Lecture 9

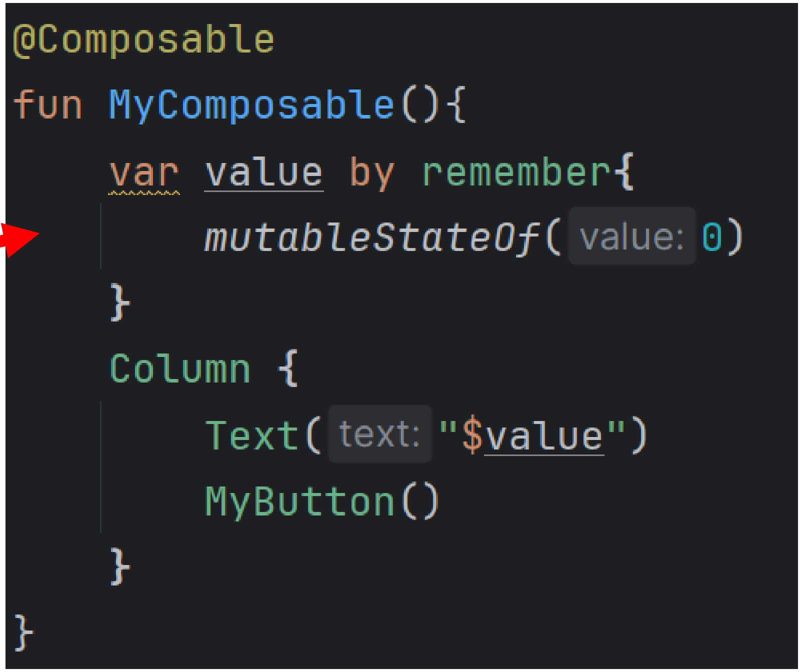
COMP 3717- Mobile Dev with Android Tech

State hoisting

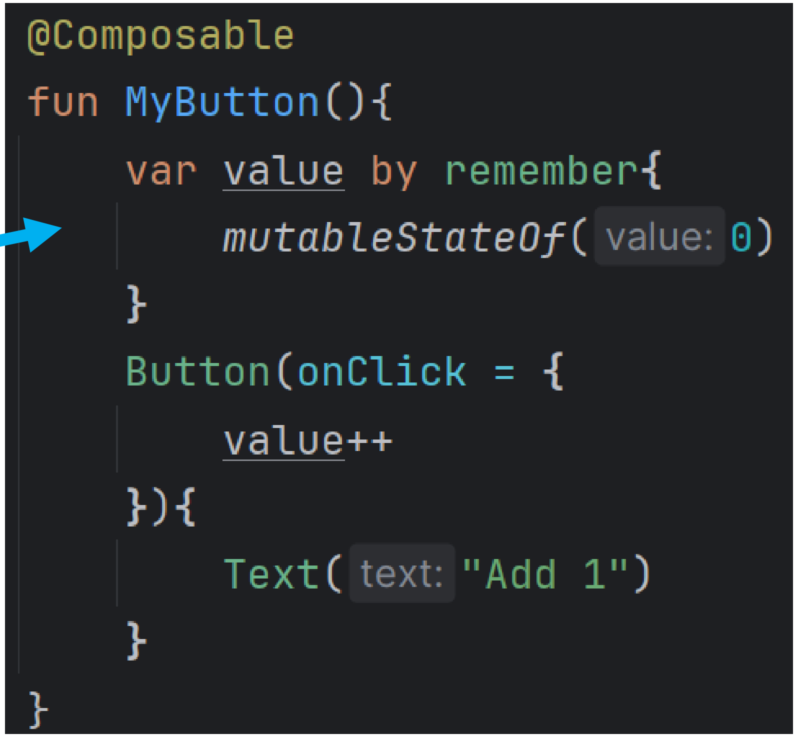
- Moving state to the *lowest common ancestor*
  - Maintains a clear single source of truth
  - Encourages stateless composables
  - Promotes reusability and more maintainable code
  - Avoids unnecessary recompositions

Single source of truth

- Below demonstrates a lack of a *single source of truth*
  - *Source 1*
  - *Source 2*



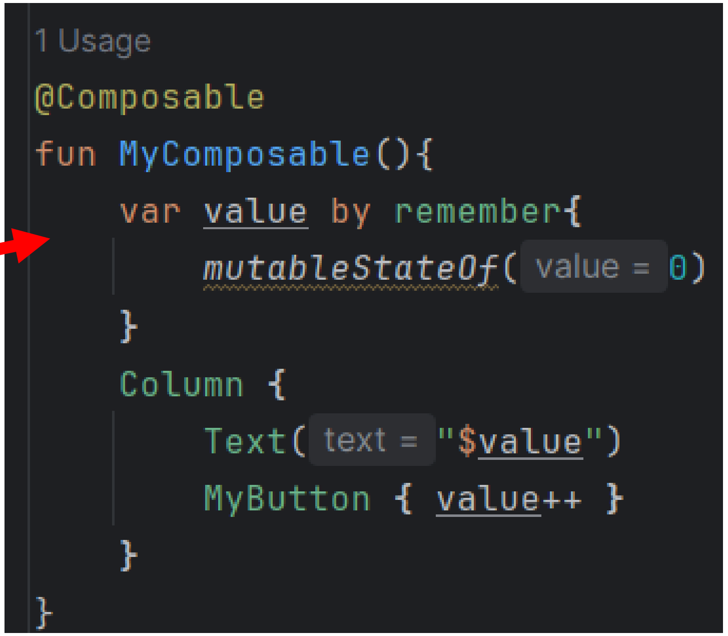
```
@Composable
fun MyComposable(){
    var value by remember{
        mutableStateOf( value: 0)
    }
    Column {
        Text( text: "$value")
        MyButton()
    }
}
```



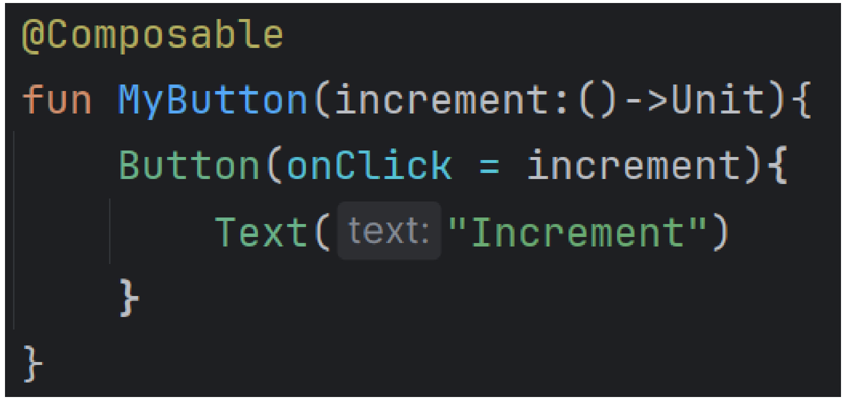
```
@Composable
fun MyButton(){
    var value by remember{
        mutableStateOf( value: 0)
    }
    Button(onClick = {
        value++
    }){
        Text( text: "Add 1")
    }
}
```

Single source of truth (cont.)

- By refactoring the previous example, we now have a *single source of truth*



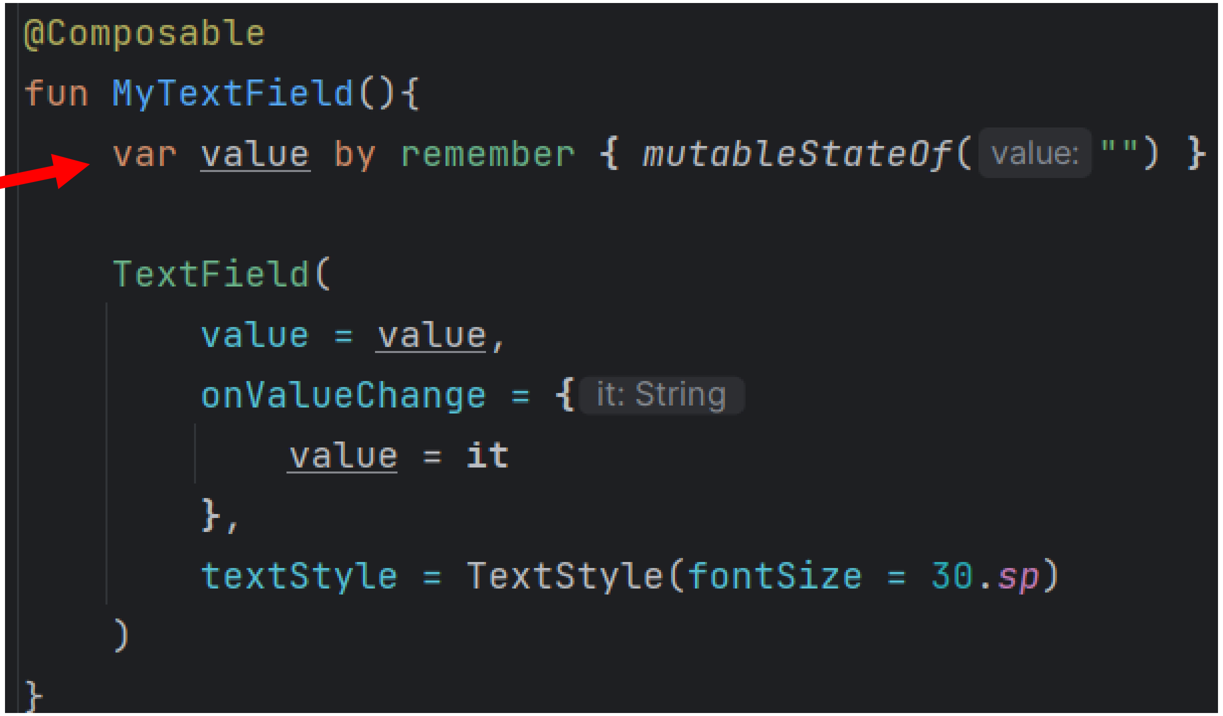
```
1 Usage
@Composable
fun MyComposable(){
    var value by remember{
        mutableStateOf( value = 0)
    }
    Column {
        Text( text = "$value")
        MyButton { value++ }
    }
}
```



```
@Composable
fun MyButton(increment:()->Unit){
    Button(onClick = increment){
        Text( text: "Increment")
    }
}
```

Stateful

- Here is another example of a *single source of truth*
    - A composable that contains state is considered *stateful*

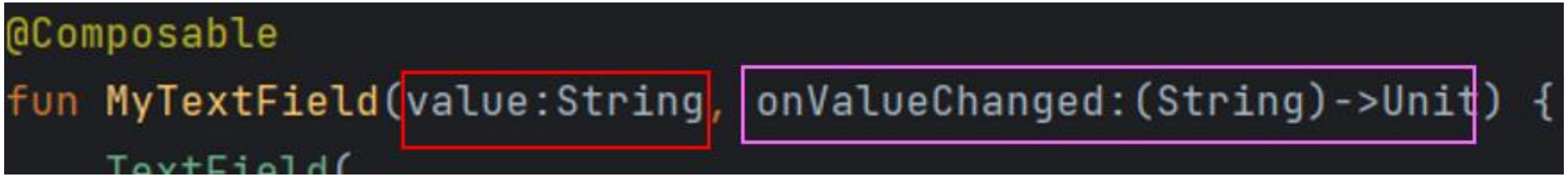


```
@Composable
fun MyTextField(){
    var value by remember { mutableStateOf( value: "") }

    TextField(
        value = value,
        onValueChange = { it: String
            value = it
        },
        textStyle = TextStyle(fontSize = 30.sp)
    )
}
```

Stateless

- When we hoist state, we make the composable *stateless*

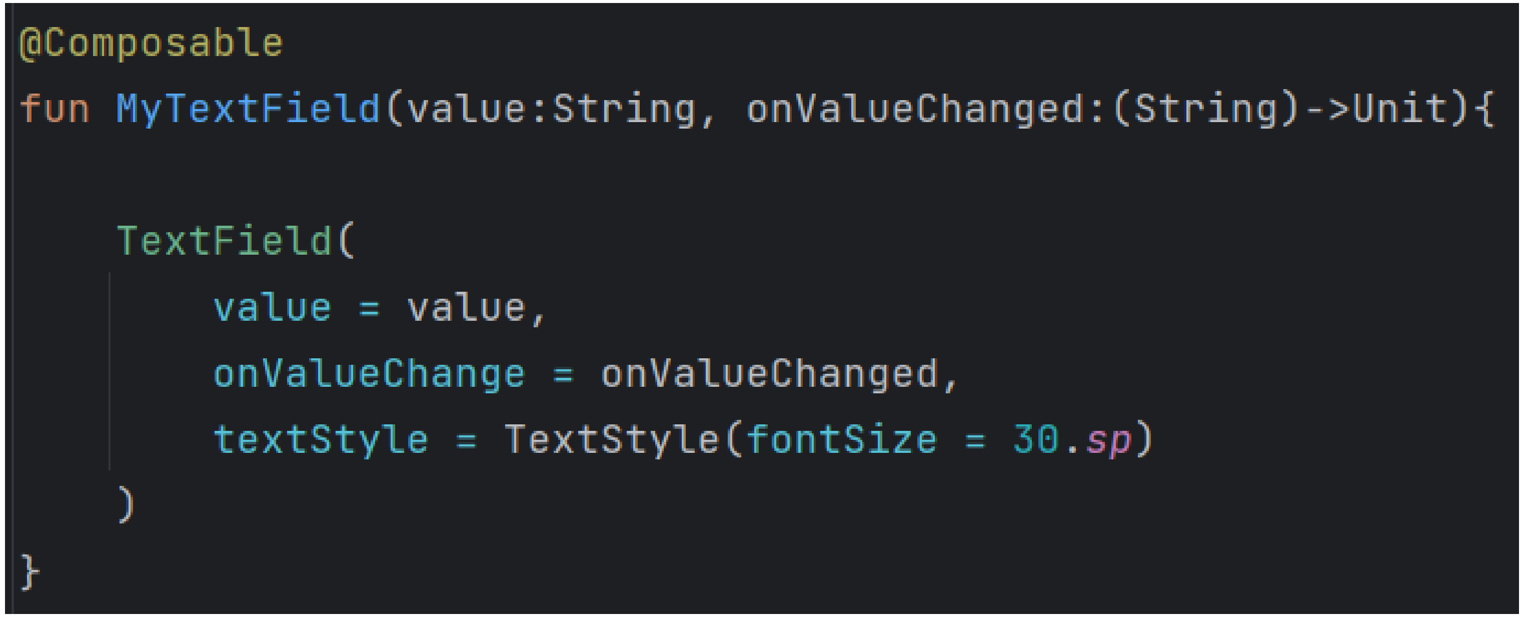
- The state variable is usually replaced with
  - The current value that is read
  - An event callback that sets the value



```
@Composable
fun MyTextField(value:String, onValueChanged:(String)->Unit) {
    TextField(
```

Stateless (cont.)

- When we hoist state, we make the composable *stateless*
  - Stateless composables don't hold or modify state

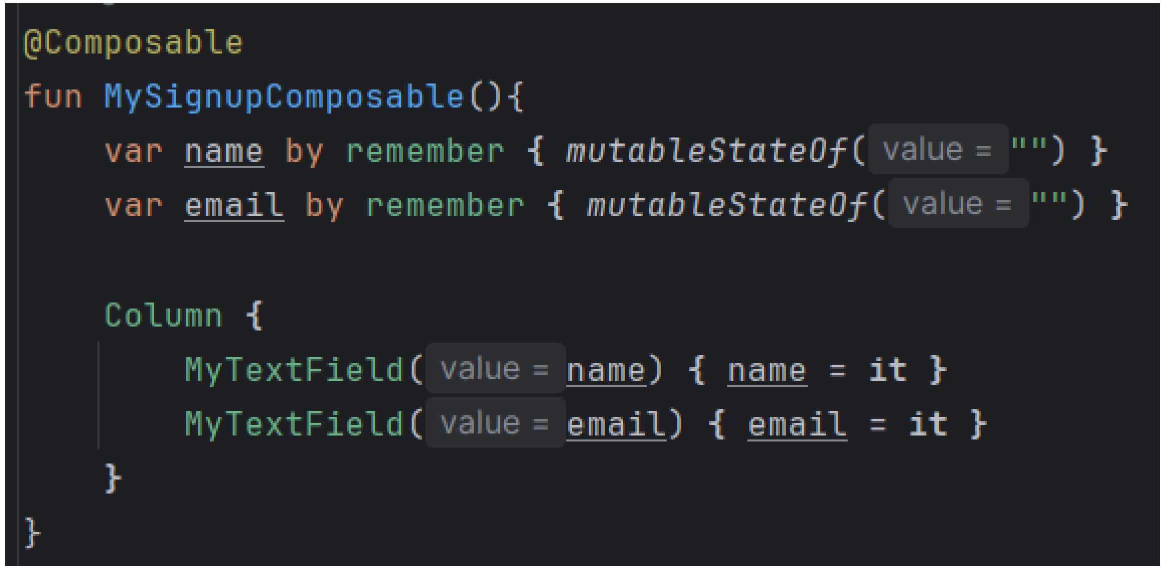


```
@Composable
fun MyTextField(value:String, onValueChanged:(String)->Unit){

    TextField(
        value = value,
        onValueChange = onValueChanged,
        textStyle = TextStyle(fontSize = 30.sp)
    )
}
```

State hoisting (cont.)

- *MyTextField* now becomes more decoupled

- We can reuse it with different values and event callbacks



```
@Composable
fun MySignupComposable(){
    var name by remember { mutableStateOf( value = "") }
    var email by remember { mutableStateOf( value = "") }

    Column {
        MyTextField( value = name) { name = it }
        MyTextField( value = email) { email = it }
    }
}
```

State holder

- Usually, a plain class or *ViewModel*

- Used when your state and logic become too hard be to maintained within the composable itself

- Types
  - UI logic state holder
  - Business logic state holder

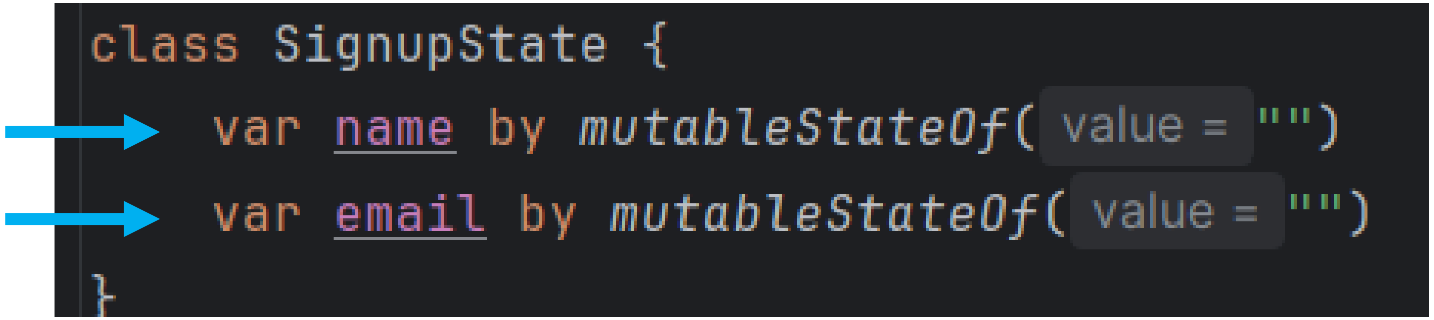
UI logic state holder

- In this scenario the state holder contains the UI state and logic
    - Typically, a plain class

- The composables responsibility is just to oversee the emitting of UI elements
    - Which favors the separation of concerns principle

UI logic state holder (cont.)

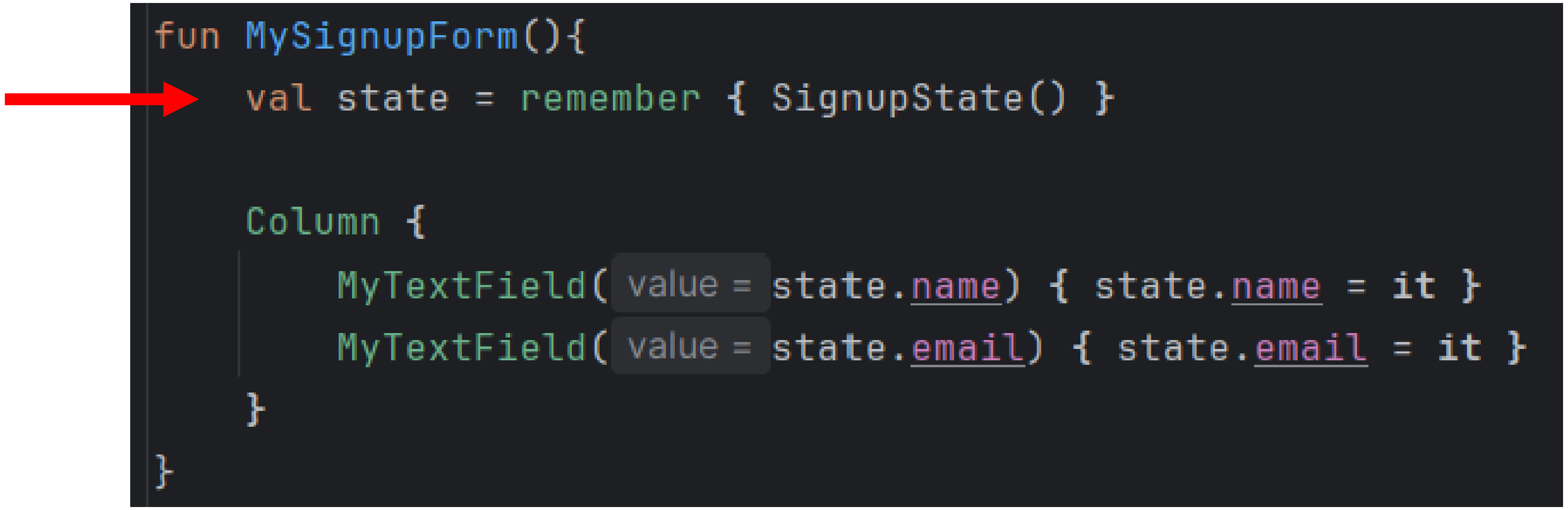
- Usually when a composable has multiple state objects, or the logic is too complex, should we *hoist* the state to a state holder



```
class SignupState {
    var name by mutableStateOf( value = "")
    var email by mutableStateOf( value = "")
}
```

UI logic state holder (cont.)

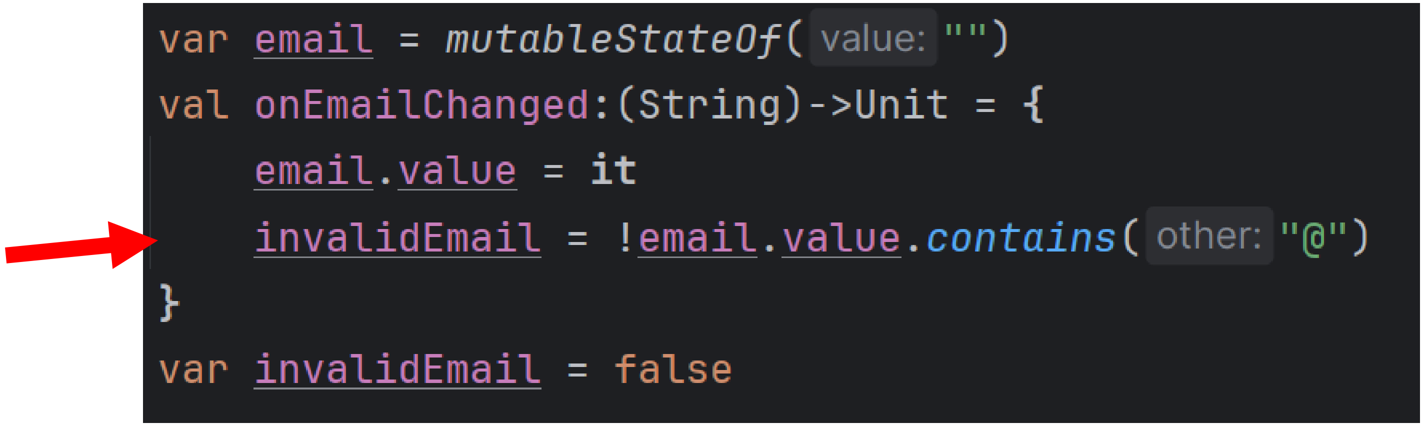
- You can then use one object to manage all your state within the composable



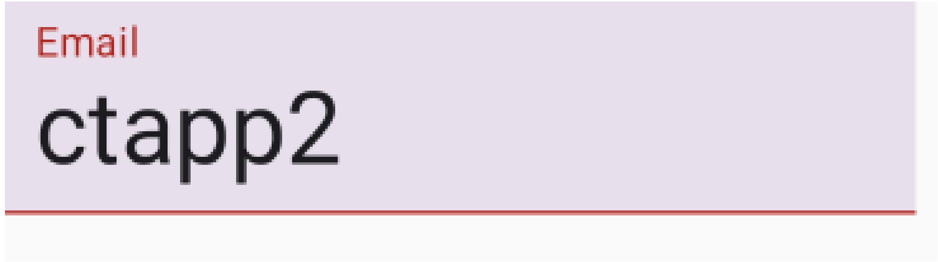
```
fun MySignupForm(){
    val state = remember { SignupState() }

    Column {
        MyTextField( value = state.name) { state.name = it }
        MyTextField( value = state.email) { state.email = it }
    }
}
```

UI logic

- *How* the content is being displayed and experienced
  - E.g. Highlight the *TextField* red if the text does not contain an @ character



```
var email = mutableStateOf( value: "")
val onEmailChanged:(String)->Unit = {
    email.value = it
    invalidEmail = !email.value.contains( other: "@")
}
var invalidEmail = false
```



Email

ctapp2

Business logic state holder

- In this scenario the state holder contains the business state and logic
    - Either a plain class or a *ViewModel*

- An intermediary that coordinates application data between the data layer and UI layer
    - application data: The information that is generated, used, and stored within our app

Business logic

- Rules and requirements for data before it is processed by the data layer
  - Use cases (Domain)
  - Validation

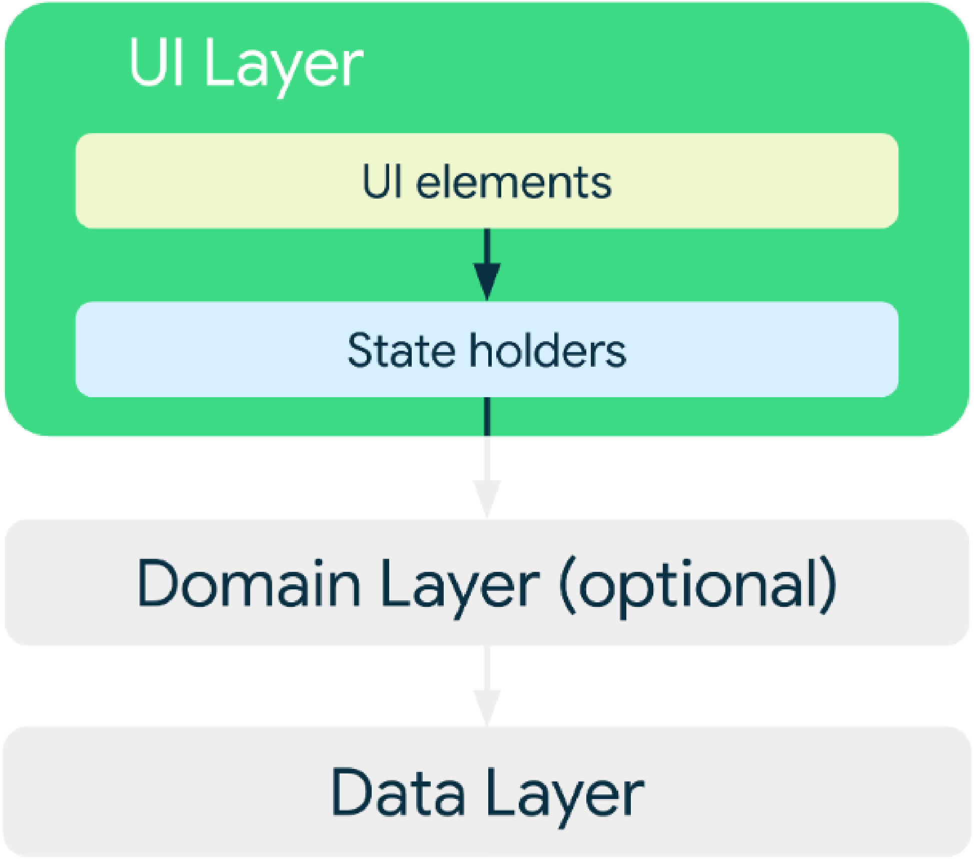
- Think of *business* as the company or organization building the app

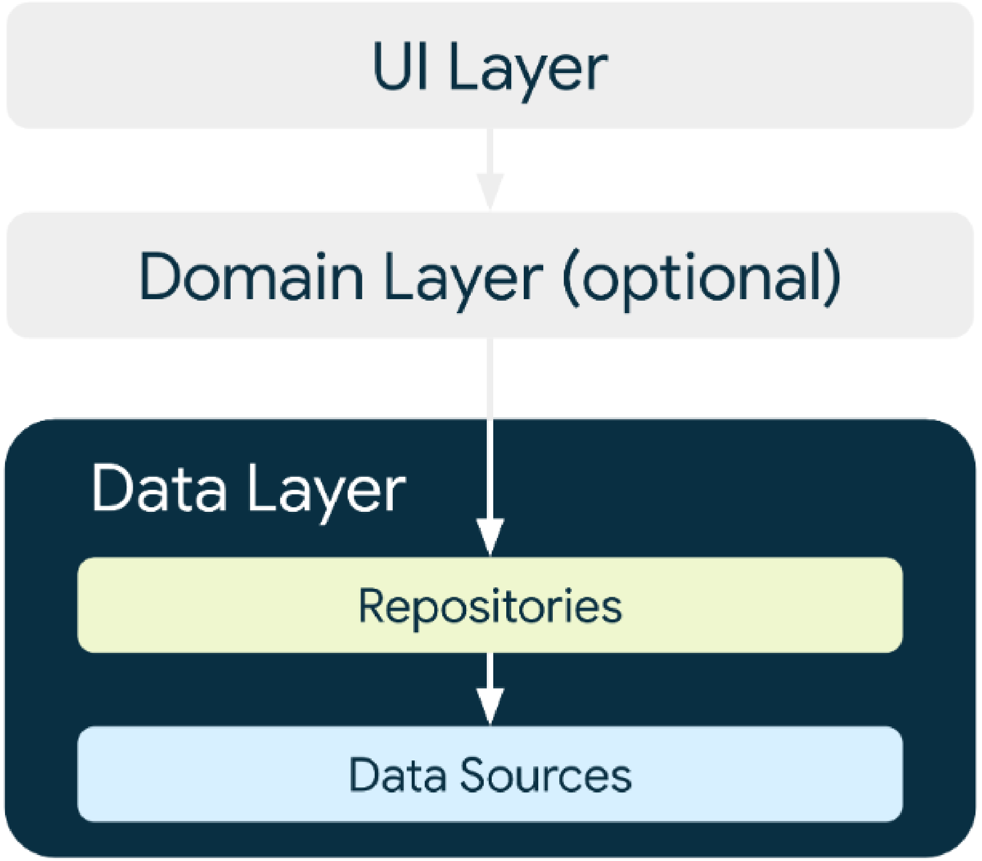
Layers

- When our app starts scale, we need to maintain clean architecture
  - Decoupling the layers completely

- The two basic layers are the UI layer and Data layer
  - UI layer
    - UI elements (composables)
    - State holders (UI and Business)
  - Data layer
    - Data sources
    - Repositores

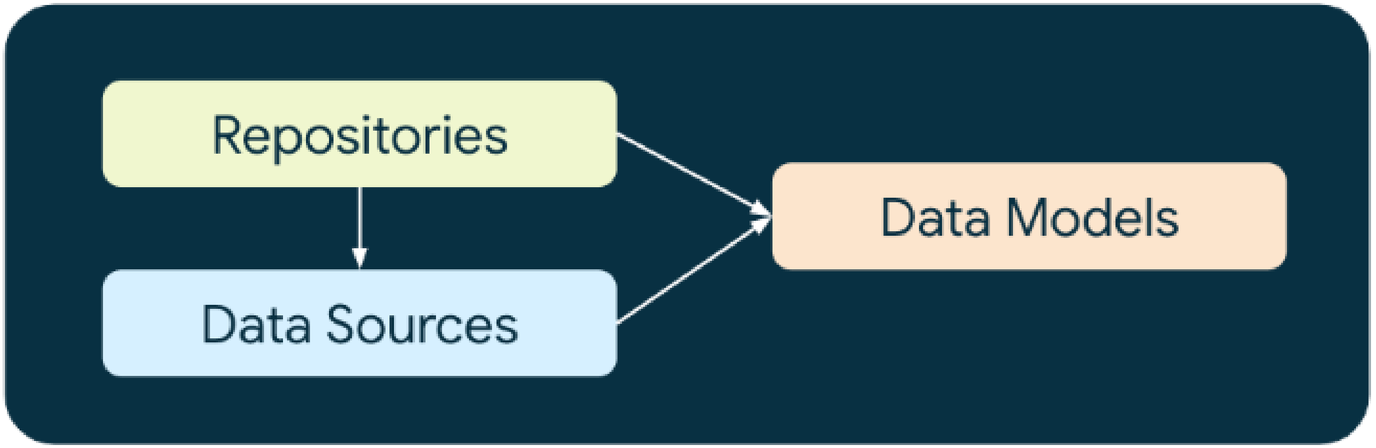
Layers (cont.)


UI Layer
UI elements
State holders
Domain Layer (optional)
Data Layer


UI Layer
Domain Layer (optional)
Data Layer
Repositories
Data Sources

Data layer

- A data layer contains three important parts
  - Data sources
  - Repositories
  - Data models



Data layer (cont.)

- A data layer can contain one or multiple data sources
  - Local or remote

- Local data sources
  - File
    - Ideal for storing raw complex data
  - Local database
    - Ideal for storing structured and relational data with querying capabilities
  - DataStore (Jetpack library)
    - Ideal for storing small and simple datasets

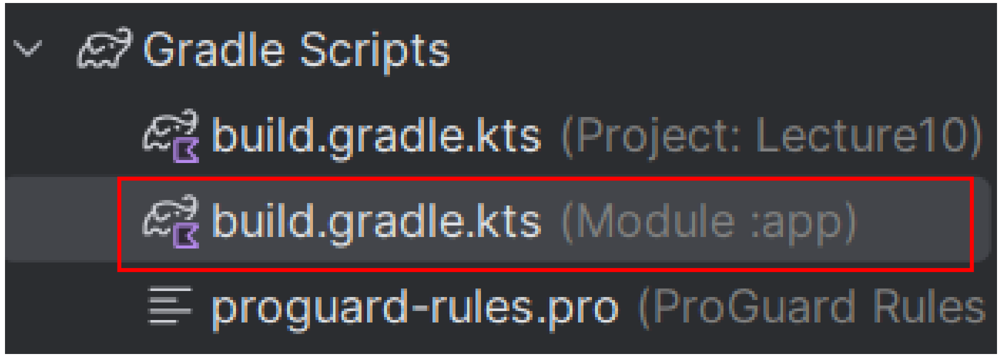
Room

- Room is one of the Jetpack libraries that provides access to a local SQLite database (Relational)

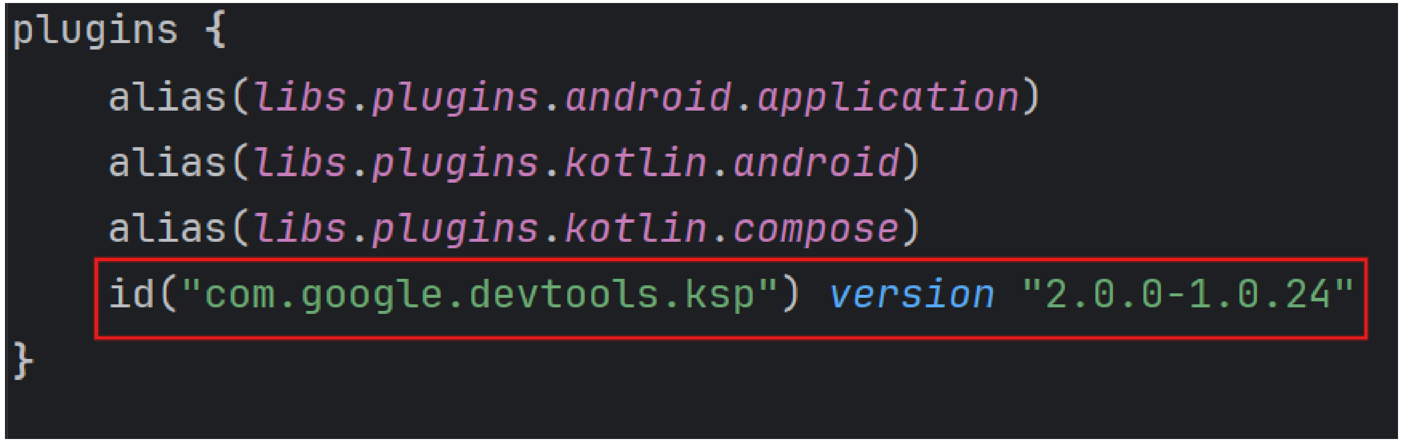
- The primary components in Room are:
  - Data entities
  - Data access objects
  - Database class

Room dependencies

- Update your *Module-level build.gradle* with *Kotlin Symbol Processing*



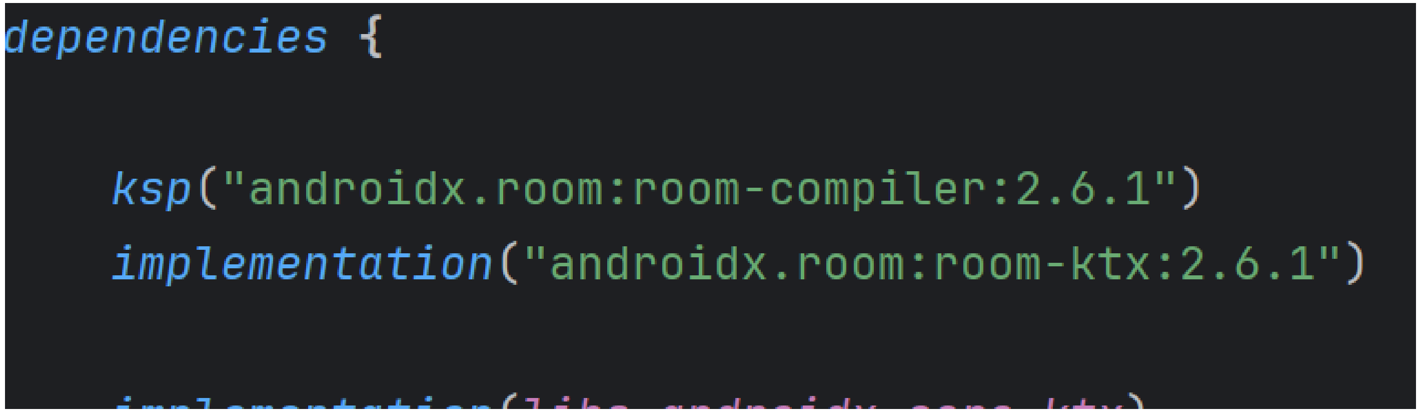
Gradle Scripts
build.gradle.kts (Project: Lecture10)
build.gradle.kts (Module :app)
proguard-rules.pro (ProGuard Rules



```
plugins {
    alias(libs.plugins.android.application)
    alias(libs.plugins.kotlin.android)
    alias(libs.plugins.kotlin.compose)
    id("com.google.devtools.ksp") version "2.0.0-1.0.24"
}
```

Room dependencies (cont.)

- Then add the Room dependencies at the bottom



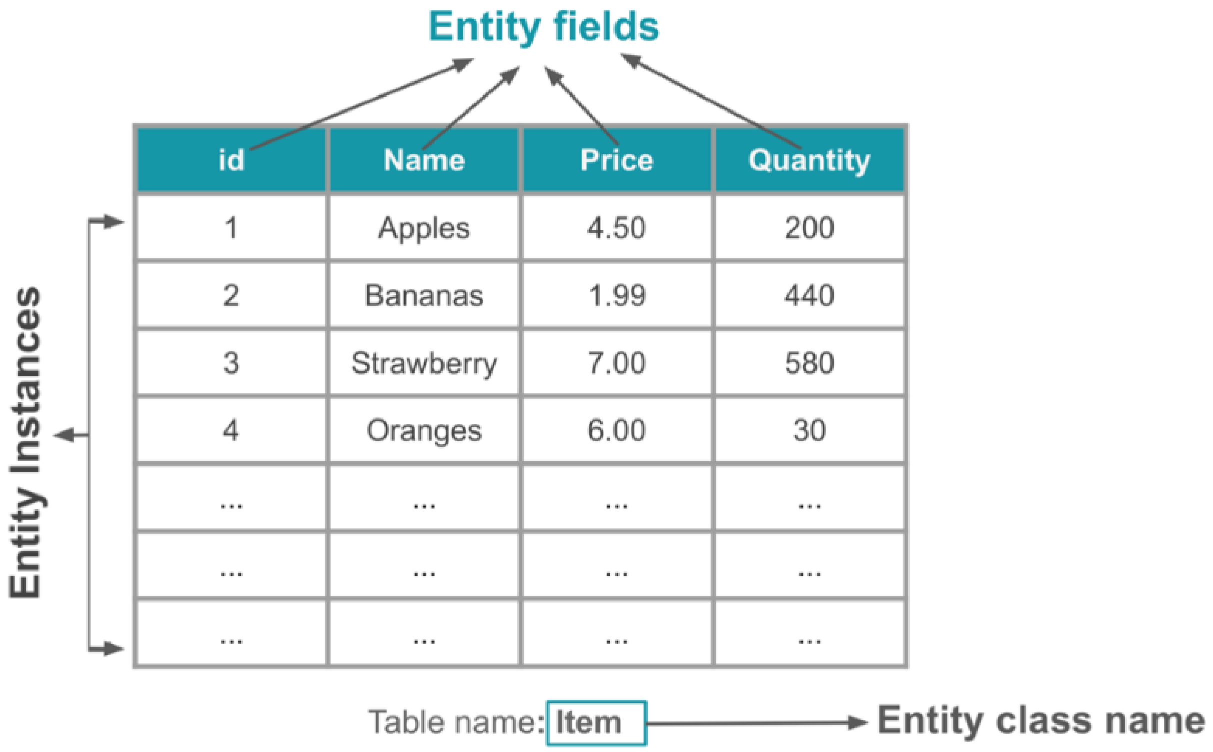
```
dependencies {

    ksp("androidx.room:room-compiler:2.6.1")
    implementation("androidx.room:room-ktx:2.6.1")

    implementation(libs.androidx.core.ktx)
```

- Sync your project and run your app to check if it works

Room (cont.)

- Data entities represent tables in your database

- Each instance of an Entity class represents a row in the table

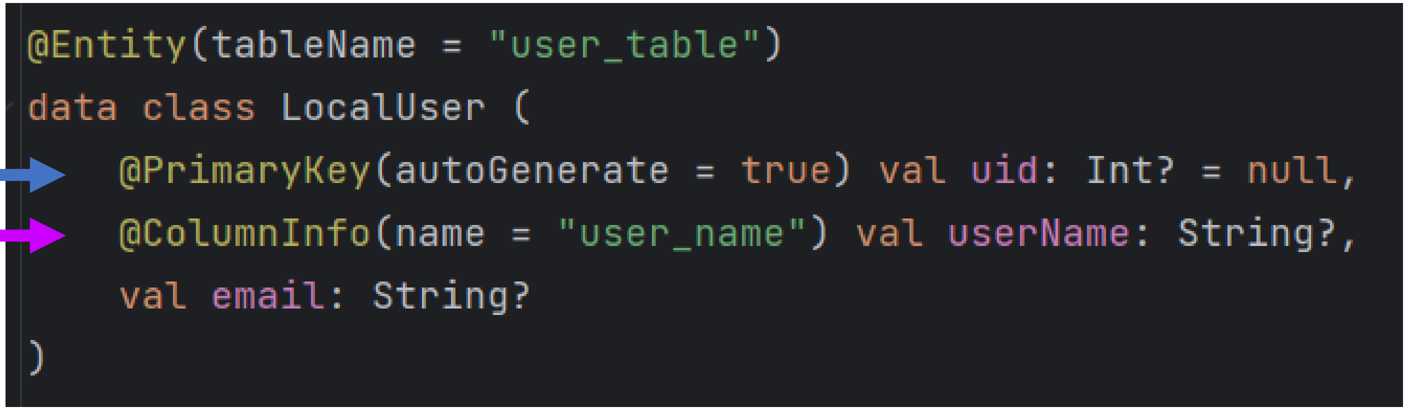


id	Name	Price	Quantity
1	Apples	4.50	200
2	Bananas	1.99	440
3	Strawberry	7.00	580
4	Oranges	6.00	30
...	...	...	...
...	...	...	...
...	...	...	...

Table name: Item

Room (cont.)

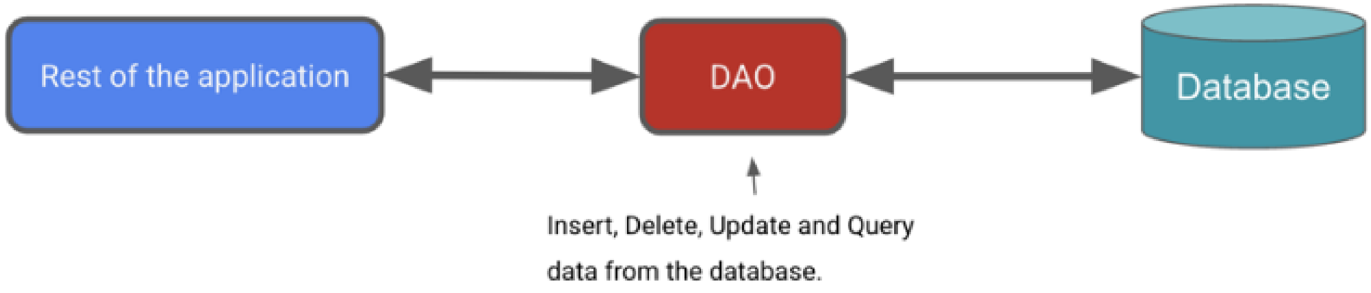
- *@Entity* marks a class as a database Entity class
- @PrimaryKey marks a field as the primary key
    - Every entity instance must have a primary key
- Each field is represented as a column in the database
    - *@ColumnInfo allows us to provide a custom name for it*



```
@Entity(tableName = "user_table")
data class LocalUser (
    @PrimaryKey(autoGenerate = true) val uid: Int? = null,
    @ColumnInfo(name = "user_name") val userName: String?,
    val email: String?
)
```

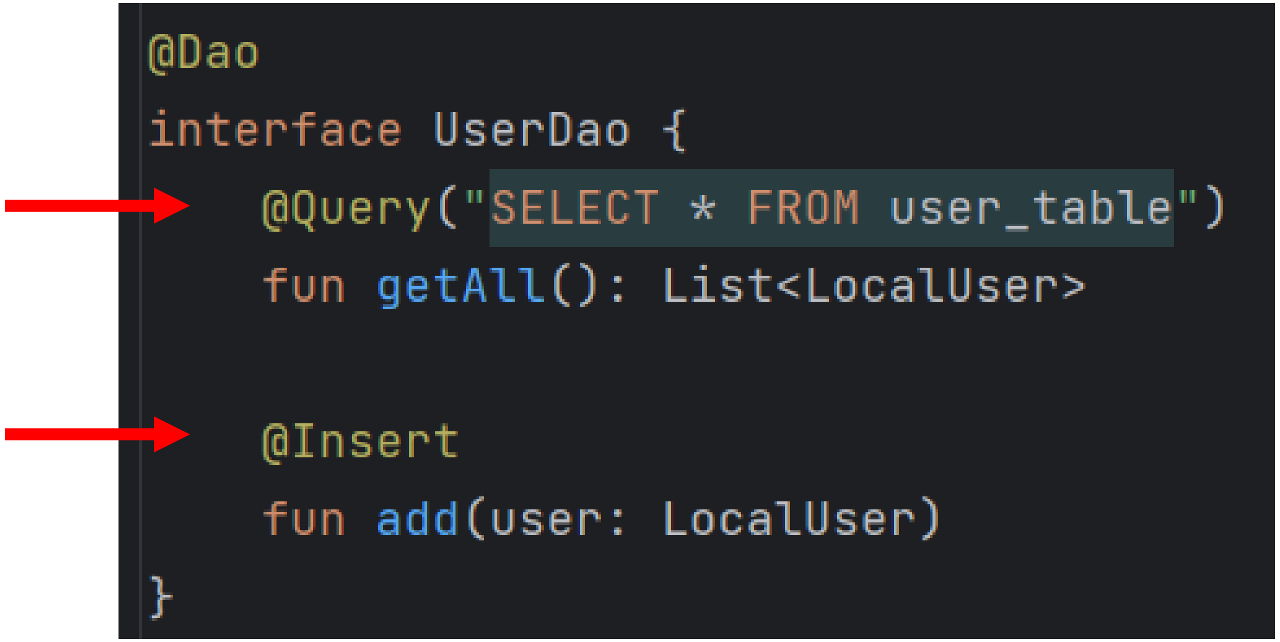
Room (cont.)

- Data Access Objects (DOAs) provide the CRUD functions the app uses to interact with database
    - Insert, query, update, delete, etc



Room (cont.)

- The Room library provides convenience annotations without requiring you to write an SQL statement

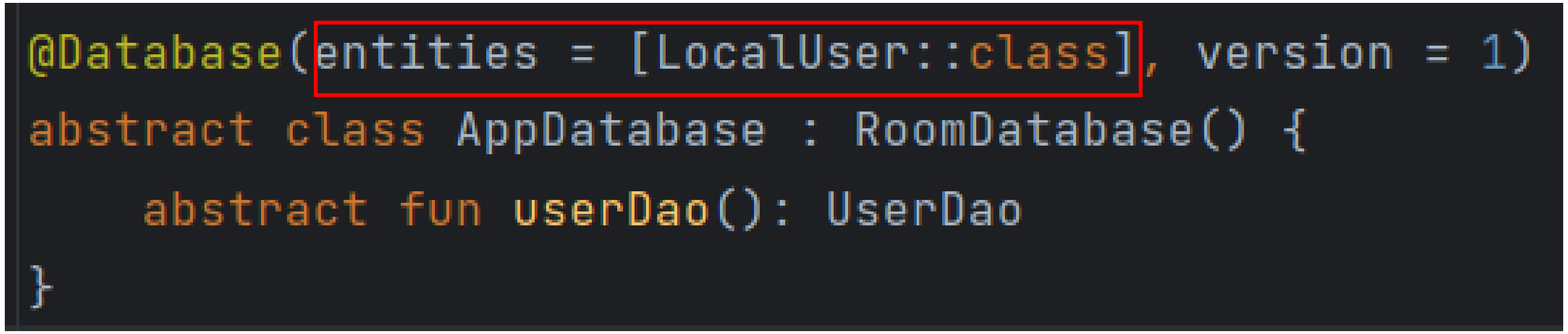


```
@Dao
interface UserDao {
    @Query("SELECT * FROM user_table")
    fun getAll(): List<LocalUser>

    @Insert
    fun add(user: LocalUser)
}
```

Room (cont.)

- The database class annotated with *@Database* holds the database
  - The main access point to the persisted data

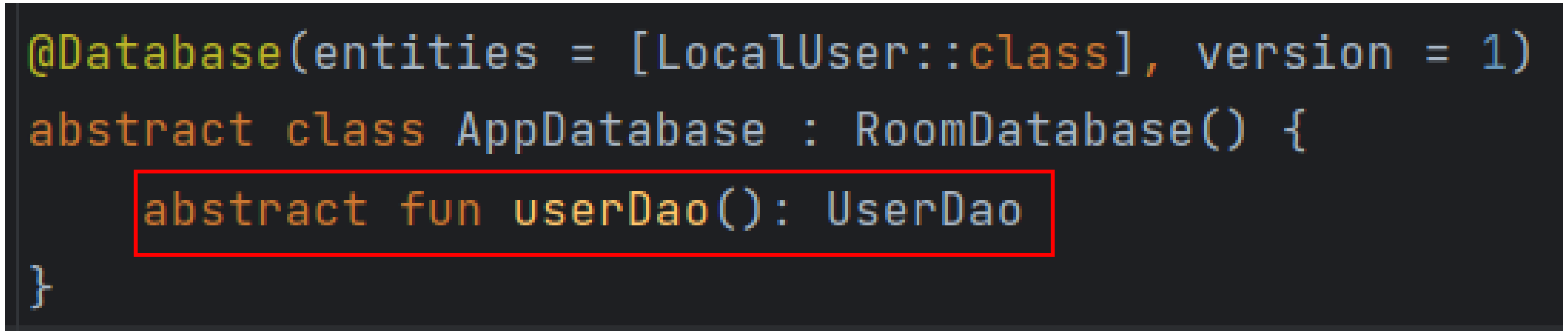
- It defines the list of entities
  - In this example we just have LocalUser



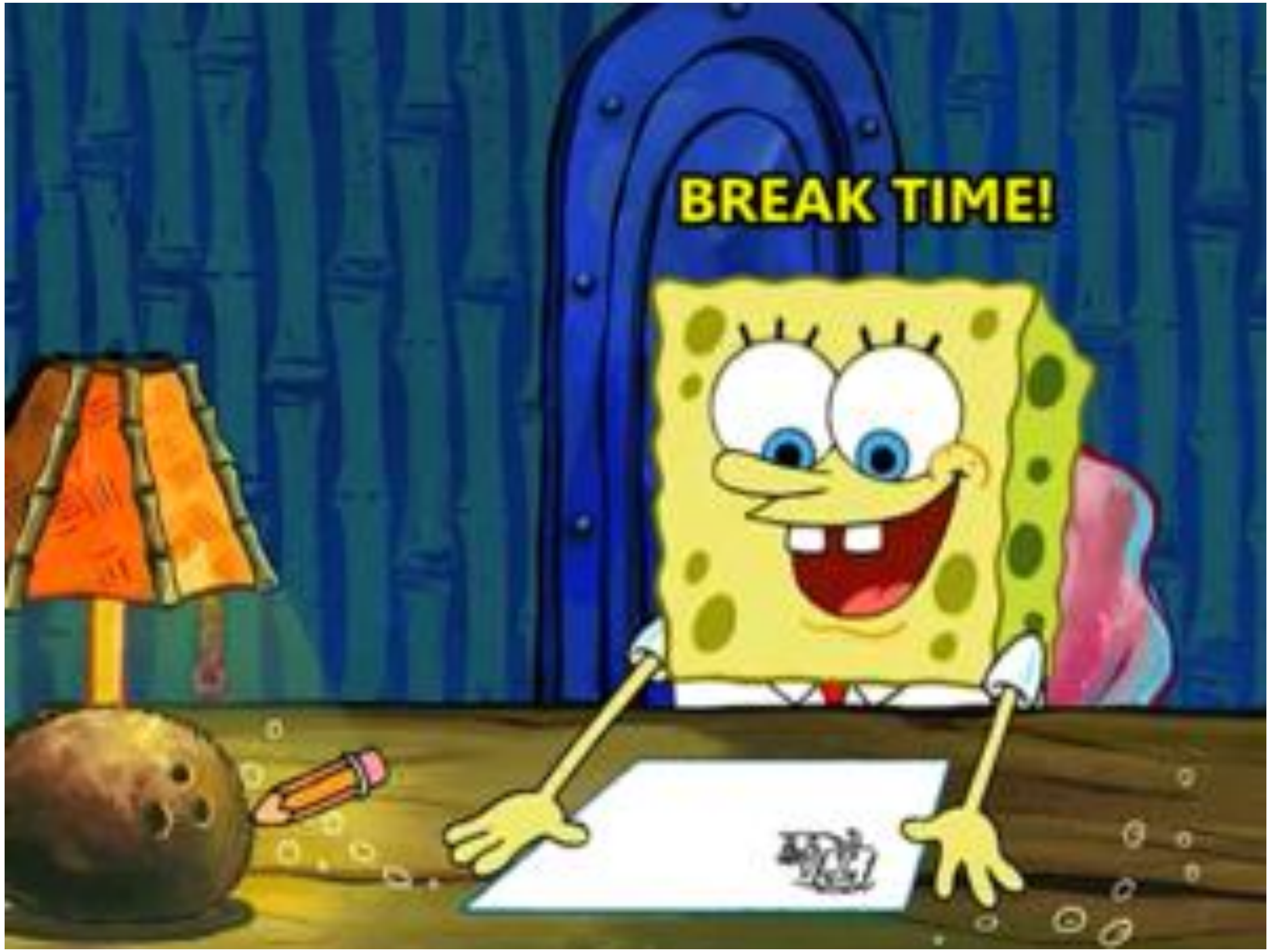
```
@Database(entities = [LocalUser::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

Room (cont.)

- The database class also provides the instances of the DAOs
- The DAOs are abstract because Room auto creates the implementation for us at compile time



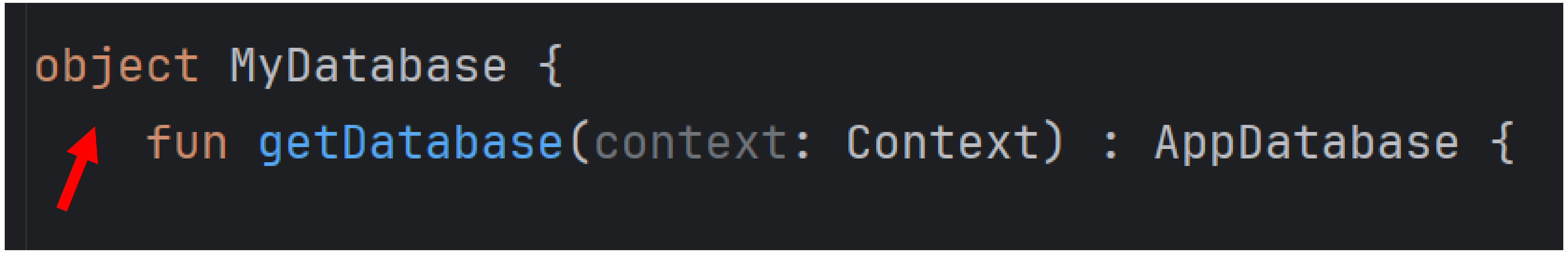
```
@Database(entities = [LocalUser::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```


BREAK TIME!

Singleton

- We only ever want one instance of our DB so let’s use a singleton
  - Singleton: A design pattern that ensures a class has only one instance

- Kotlin reduces a lot of the boilerplate code when creating singleton classes by using the object keyword

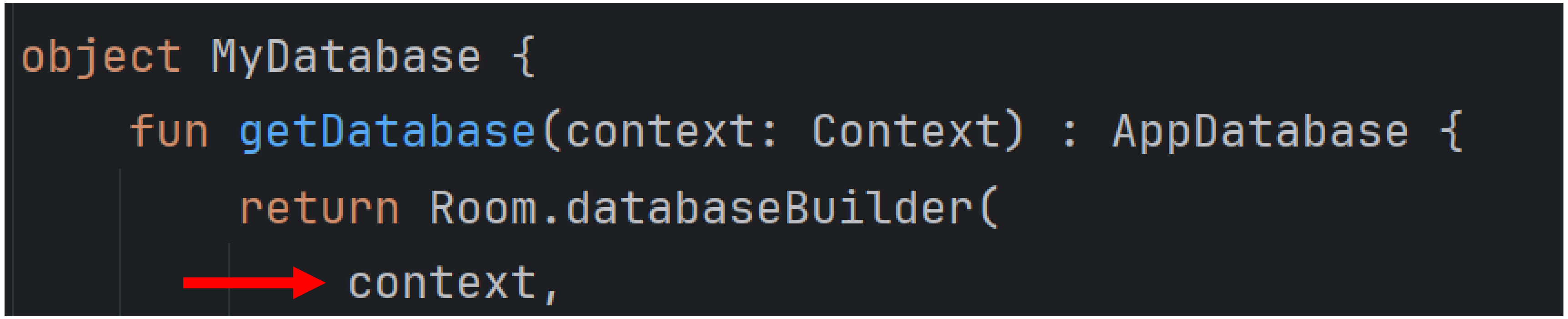


```
object MyDatabase {
    fun getDatabase(context: Context) : AppDatabase {
```

Application Context

- Application context is used to obtain information about the application

- Room databases are stored locally on the device in a directory specific to the app itself
  - When we create our DB instance, we will pass in the application context

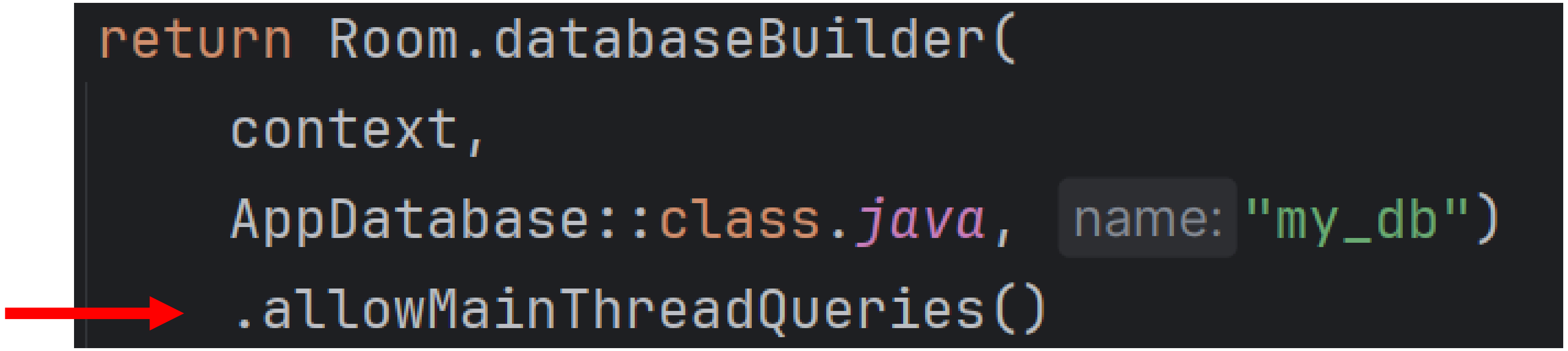


```
object MyDatabase {
    fun getDatabase(context: Context) : AppDatabase {
        return Room.databaseBuilder(
            context,
```

Room (cont.)

- Room databases can’t run queries on the main thread by default
  - It could freeze or slow down the main thread significantly

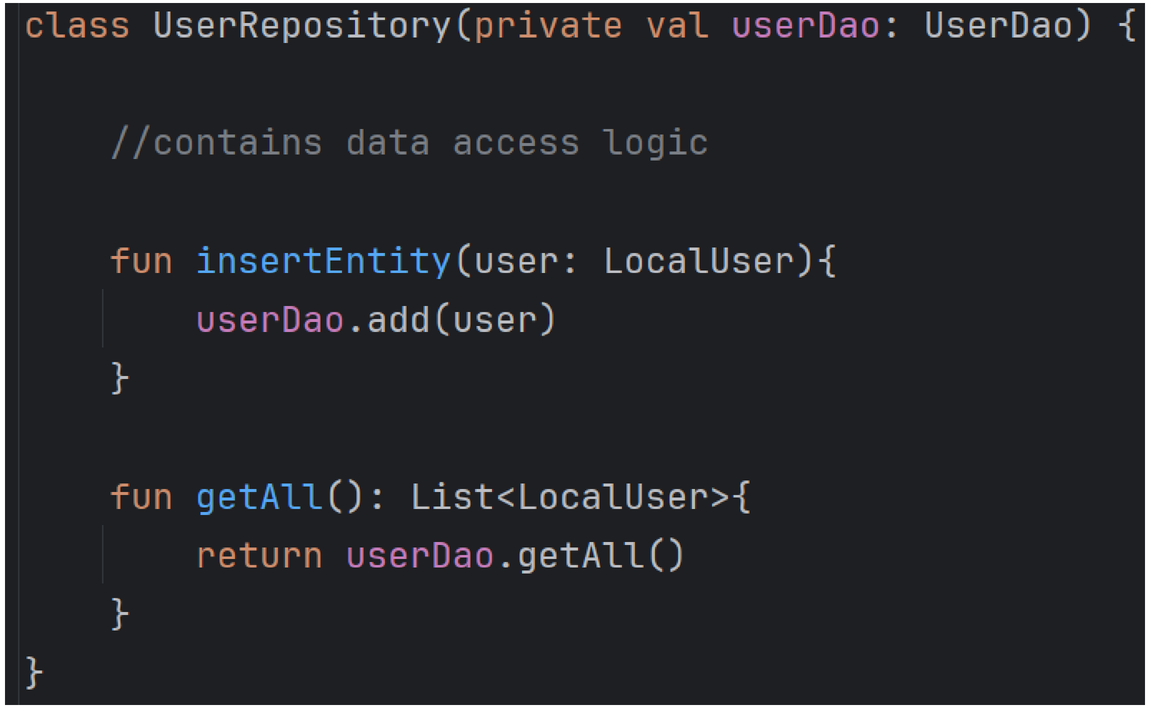
- But for this lesson, we will allow it



```
return Room.databaseBuilder(
    context,
    AppDatabase::class.java, name: "my_db")
    .allowMainThreadQueries()
```

Repository

- Now that we created our data source, we need a *Repository* to access it

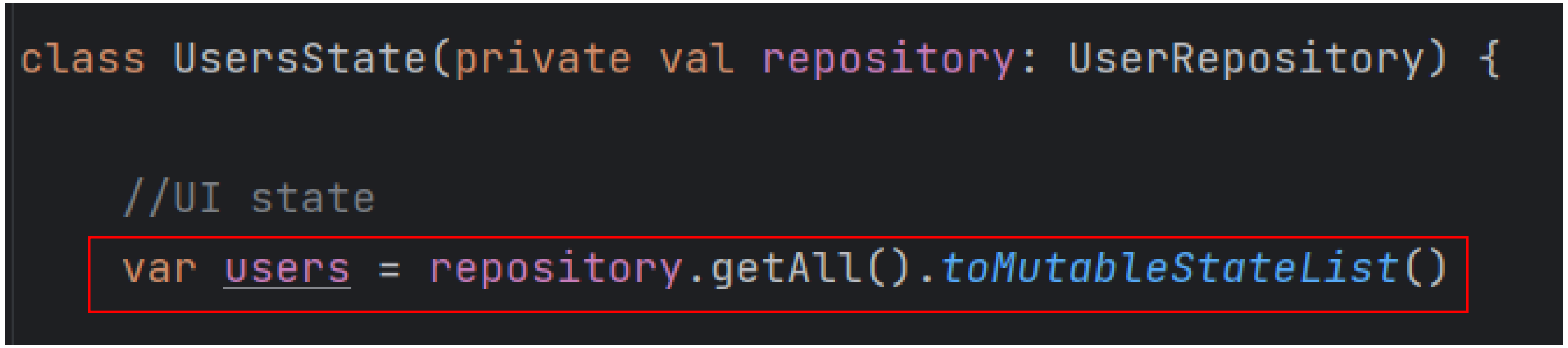


```
class UserRepository(private val userDao: UserDao) {

    //contains data access logic

    fun insertEntity(user: LocalUser){
        userDao.add(user)
    }

    fun getAll(): List<LocalUser>{
        return userDao.getAll()
    }
}
```

Business logic state holder (cont.)

- Jumping back to the UI layer we need to create a state holder for our application data

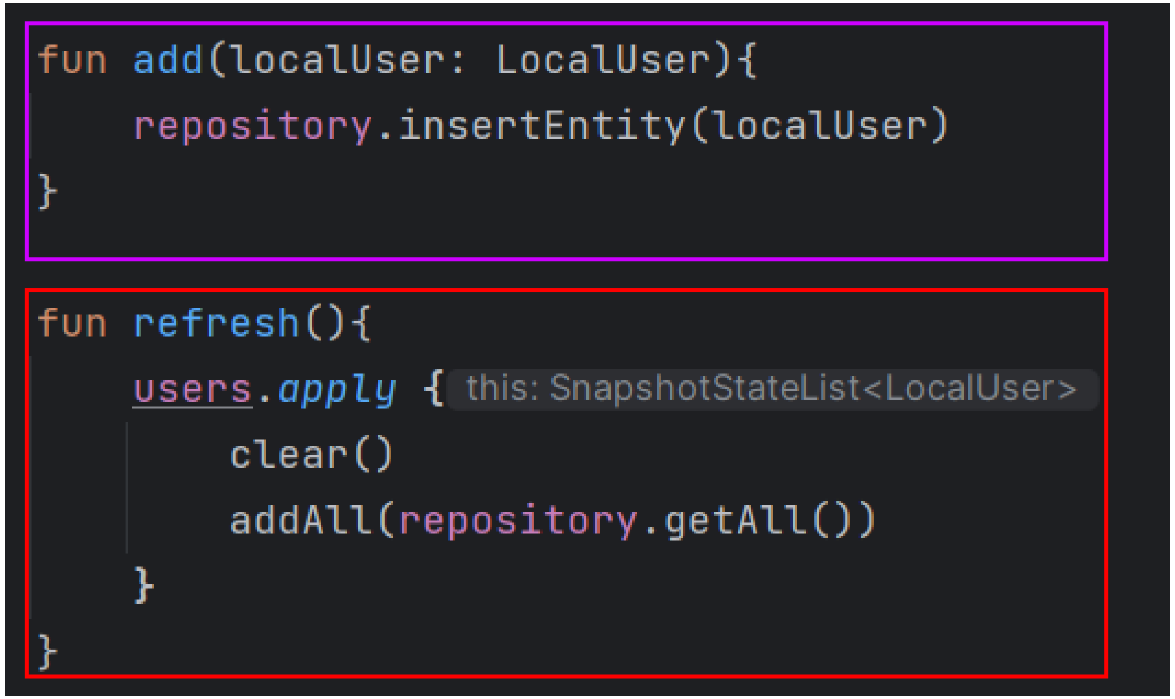
- First, we can create some state that reflects our current users



```
class UsersState(private val repository: UserRepository) {

    //UI state
    var users = repository.getAll().toMutableStateList()
```

Business logic state holder (cont.)

- Here we have two functions
  - A way to insert an entity in the database
  - A way to set our state with the current users in the database



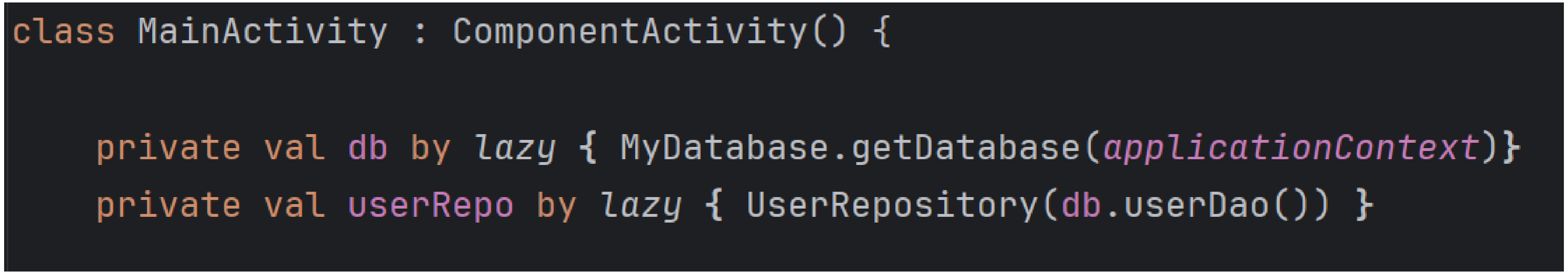
```
fun add(localUser: LocalUser){
    repository.insertEntity(localUser)
}
```

```
fun refresh(){
    users.apply { this: SnapshotStateList<LocalUser>
        clear()
        addAll(repository.getAll())
    }
}
```

Putting it all together

- We then need to initialize or DB and Repository

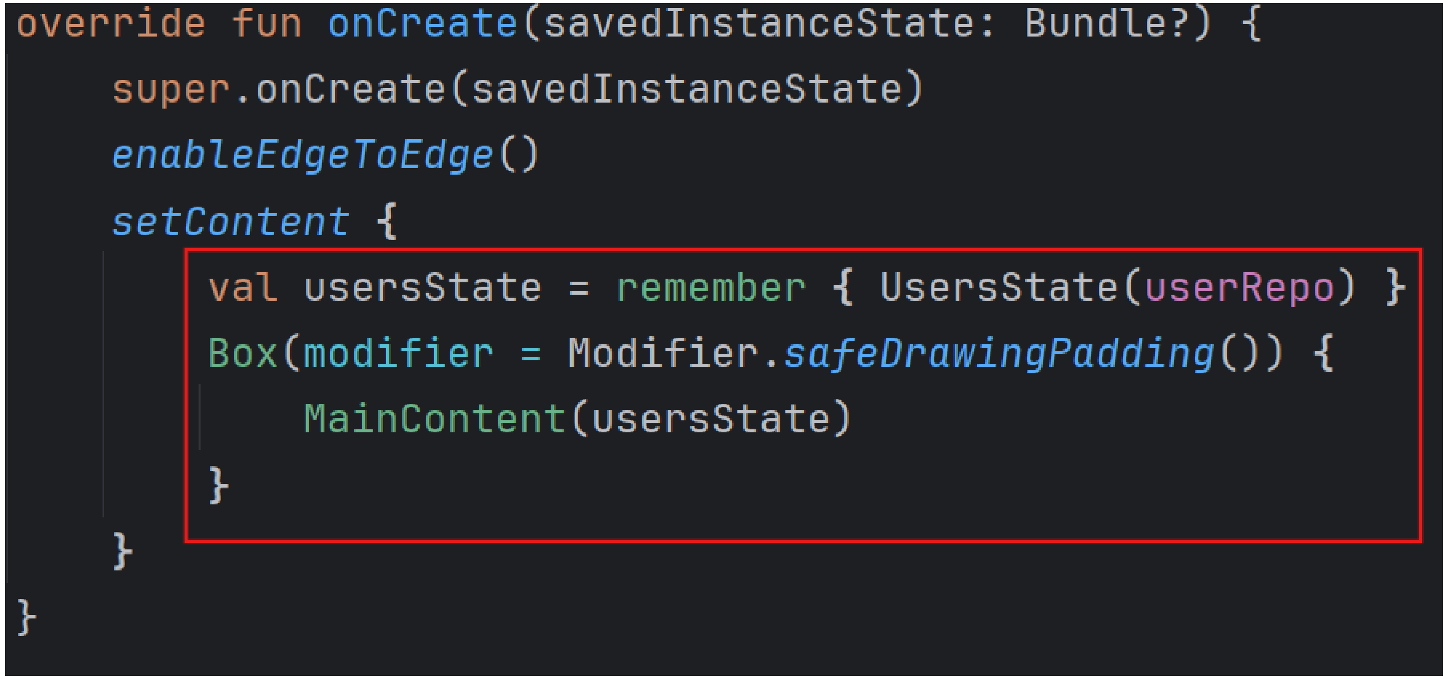
- This should be done in *MainActivity*, outside of *onCreate*



```
class MainActivity : ComponentActivity() {

    private val db by lazy { MyDatabase.getDatabase(applicationContext)}
    private val userRepo by lazy { UserRepository(db.userDao()) }
```

Putting it all together (cont.)

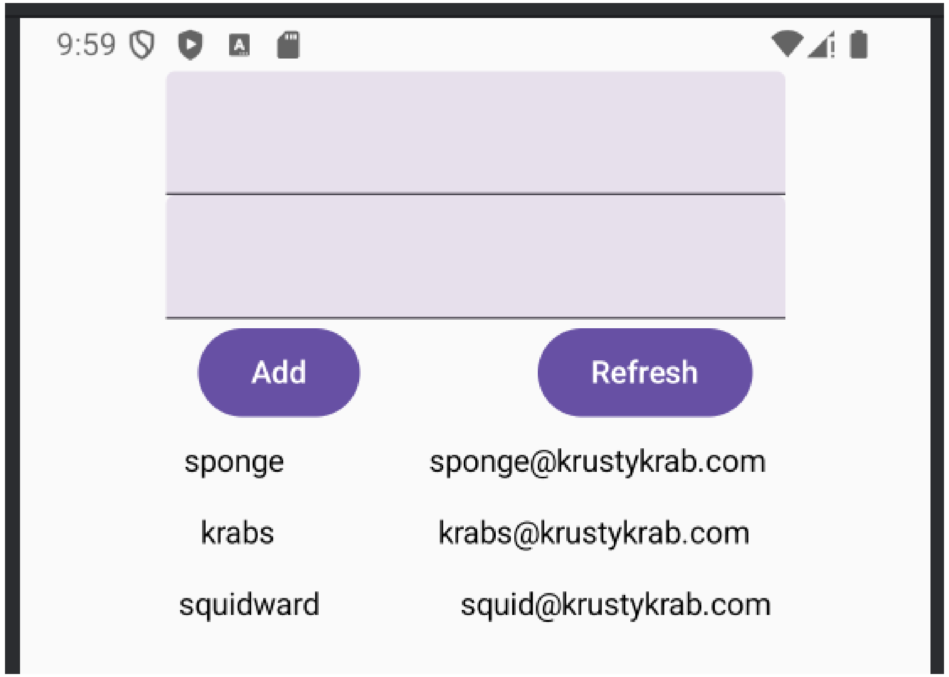
- Lastly, we inject our *Repository* into our state holder class
  - We are now ready to build our UI



```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    enableEdgeToEdge()
    setContent {
        val usersState = remember { UsersState(userRepo) }
        Box(modifier = Modifier.safeDrawingPadding()) {
            MainContent(usersState)
        }
    }
}
```

Putting it all together (cont.)

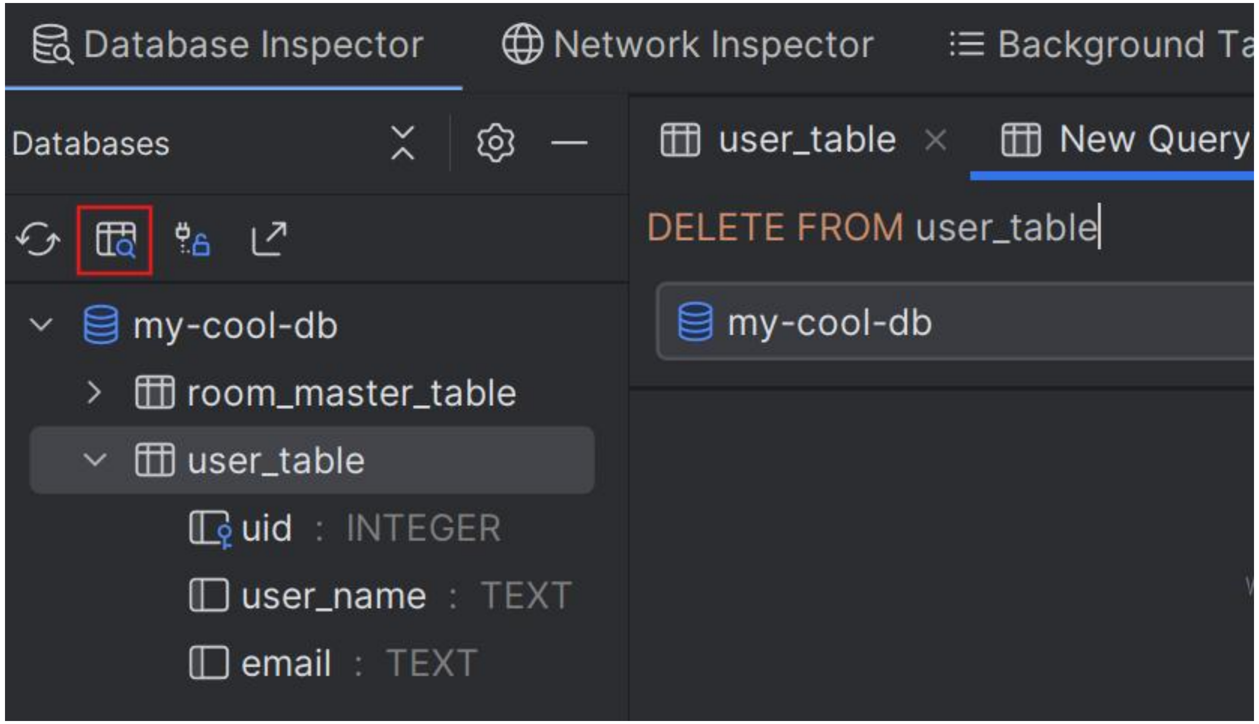
- See if you can finish off the UI
  - The data should now persist within our local database
  - Try closing and reopening the app to see for yourself



sponge	sponge@krustykrab.com
krabs	krabs@krustykrab.com
squidward	squid@krustykrab.com

App Inspector

- See a GUI of your database
  - View->Tools Windows->App Inspection
  - You can also call queries directly from the Database Inspector



```
DELETE FROM user_table
```