Lecture 9

COMP 3717- Mobile Dev with Android Tech

第9讲

COMP 3717 - 使用Android技术进行移动开发

State hoisting

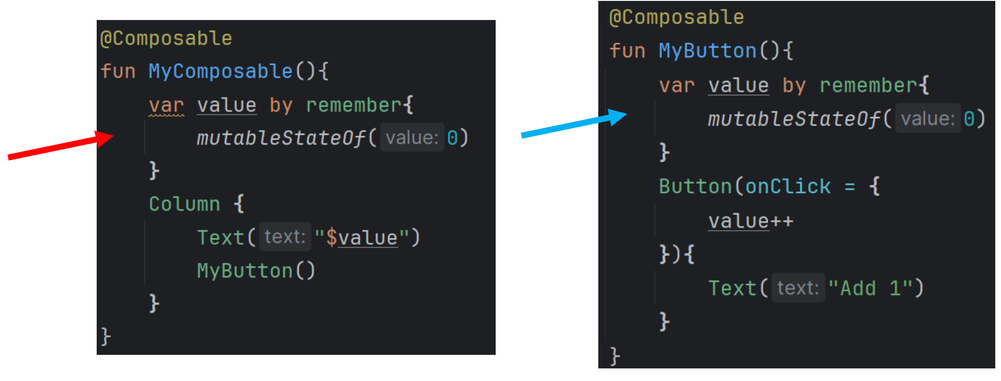
- Moving state to the *lowest common ancestor*
  - Maintains a clear single source of truth
  - Encourages stateless composables
  - Promotes reusability and more maintainable code
  - Avoids unnecessary recompositions

状态提升

- 将状态移至 最近公共祖先
  - 保持清晰的单一数据源
  - 鼓励使用无状态可组合项
  - 促进代码的可重用性和可维护性
  - 避免不必要的重组

Single source of truth

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

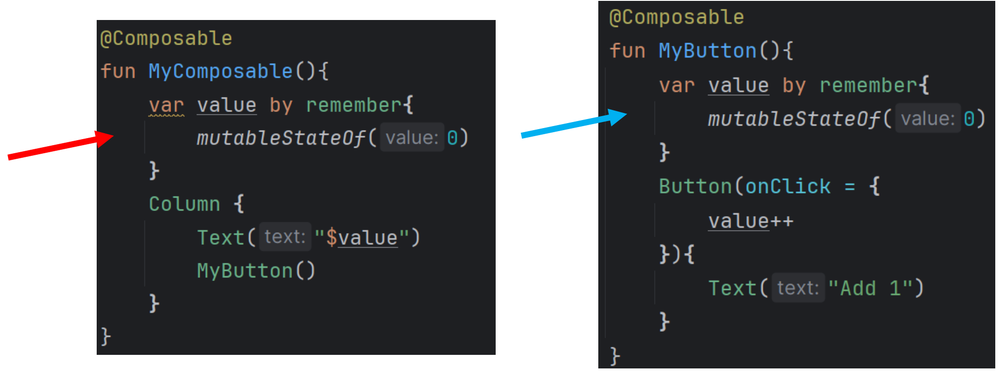


```
@Composable
fun MyComposable(){
    var value by remember{
        mutableStateOf( value: 0)
    }
    Column {
        Text( text: "$value")
        MyButton()
    }
}
```

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

单一事实来源

- 以下展示了缺乏单一事实来源的情况
  - 来源 1
  - 来源 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")
    }
}
```

单一数据源（续）

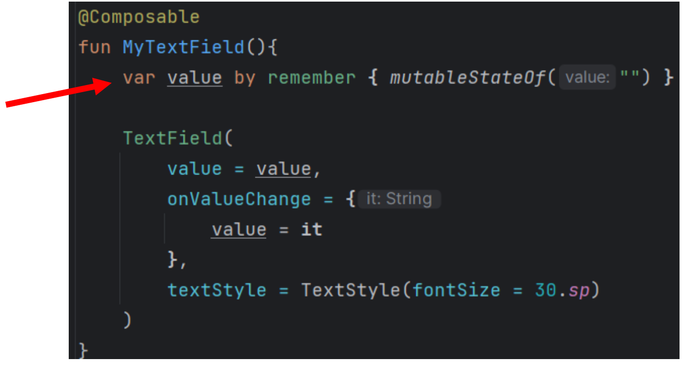
- 通过重构前面的示例，我们现在拥有了一个 单一数据源

```
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

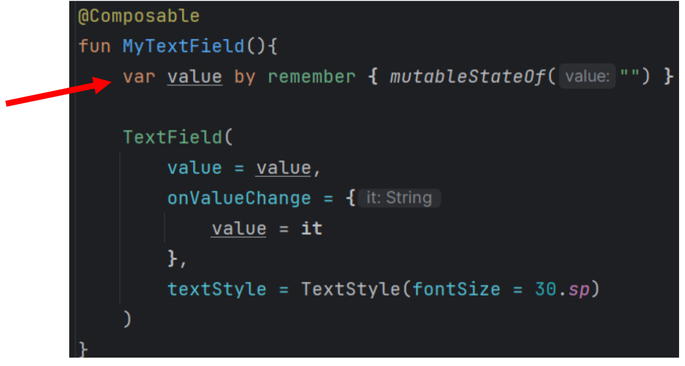
- Here is another example of a *single source of truth*
  - A composable that contains state is considered *stateful*



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

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

有状态的

- 这是另一个 单一可信来源 的示例
  - 一个包含状态的可组合项被视为有状态的



```
@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

无状态

- 当我们提升状态时，我们使可组合项变为无状态

- 状态变量通常被替换为
  - 正在读取的当前值内容
  - 用于设置值的事件回调

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)
    )
}
```

无状态（续）

- 当我们提升状态时，会使可组合项变为无状态
  - 无状态的可组合项不持有也不修改状态

```
@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 }
    }
}
```

状态提升（续）

- *MyTextField*现在变得更加解耦
- 我们可以使用不同的值和事件回调来重用它

```
@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

状态持有者

- 通常是一个普通的类或 *ViewModel*
- 当您的状态和逻辑在可组合项内部难以维护时使用
- 类型
  - UI 逻辑状态持有者
  - 业务逻辑状态持有者

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 逻辑状态持有者

- 在此场景中，状态持有者包含 UI 状态和逻辑
  - 通常是一个普通类
- 可组合项的职责仅仅是负责 UI 元素的生成
  - 这符合关注点分离原则

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 = "")
}
```

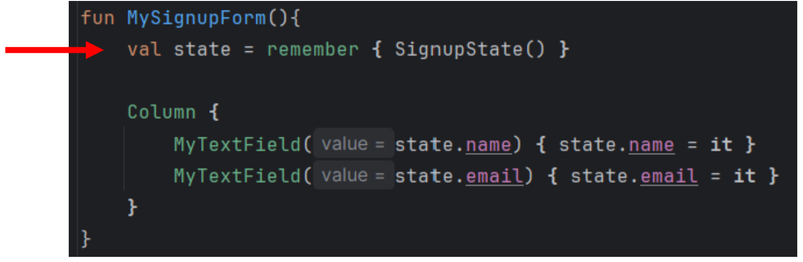
UI 逻辑状态持有者（续）

- 通常，当一个可组合项具有 多个状态对象 或逻辑过于复杂时是否应将状态 提升 到状态持有者中

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

UI logic state holder (cont.)

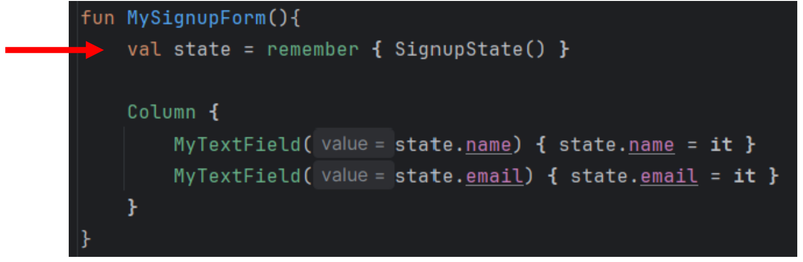
- You can then use one object to manage all your state within the composable



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

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

UI 逻辑状态持有者（续）

- 然后可以使用一个对象来管理可组合项中的所有状态

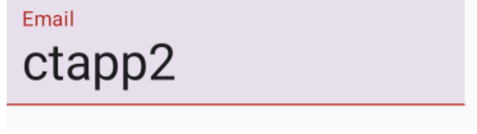


```
fun MySignupForm(){
    val state = remember { SignupState() }

    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

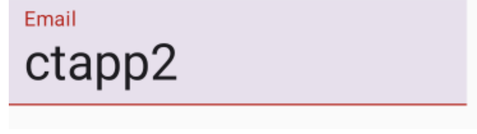
```
var email = mutableStateOf( value: "")
val onEmailChanged:(String)->Unit = {
    email.value = it
    invalidEmail = !email.value.contains( other: "@")
}
var invalidEmail = false
```



UI 逻辑

- 如何展示内容以及用户的体验方式
  - 例如，如果文本不包含 @ 字符，则将 *TextField* 高亮为红色

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



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

业务逻辑状态持有者

- 在此场景中，状态持有者包含业务状态和逻辑
  - 可以是一个普通类或一个 *ViewModel*
- 一个协调数据层与 UI 层之间应用程序数据的中间组件
  - 应用程序数据：在我们的应用中生成、使用和存储的信息

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

业务逻辑

- 数据在被数据层处理之前所需遵循的规则和要求
  - 用例（领域）
  - 验证

- 将业务理解为开发该应用的公司或组织

Layers

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

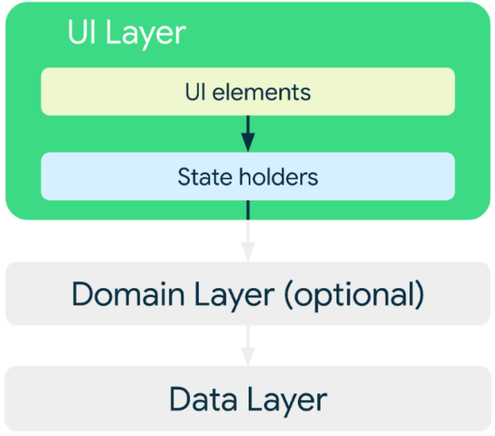
- The two basic layers are the UI layer and Data layer
  - UI layer
    - UI elements (composables)
    - State holders (UI and Business)
  - Data layer
    - Data sources
    - Repositores

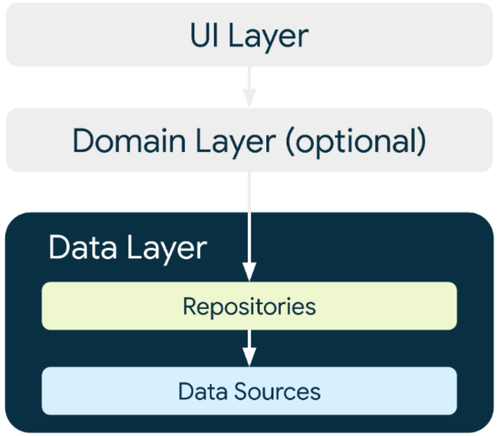
层级

- 当我们的应用程序开始扩展时，需要保持架构的清晰性
  - 完全解耦各层级

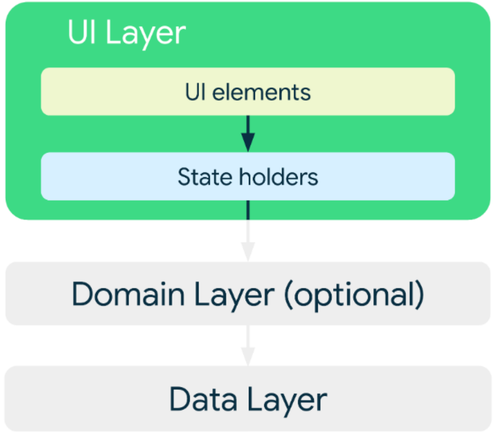
- 两个基本层级是 UI 层和数据层
  - UI 层
    - UI 元素（可组合项）
    - 状态持有者（UI 和业务）
  - 数据层
    - 数据源
    - 仓库

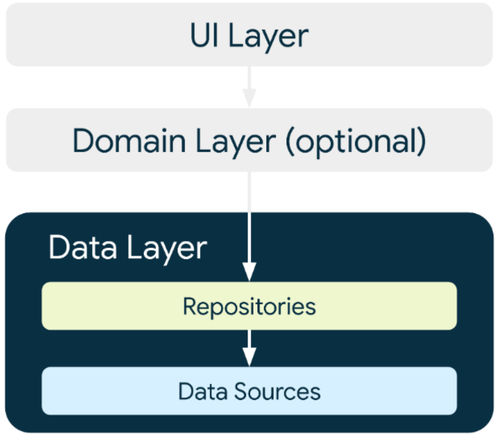
Layers (cont.)





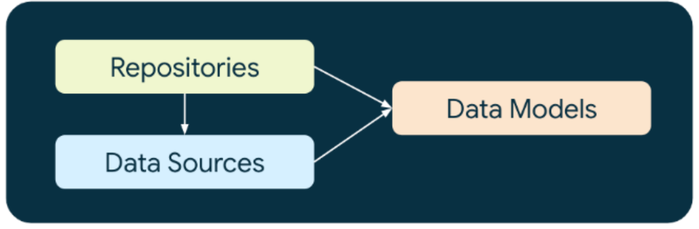
图层（续）





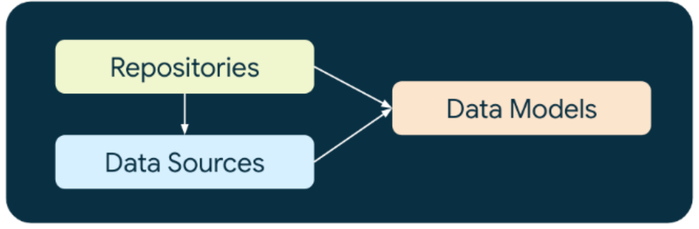
Data layer

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



数据层

- 数据层包含三个重要部分
  - 数据源
  - 仓库
  - 数据模型



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

数据层（续）

- 一个数据层可以包含一个或多个数据源
  - 本地或远程

- 本地数据源
  - File
    - 适用于存储原始的复杂数据
  - 本地数据库
    - 适用于存储具有查询功能的结构化和关系型数据
  - DataStore（Jetpack 库）
    - 适用于存储小型且简单的数据集

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

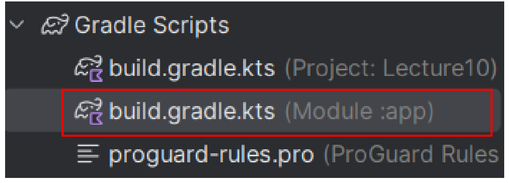
Room

- Room 是一个 Jetpack 库，提供对本地 SQLite 数据库（关系型）的访问

- Room 中的主要组件包括：
  - 数据实体
  - 数据访问对象
  - 数据库类

Room dependencies

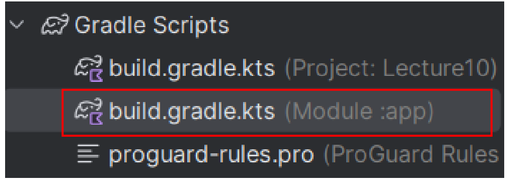
- Update your *Module-level build.gradle* with *Kotlin Symbol Processing*




```
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"
}
```

房间依赖关系

- 更新你的 模块级 build.gradle 以使用Kotlin Symbol Processing




```
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 依赖关系（续）

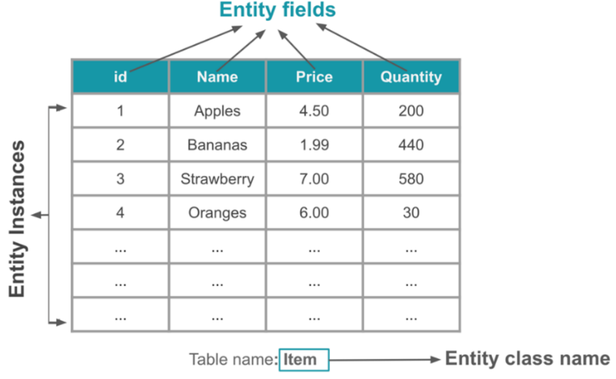
- 然后在底部添加 Room 依赖关系

```
dependencies {

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

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

- 同步你的项目并运行应用，检查是否正常工作

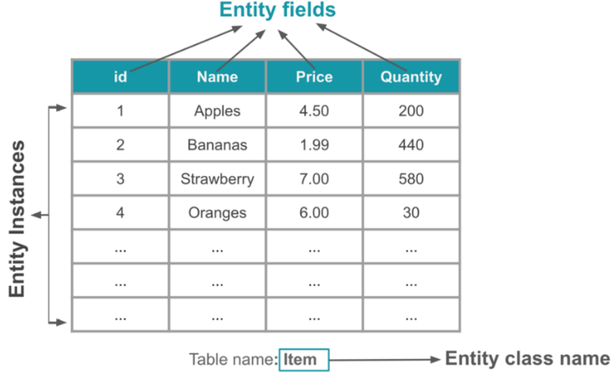
Room (cont.)

- Data entities represent tables in your database
- Each instance of an Entity class represents a row in the table



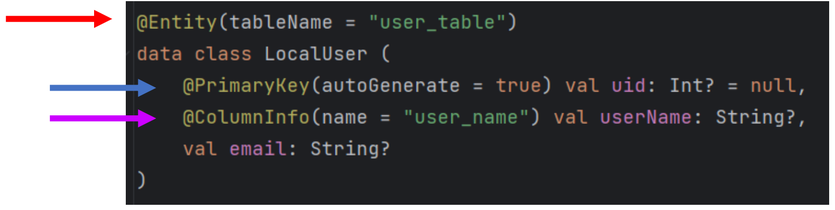
房间（续）

- 数据实体表示数据库中的表
- 实体类的每个实例代表表中的一行



Room (cont.)

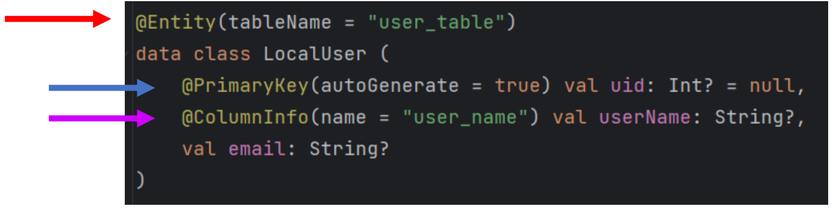
- *@Entity* marks a class as a database Entity class
- @PrimaryKey marks a field as the primary key
  - Every entity instance must have a primary key
- Each field is represented as a column in the database
  - *@ColumnInfo allows us to provide a custom name for it*



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

房间（续）

- *@Entity*将一个类标记为数据库实体类
- @PrimaryKey将一个字段标记为主键
  - 每个实体实例都必须有一个主键
- 每个字段在数据库中表示为一列
  - *@ColumnInfo* 允许我们为其提供自定义名称



```
@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

房间（续）

- 数据访问对象（DOA）提供了应用程序用于与数据库交互的增删改查功能
  - 插入、查询、更新、删除等

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)
}
```

房间（续）

- Room 库提供便捷的注解而无需要求你编写 SQL 语句

```
@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
}
```

房间（续）

- 使用*@Database*注解的数据库类包含该数据库
  - 持久化数据的主要访问入口
- 它定义了实体列表
  - 在此示例中，我们只有LocalUser

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

Room (cont.)

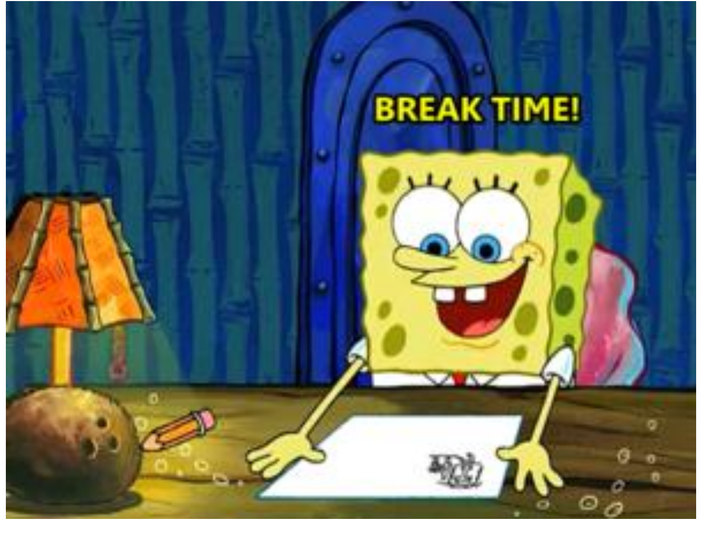
- The database class also provides the instances of the DAOs
- The DAOs are abstract because Room auto creates the implementation for us at compile time

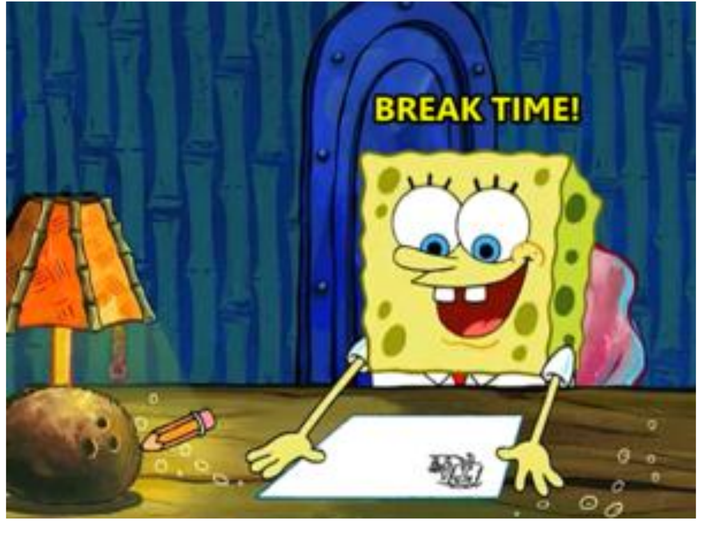
```
@Database(entities = [LocalUser::class], version = 1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

房间（续）

- 数据库类还提供了DAO 实例
- DAO 是抽象的，因为 Room 会在编译时自动为我们创建实现


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


BREAK TIME!


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 {
```

单例

- 我们只希望数据库有一个实例，因此让我们使用单例
  - 单例：一种设计模式，确保一个类只有一个实例
- Kotlin 通过使用 关键字来减少创建单例类时的大量模板代码object。

```
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,
```

应用上下文

- 应用上下文用于获取有关应用程序的信息
- Room 数据库存储在设备上特定于应用本身的目录中
  - 创建数据库实例时，我们将传入 应用上下文

```
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()
```

房间（续）

- Room 数据库默认无法在主线程上执行查询
  - 这可能会导致主线程冻结或显著变慢
- 但在此课中，我们将 允许这样做

```
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()
    }
}
```

仓库

- 现在我们已经创建了数据源，需要一个仓库来访问它

```
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()
```

业务逻辑状态持有者（续）

- 跳回到 UI 层，我们需要为应用程序数据创建一个状态持有者
- 首先，我们可以创建一些 状态来反映当前的用户

```
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())
    }
}
```

业务逻辑状态持有者（续）

- 这里有两个函数
  - 一种在数据库中插入实体的方法
  - 一种设置状态以包含数据库中当前用户的方法


```
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()) }
```

整合所有内容

- 然后我们需要初始化数据库和仓库
- 这应该在 *MainActivity* 中完成，位于 *onCreate* 之外

```
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)
        }
    }
}
```

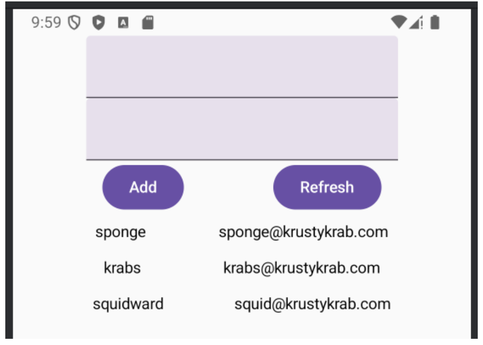
整合全部内容（续）

- 最后，我们将 *Repository* 注入到我们的状态持有类中
  - 我们现在可以开始构建我们的用户界面了

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

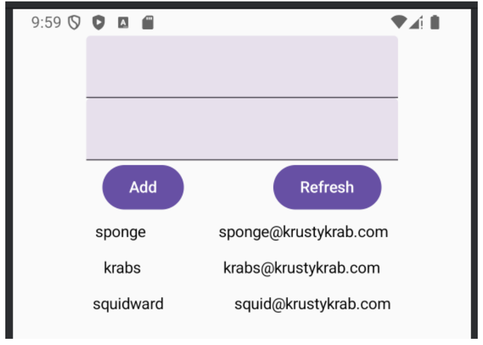
Putting it all together (cont.)

- 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



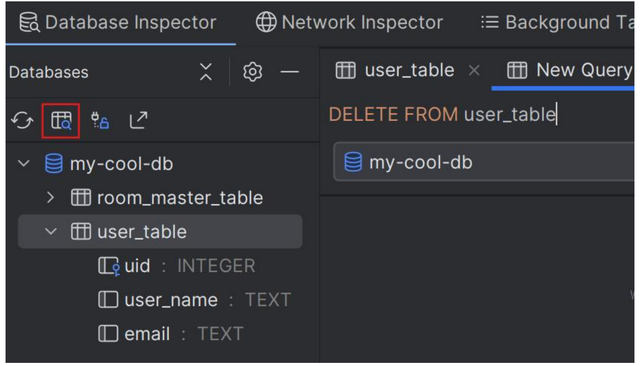
整合所有内容（续）

- 看看你是否能完成 UI 的编写
  - 数据现在应已保存在我们的本地数据库中
  - 尝试关闭并重新打开应用程序，亲自验证一下



App Inspector

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



应用检查器

- 查看数据库的图形用户界面
  - 视图->工具窗口->应用检查
  - 您还可以直接从数据库检查器调用查询