BCS 371 Mobile Application Development I

Arthur Hoskey, Ph.D.
Farmingdale State College
Computer Systems Department

© 2024 Arthur Hoskey. All rights reserved.

Today's Lecture

- State
- State Hoisting
- ViewModel
- Repository
- Dependency Injection

© 2024 Arthur Hoskey. All rights reserved.

mutableStateOf and remember

- mutableStateOf
  - Creates an observable type integrated with the compose runtime.
  - Any changes to this observable type will trigger a recomposition of any composable functions that read the observable's value.
  - This means that it will send out notifications when its value changes.
- remember
  - The variable's value is saved through recomposition.
- For example:

mutableStateOf creates an observable type. This is then given to remember.

var num by **remember** { **mutableStateOf(0)** }

0 is the default value for the variable

mutableStateOf and remember

© 2024 Arthur Hoskey. All rights reserved.

mutableStateListOf

- Similar to mutableStateOf except it operates on a list.
- mutableStateListOf will also trigger a recomposition if an add, remove, update, etc… is done on the list (this will not happen with mutableStateOf).
- It differs from mutableStateOf since mutableStateOf only triggers a recomposition if the list's object reference changes.
- Examples for creating a mutableStateListOf:

Create from an existing list

```
var myList : List<String> = listOf("a", "b", "c")
var obsList = remember {  myList.toMutableStateList() }
```

Or

Put values in directly

```
var obsList = remember { mutableStateListOf("a", "b", "c")}
```

mutableStateListOf

© 2024 Arthur Hoskey. All rights reserved.

- Now on to state hoisting…

State Hoisting

© 2024 Arthur Hoskey. All rights reserved.

State Hoisting

- A pattern of moving state to a composable's caller.
- The goal is to make a stateless composable.
- Composables that have state are less reusable and harder to test.
- If the state is "decoupled" in this manner it makes it easier to make changes to the app.

- Taken from:

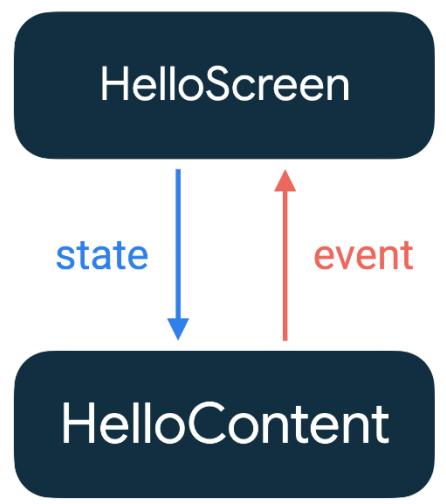
https://developer.android.com/jetpack/compose/state

State Hoisting

© 2024 Arthur Hoskey. All rights reserved.

State Hoisting

- State should be passed down.
- Events should go up.



HelloContent has parameters where the "state" is passed in (this makes HelloContent more resusable)

- Picture taken from:

https://developer.android.com/jetpack/compose/state

State Hoisting

© 2024 Arthur Hoskey. All rights reserved.

State Hoisting Example - Text

```
// State is NOT hoisted
@Composable
fun ShowMessage() {
    var message by rememberSaveable { mutableStateOf("") }
    Text(text="$message")
}
```

ORIGINAL STATE NOT HOISTED
Text and its state are defined in the same function

```
// State Hoisted (state of Text in calling function)
@Composable
fun ShowMessage() {
    var message by rememberSaveable { mutableStateOf("") }
    ShowMessage(message)
}

@Composable
fun ShowMessage(m: String) {
    Text(text="$m")
}
```

Pass state variable to the new function where the Text is now defined

Add a new function that takes the state as a parameter. The Text is now here, and it uses the parameter in its definition.

UPDATED STATE HOISTED
Split into two function. The Text's state has been hoisted up to the calling function. ShowMessge(String) is easily reusable since it does not store any state.

State Hoisting Example - Text

© 2024 Arthur Hoskey. All rights reserved.

State Hoisting Example - TextField

```
@Composable
fun ShowTextField() {
    var data by rememberSaveable { mutableStateOf("") }
    TextField(
        value = data,
        onValueChange = { data = it }
    )
}
```

TextField and its state are defined in the same function

```
@Composable
fun ShowTextField() {
    var data by rememberSaveable { mutableStateOf("") }
    ShowTextField(data, onDataChange = {data=it})
}

@Composable
fun ShowTextField(d:String, onDataChange: (String)->Unit) {
    TextField(
        value = d,
        onValueChange = onDataChange
    )
}
```

Pass both the state and the function reference used by onValueChange

TextField uses the parameters in its definition

The TextField's state has been hoisted up to the calling function

State Hoisting Example - TextField

© 2024 Arthur Hoskey. All rights reserved.

- Now on to ViewModel…

ViewModel

© 2024 Arthur Hoskey. All rights reserved.

ViewModel

- One of the new Android Architecture Components.
- Used to store **data** that will be **retained through a device configuration change**.
- **Each screen should have its own ViewModel**.
- Suggest using API 28 or higher.
- Need Gradle dependency to make it work (on an upcoming slide).

ViewModel

© 2024 Arthur Hoskey. All rights reserved.

- Add the following to build.gradle (lower-level file)

```
dependencies {
 ...

    var lifecycle_version = "2.6.2"

    // ViewModel
    implementation("androidx.lifecycle:lifecycle-viewmodel-compose:$lifecycle_version")
    implementation("androidx.lifecycle:lifecycle-viewmodel-ktx:$lifecycle_version")
}
```

Reminder!
Make sure to sync the Gradle file with the project after changing dependencies

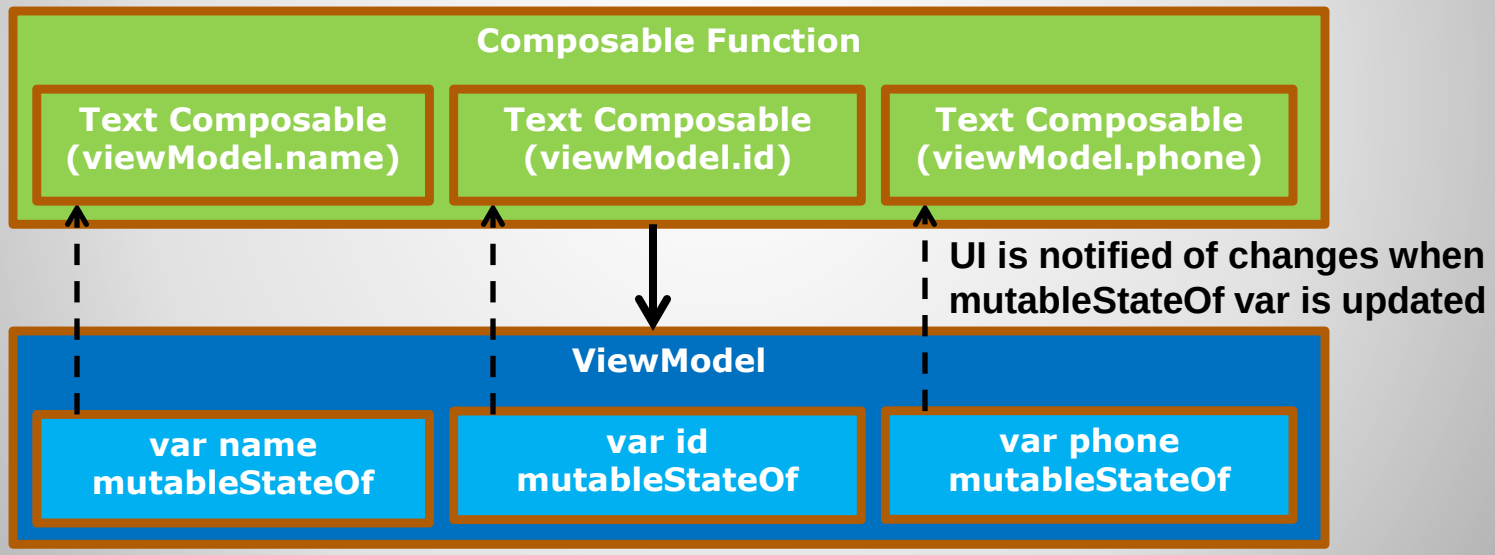
Check this link for latest dependencies:
https://developer.android.com/jetpack/androidx/releases/lifecycle

ViewModel – Gradle Dependencies

© 2024 Arthur Hoskey. All rights reserved.

ViewModel Overview

- The ViewModel stores data used by the UI.
- The Text composable refers to a variable on the view model.
- When the variable on the view model is updated, the Text composable is recomposed and will display the new value.

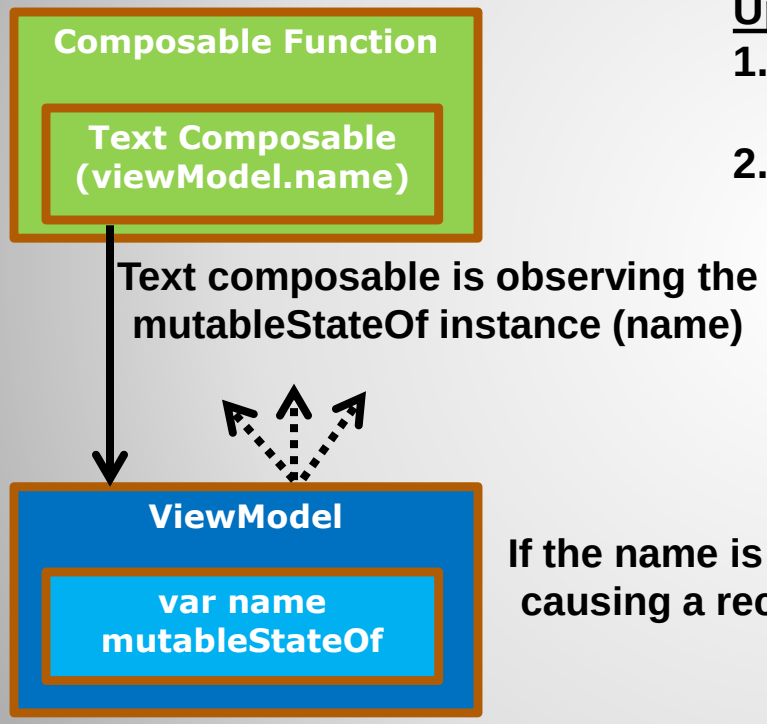


ViewModel Overview

© 2024 Arthur Hoskey. All rights reserved.

Observe mutableStateOf Variable

- mutableStateOf variable - Observable data holder.
- This variable sends out notifications when its data changes.



Update Sequence

1. **Data is changed in mutableStateOf instance**
2. **Notifications are sent out to all observers of that mutableStateOf instance**

If the name is changed a notification will be sent out causing a recomposition of the composable that is observing it

Observe mutableStateOf Variable

© 2024 Arthur Hoskey. All rights reserved.

Simple ViewModel Subclass (simple)

- Create a **ViewModel** subclass.
- This simple version does not take any parameters.
- Note: Parameters will be passed into the ViewModel when incorporating it in a more layered architecture (upcoming slides describe this).

```
import androidx.lifecycle.ViewModel

class MainScreenViewModel : ViewModel() {

    var name by mutableStateOf("")

}
```

Variables and methods for data you want to store for the composables go here

Since name is mutableStateOf, whenever its value changes, any composables that use it will be recomposed

© 2024 Arthur Hoskey. All rights reserved.

Associate ViewModel and Composable (simple)

Get the ViewModel class instance (simple version just uses the default constructor, no dependencies injected)

```
@Composable
fun mainScreen() {
    val viewModel = viewModel { MainScreenViewModel() }

    Text( viewModel.name )

    // Other composable code for GUI goes here…
}
```

Use the name variable from the ViewModel class in the Text composable

Any updates to the name variable on the ViewModel will cause a recomposition of the Text composable

Associate ViewModel and Composable (simple)

© 2024 Arthur Hoskey. All rights reserved.

Gettting a ViewModel Instance

- The ViewModel instance should only be created once (not every time there is a recomposition.
- The following ways will work (only created once):

```
val viewModel = viewModel {MainScreenViewModel() }
    OR
val viewModel : MainScreenViewModel = viewModel()
    OR
val viewModel = viewModel< MainScreenViewModel >()
```

Each of these will only create the ViewModel instance once no matter how many recompositions occur

- The line below creates a new ViewModel for each recomposition and could cause logic errors:

~~val viewModel = MainScreenViewModel()~~

Creating a new ViewModel instance for every recomposition could introduce logic errors in the app

Getting a ViewModel Instance

© 2024 Arthur Hoskey. All rights reserved.

Exposing Immutable State

- It is suggested to only expose immutable state to the composable functions (so composable functions cannot directly update the view model variables).
- Just make the set for the view model's mutableStateOf variable private.
- Here is some sample code:

```
class MyViewModel : ViewModel() {
    var num by mutableStateOf(0)
        private set

    fun setNum(n: Int) {
        num = n
    }
}
// This code is in the composable function
val viewModel = viewModel { MainScreenViewModel() }
val num = viewModel.num
Text(num.toString())
//viewModel.num = 20
```

Make set private on the mutableStateOf variable

This function can use the set because it is a member of the class

Use the view model num member variable.

This will not work because num's set is private in the view model

Exposing Immutable State

© 2024 Arthur Hoskey. All rights reserved.

AndroidViewModel Base Class

Create AndroidViewModel Subclass

- A view model class can inherit from AndroidViewModel instead of ViewModel.
- The AndroidViewModel class stores the Application instance for the app.
- The subclass must take an Application instance as a parameter and pass it to AndroidViewModel.
- AndroidViewModel is useful if code needs something from the Application instance such as the context.
- For example, Preferences Datastore needs the context so if you were using it inside a view model the application should be passed in.

```
import androidx.lifecycle.AndroidViewModel

class MyViewModel(application:Application)
    : AndroidViewModel(application)
{
   // Other view model code goes here
}
```

The MyViewModel subclass constructor takes the application as a parameter and passes it to AndroidViewModel

© 2024 Arthur Hoskey. All rights reserved.

Gettting AndroidViewModel Instance

- The following code should be run from a composable function.

Get the current context

```
val context = LocalContext.current
var viewModel = viewModel {
MainScreenViewModel(context.applicationContext as Application)}
```

Get the application from the current context and pass it to the view model constructor. The as keyword is being used because the applicationContext variable needs to be treated as an Application instance.

Getting AndroidViewModel Instance

© 2024 Arthur Hoskey. All rights reserved.

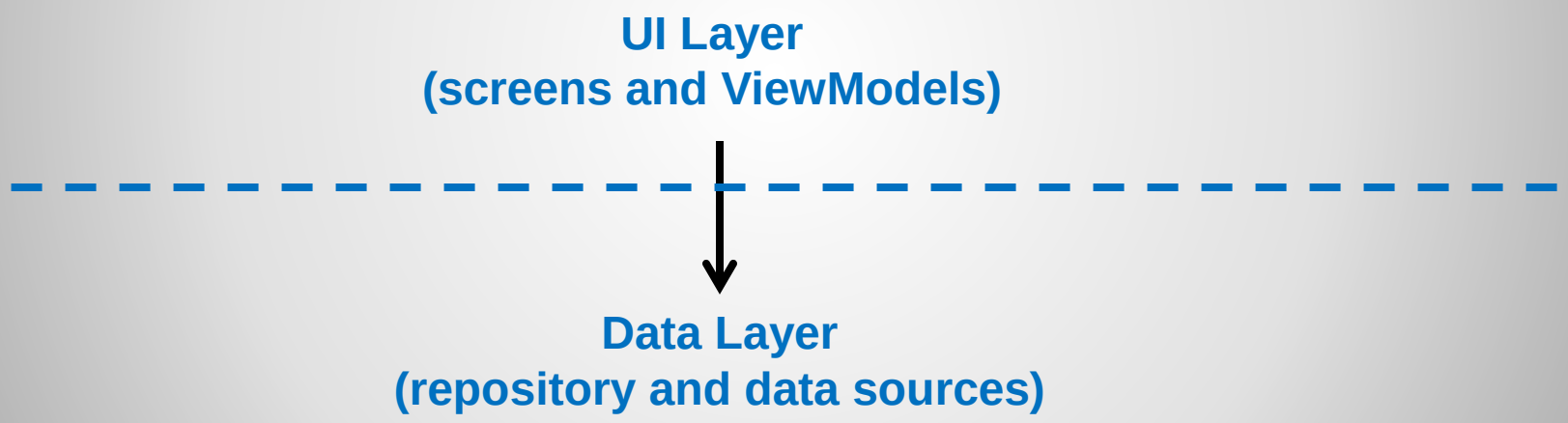
- Now on to higher-level architecture using ViewModels and repositories…

Higher-Level Architecture

© 2024 Arthur Hoskey. All rights reserved.

High-Level Architecture

- Use a layered high-level architecture.
- In general, changes to the app will be less invasive in a layered architecture.
- The code related to the UI will not be mixed with the code that stores the app's data.
- The UI makes calls into the data layer.
- The data layer has no knowledge of the UI layer.

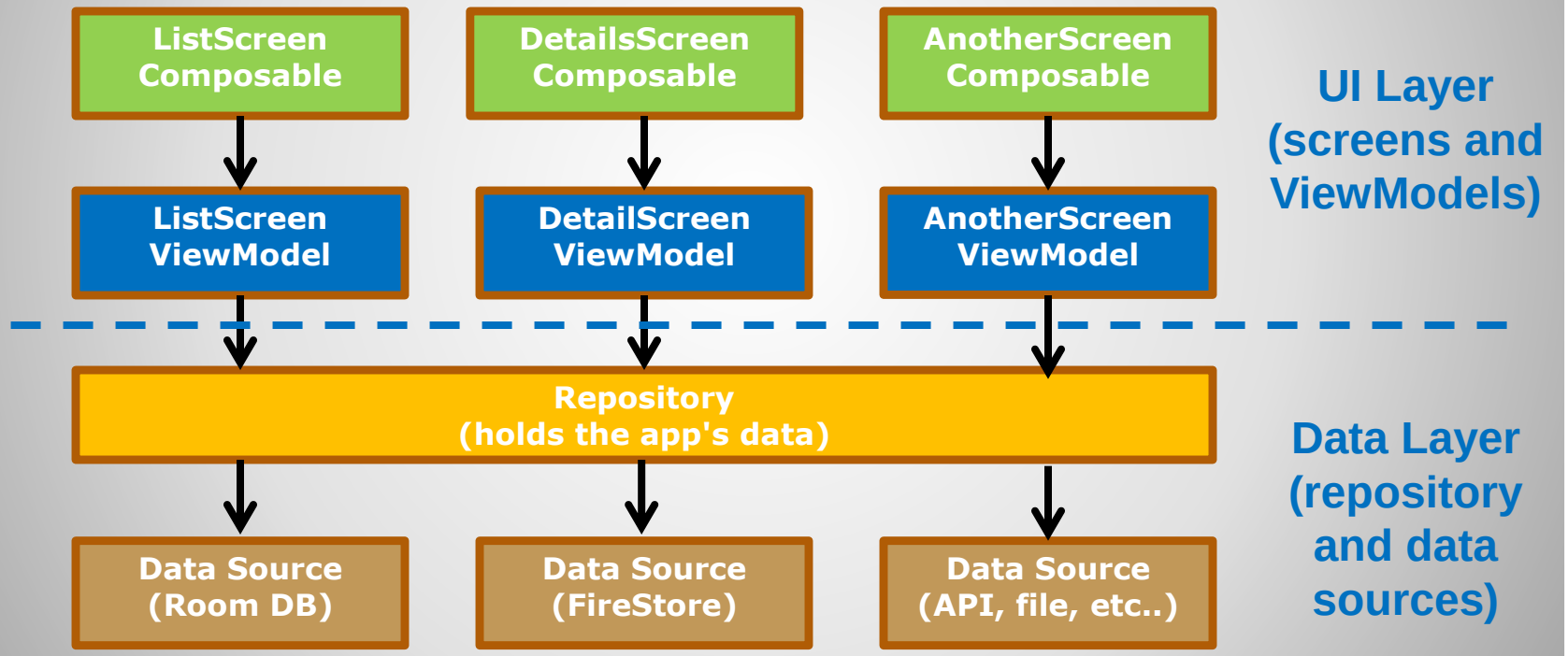


High-Level Architecture

© 2024 Arthur Hoskey. All rights reserved.

High-Level Architecture

- Each screen composable should have its own ViewModel associated with it.
- ViewModels should get the data they need from a repository.
- The repository is the single source of truth for the app's data.
- Repositories access different data sources (Firestore, Room DB, File, API, etc..)

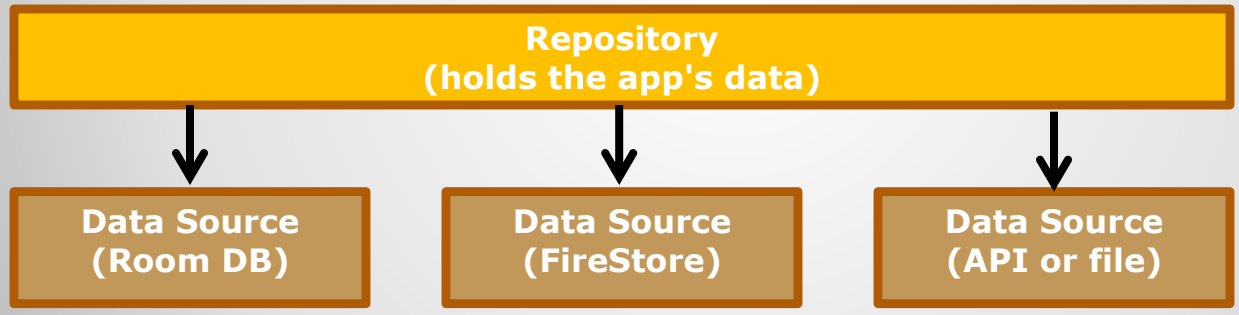


High-Level Architecture

© 2024 Arthur Hoskey. All rights reserved.

Data Layer - Repository

- A repository is meant to be a common access point to retrieve any data that the app might need (it exposes data to the rest of the app).
- This is an implementation of a Single Source of Truth architecture (SSOT).
- A repository will access one or more data sources to get the data it needs (Firestore, SQLite, some web API, file, etc…).
- Details about exactly how the data is stored are hidden from the UI layer.
- Some content take from: https://developer.android.com/topic/architecture/data-layer



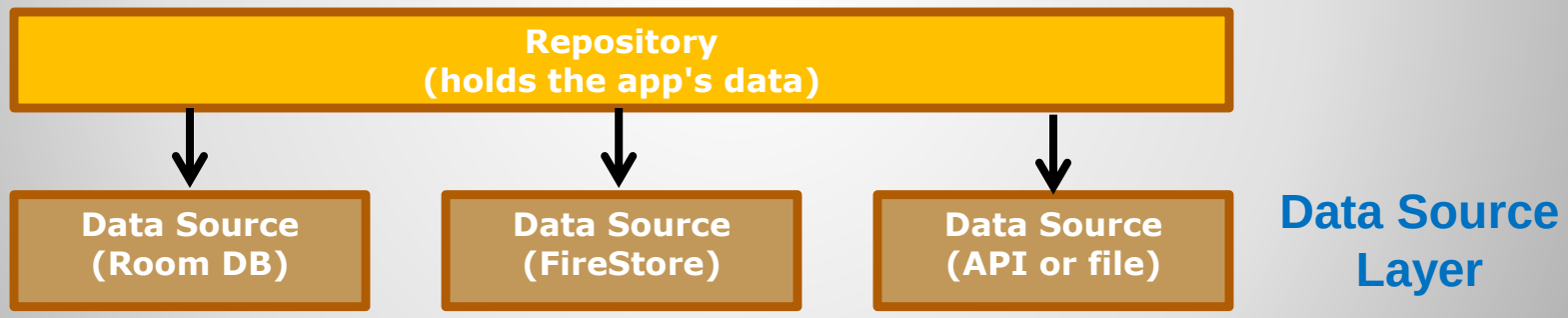
Repository Layer

Data Layer - Repository

© 2024 Arthur Hoskey. All rights reserved.

Data Layer – Data Source

- Each data source class should work with only one source of data.
- A source would be a Room DB, Firestore, network API, file, etc…
- The repository interacts with the data sources.
- Other parts of the app should never access a data source directly (they should go through the repository).
- Some content take from: https://developer.android.com/topic/architecture/data-layer



Data Layer – Data Source

© 2024 Arthur Hoskey. All rights reserved.

- Now on to dependency injection…

Dependency Injection

© 2024 Arthur Hoskey. All rights reserved.

Injection

- An injection (often and usually referred to as a "shot" in US English, a "jab" in UK English, or a "jag" in Scottish English and Scots) is the act of administering a liquid, especially a drug, into a person's body using a needle (usually a hypodermic needle) and a syringe.
- This definition was taken from: https://en.wikipedia.org/wiki/Injection_(medicine)

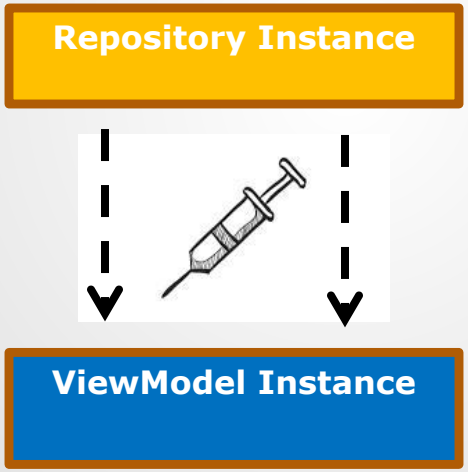
- Pic taken from: https://www.homage.com.my/health/injection/

Injection

© 2024 Arthur Hoskey. All rights reserved.

Dependency Injection

- An object is given other objects that it requires to do its work.
- For example, a ViewModel needs access to a repository to do its work.
- A repository instance should be "injected" into the ViewModel (do not create the repository instance inside the ViewModel).
- Just add a repository as a parameter to the ViewModel constructor (the repository will be "injected" into the ViewModel via the constructor).
- The ViewModel can now call into the repository.



The repository is "injected" into a ViewModel instance

- Pic taken from: https://www.istockphoto.com/vector/sketch-icon-syringe-gm1021808098-274385908

Dependency Injection

© 2024 Arthur Hoskey. All rights reserved.

Inject Repository into ViewModel

- A **ViewModel** needs to have access to the **repository**.
- When the ViewModel instance is created the repository will be passed in as a parameter to the constructor.
- Write methods on the ViewModel that will access the repository and return the data.

```
class MyViewModel(var myRepository: Repository) : ViewModel()
{
    fun getEmployeeData() : List<Employee> {
        return myRepository.getEmployees()
    }
    // Write other functions as necessary to access data
}

class Repository (
    // Repository parameters here
) {
   fun getGetEmployees() : List<Employee> { // Function code here }
}
```

Pass the Repository as a parameter to the ViewModel constructor

Sample ViewModel function to access repository data (assumes that getEmployees is a member function of the repository class that returns a list of employee)

Repository class snippet. It has a function to return a list of Employee as a member

Inject Repository into ViewModel

© 2024 Arthur Hoskey. All rights reserved.

Inject DataSource into Repository

- A **Repository** needs to have access to the **data source(s)**.
- When the Repository instance is created the data source will be passed in as a parameter to the constructor.
- Write methods on the Repository that will access the data source and return the data.

Pass the data source(s) as parameter(s) to the Repository constructor. If there are multiple data sources, then pass them all here.

```
class Repository (
    var myDataSource: MyDataSource
) {
    fun getEmployeeData() : List<Employee> {
        return myDataSource.getEmployees()
    }
    // Write other functions as necessary to access data
}
```

Sample Repository function to access a data source

Data source class snippet. It has a function to return a list of Employee as a member. The code that is in here will depend on where the data is coming from (Room DB, Firestore, file, API, etc…)

```
class MyDataSource (
    // Data source parameters here
) {
   fun getGetEmployees() : List<Employee> { // Function code here }
}
```

Inject Data Source into Repository

© 2024 Arthur Hoskey. All rights reserved.

DataSource Implementations

- The specific code in a data source will vary according to where the data is stored.
- Code that uses the repository is unaware of internal details of the data source(s).
- In the code below, all the data sources return a List<Employee>. Using this setup, one data source can easily be swapped out for another in the repository.

```
class RoomDBDataSource ( // Data source parameters here ) {
  fun getGetEmployees() : List<Employee> { // Code specific to getting data from a Room DB }
}

class FirestoreDataSource ( // Data source parameters here ) {
  fun getGetEmployees() : List<Employee> { // Code specific to getting data from a Firestore DB }
}

class FileDataSource (  // Data source parameters here ) {
  fun getGetEmployees() : List<Employee> { // Code specific to getting data from a normal file }
}

class NetworkAPIDataSource (  // Data source parameters here ) {
  fun getGetEmployees() : List<Employee> { // Code specific to getting data from a network API }
}
```

Data Source Implementations

© 2024 Arthur Hoskey. All rights reserved.

Repository Creation

- One place to create the repository is in the app's Application class.
- The Application class can be accessed from anywhere in the app.
- Do the following:

1. Create your own Application class (say MyApp). Add a repository type member variable as a companion object of your Application class (call it myRep).
2. Add a function override for Application.onCreate in your Application class. Application.onCreate gets called only once so we can create the repository instance here (for example, put the new repository instance in myRep).
3. Register the name of your Application class (MyApp) in AndroidManifest.xml
4. Getting the repository instance. Your ViewModel class can now access the repository instance through the MyApp class. For example, MyApp.myRep.

Repository Creation

© 2024 Arthur Hoskey. All rights reserved.

- End of Slides

End of Slides

© 2024 Arthur Hoskey. All rights reserved.