine iz viewModelScope-a, ne iz lifecycleScope-a
- Nikako ne lansirati korutine iz Composable funkcija
- Sloj podataka i logike bi trebalo da koriste korutine za CRUD operacije, a Flow za praćenje promena u podacima
- Omogućiti da korutina može biti prekinuta u nekom trenutku i paziti na obradu izuzetaka (exception handling)

Data Store

prof. dr Bratislav Predić
dipl. inž. Nevena Tufegdžić

Data Layer
Razvoj mobilnih aplikacija i servisa

Data Store

- Način perzistentnog čuvanja podataka na Android-u
- Preferences Data Store i Proto Data Store
- Preferences Data Store skladišti podatke kao key-value parove, dok Proto Data Store koristi protocol bafere kako bi skladištila tipizirane objekte
  - Protocol buffer – standard za serijalizaciju struktuiranih podataka (nezavistan od jezika i platforme, tvorevina Google-a)

Data Store

Feature	SharedPreferences	PreferencesDataStore	ProtoDataStore
Async API	✅ (only for reading changed values, via listener)	✅ (via `Flow` and RxJava 2 & 3 `Flowable`)	✅ (via `Flow` and RxJava 2 & 3 `Flowable`)
Synchronous API	✅ (but not safe to call on UI thread)	❌	❌
Safe to call on UI thread	❌1	✅ (work is moved to `Dispatchers.IO` under the hood)	✅ (work is moved to `Dispatchers.IO` under the hood)
Can signal errors	❌	✅	✅
Safe from runtime exceptions	❌2	✅	✅
Has a transactional API with strong consistency guarantees	❌	✅	✅
Handles data migration	❌	✅	✅
Type safety	❌	❌	✅ with Protocol Buffers

Preferences Data Store

- Ažuriranje podataka se vrši u transakcijama
- Trenutno stanje podataka je otkriveno u obliku Flow objekta
- Iako radi sa transakcijama, ne postoji eksplicitni apply ili commit, već se to radi implicitno
- Ne otkriva unutrašnje stanje u obliku mutable objekata
- Pruža API za rad sa podacima, poštujući immutability principle, na način sličan radu sa Map objektima

```
implementation "androidx.datastore:datastore-preferences:1.0.0"
```

Preferences Data Store

- Instanciranje: na samom početku Kotlin fajla

```
val Context.dataStore: DataStore<Preferences> by
preferencesDataStore(name = "settings")
```

- Koristi se delegat – by preferencesDataStore
  - Funkcija koja vraća referencu na data store sa zadatim imenom
  - Kreira ako ne postoji, uzima referencu ako postoji
- `DataStore<Preferences>` je klasa koja služi za rad sa preferences data store

Čitanje podataka

```
val EXAMPLE_COUNTER = intPreferencesKey("example_counter")
val exampleCounterFlow: Flow<Int> = context.dataStore.data
  .map { preferences ->
    // No type safety.
    preferences[EXAMPLE_COUNTER] ?: 0
}
```

Čitanje podataka

- Pre svega je potrebno kreirati prefereces key – poseban objekat koji predstavlja key za pristup određenom tipu podataka (u primeru je to Int)
- Zatim se pristupa Flow objektu koji enkapsulira datu Int vrednost – svaki put kada se ova vrednost ažurira, exampleCounterFlow će imati najnoviju vrednost
- Pošto se radi o nekoj vrsti mape, potrebno je iskoristiti funkciju map kako bi se podaci „iščupali“ iz data store-a

Upis podataka

```
suspend fun incrementCounter() {
  context.dataStore.edit { settings ->
    val currentCounterValue = settings[EXAMPLE_COUNTER] ?: 0
    settings[EXAMPLE_COUNTER] = currentCounterValue + 1
  }
}
```

Upis podataka

- Koristi se intPreferencesKey za pristup vrednosti koju treba upisati ili ažurirati
- U okviru edit funkcije, kao i kod map funkcije, dostupan je ceo preferences data objekat
- Upis ili ažuriranje se svodi na pristup datoj vrednosti pomoću njenog ključa, i modifikaciju u okviru edit funkcije
- Nakon završetka edit bloka, vrednost će biti ažurirana kao da je izvršen commit u transakciji

Proto Data Store

- Zahteva definisanu šemu podataka, specifično smeštenu u app/src/main/proto/ direktorijumu
- Koristi se protobuf jezik za definisanje protokola
- Sadržaj proto fajla:

```
syntax = "proto3";
option java_package = "com.example.application";
option java_multiple_files = true;

message Settings {
  int32 example_counter = 1;
}
```

Proto fajl

- `syntax = "proto3„` je verzija protokola
- Message je zapravo šema podataka koji se upisuju u proto data store
- Int32 je int tip podataka sa 4 bajta
- Definiše se counter property, koji mora biti int, i podešava mu se početna vrednost

```
object SettingsSerializer : Serializer<Settings> {
  override val defaultValue: Settings = Settings.getDefaultInstance()

  override suspend fun readFrom(input: InputStream): Settings {
    try {
      return Settings.parseFrom(input)
    } catch (exception: InvalidProtocolBufferException) {
      throw CorruptionException("Cannot read proto.", exception)
    }
  }

  override suspend fun writeTo(
    t: Settings,
    output: OutputStream) = t.writeTo(output)
}
```

Kreiranje Proto Data Store-a

- Prvo je potrebno definisati Serializer tipiziran po klasi definisanoj u proto fajlu
- Ovom serializer-u je potrebno definisati default vrednost Settings objekta (protocol buffer vodi računa o ovome)
- Predefinišu se funkcije za čitanje i upis
  - Ovde se definiše kako čitanje i upis funkcionišu – kako se vrši serijalizacija objekta
- Na kraju se instancira proto data store na sličan način kao i preferences data store

Instanciranje

```
val Context.settingsDataStore: DataStore<Settings> by
dataStore(
  fileName = "settings.pb",
  serializer = SettingsSerializer
)
```

Čitanje

```
val exampleCounterFlow: Flow<Int> = context.settingsDataStore.data
  .map { settings ->
    // The exampleCounter property is generated from the proto schema
    settings.exampleCounter
  }
```

Upis

```
suspend fun incrementCounter() {
  context.settingsDataStore.updateData { currentSettings ->
    currentSettings.toBuilder()
      .setExampleCounter(currentSettings.exampleCounter + 1)
      .build()
    }
}
```

Literatura

- Android Data Layer
- Kotlin Coroutines
- Coroutines – dokumentacija
- Kotlin Flow
- DataStore Codelab
- DataStore - Android Docs

Hvala na pažnji!