



UNIVERSIDADE  
FEDERAL DO CEARÁ

CAMPUS QUIXADÁ

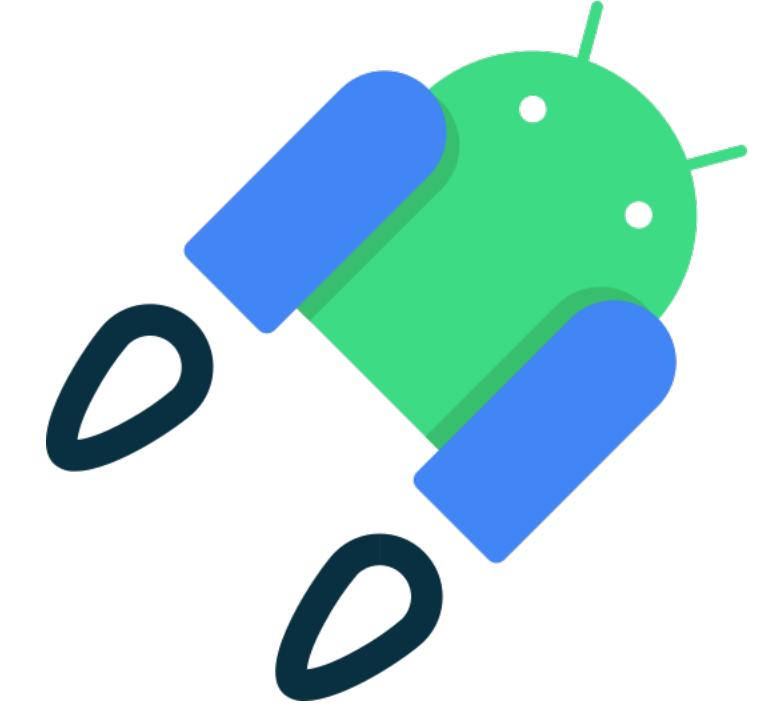
# Persistência

**QXD0102 - Desenvolvimento para Dispositivos Móveis**

Prof. Bruno Góis Mateus ([brunomateus@ufc.br](mailto:brunomateus@ufc.br))

# Conteúdo

- Room
- Kotlin Flow
- Exemplo prático



# Room



# Room

- Diversos apps podem se beneficiar da persistência desses dados localmente
- Ex: Armazenar as partes de dados relevantes em cache
  - Quando o dispositivo não conseguir acessar a rede, o usuário ainda poderá navegar pelo conteúdo enquanto estiver off-line
  - Todas as modificações de conteúdo feitas pelo usuário serão sincronizadas com o servidor quando o dispositivo ficar on-line novamente

# Room

- Historicamente o Android utiliza o banco SQLite para armazenamento local
- Alguns dos principais problemas ao usarmos o SQLite são:
  - As consultas não são verificadas em tempo de compilação
  - Mudanças no esquema do banco devem ser gerenciadas manualmente
  - É necessário fazer uso de código boilerplate para converter os resultados das consultas em objetos do domínio da app
- É altamente recomendável usar o Room em vez do SQLite

# Room

## Room

- ORM, Object Relational Mapping library
- Biblioteca de persistência que prover uma camada de abstração sobre o SQLite
- Verificação de consultas em tempo de compilação
- Série de anotações para minimizar códigos repetitivos
- Migrações simplificadas

# Room

- Existem 3 principais componentes no Room
  - A classe Banco de Dados
    - Referência o banco de dados e serve de ponto principal de acesso ao dados
  - Entidades do dados
    - Representam as tabelas do banco de dados
  - DAOs (*Data Access Objects*)
    - Prover métodos para que sua app realize, consultas, inserções, atualização e remoções no banco de dados

# Room

- A classe banco de dados prover instâncias de DAOs
- Os DAOs permitem o acesso aos dados associados as entidades
- As entidades são utilizadas nas consultas para atualizar o banco de dados

Room Database

Rest of The App

# **Room**

## **Entidade**



# Room

## *Data access object (DAO)*

```
@Dao  
interface UserDao {  
  
}
```

# Room

## A classe banco de dados

- Deve satisfazer as seguintes condições:
  - A classe deve ser anotado com `@Database` e incluir o vetor de entidades, listando todas as entidades associadas a base de dados
  - Deve ser uma **class abstrata** que herdar de `RoomDatabase`
  - Para cada **classe DAO**, a classe deve definir **um método abstrato** sem parâmetros que **retorna uma instância do DAO**

# Room

## A classe banco de dados

```
val db = Room.databaseBuilder(  
    applicationContext,  
    AppDatabase::class.java, "database-name"  
).build()
```

```
val userDao = db.userDao()  
val users: List<User> = userDao.getAll()
```

# Room

## Main-safety

- O Android **não permite** acesso ao banco de dados na **Main Thread**
- Para nossa sorte o Room inclui integrações com diversos frameworks

Tipo de consulta	Funcionalidades do Kotlin	RxJava	Guava	Jetpack Lifecycle
One-shot write	Coroutines (suspend)	Single<T>, Maybe<T>, Completable	ListenableFuture<T>	N/A
One-shot read	Coroutines (suspend)	Single<T>, Maybe<T>	ListenableFuture<T>	N/A
Observable read	Flow<T>	Flowable<T>, Publisher<T>, Observable<T>	N/A	LiveData<T>

# Room

## One-Shot queries

- Operações que são executadas uma única vez e recuperam o dado no momento da execução

```
@Dao
interface UserDao {
    @Insert(onConflict = OnConflictStrategy.REPLACE)
    suspend fun insertUsers(vararg users: User)

    @Update
    suspend fun updateUsers(vararg users: User)

    @Delete
    suspend fun deleteUsers(vararg users: User)

    @Query("SELECT * FROM user WHERE id = :id")
    suspend fun loadUserById(id: Int): User

    @Query("SELECT * from user WHERE region IN (:regions)")
    suspend fun loadUsersByRegion(regions: List<String>): List<User>
}
```

# Room

## Observable queries

- São operações de leitura que emitem um novo valor sempre que houver alguma mudança na tabela relacionada a consulta

```
@Dao
interface UserDao {
    @Query("SELECT * FROM user WHERE id = :id")
    fun loadUserById(id: Int): Flow<User>

    @Query("SELECT * from user WHERE region IN (:regions)")
    fun loadUsersByRegion(regions: List<String>): Flow<List<User>>
}
```

A consulta é re-executada sempre que a tabela é atualizada, independentemente da linha atualizada fazer parte ou não do resultado. Para evitar notificações desnecessária, utilize o operador `distinctUntilChanged()`

# Room

- Mais informações
  - Documentação oficial

# Kotlin Flow

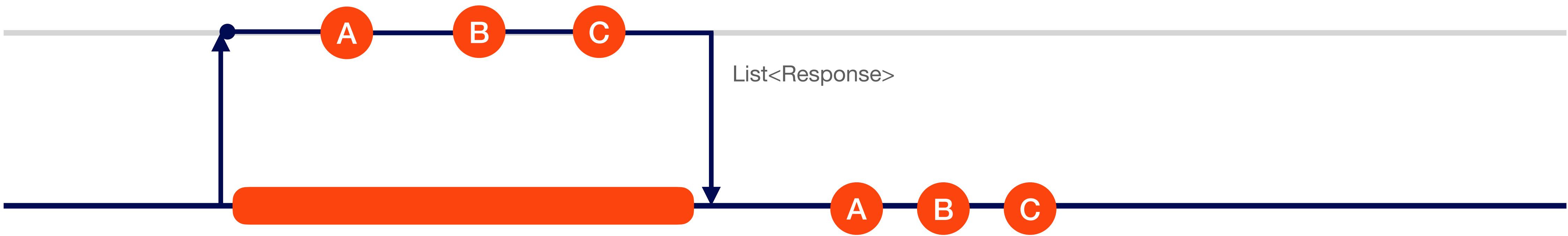


# Kotlin Flow

- Com o advento das **Coroutines**, desenvolvedores passaram a adotá-las devido a sua **simplicidade** e a **concorrência estruturada**
  - Resolveram o problema do **callback hell**
  - Não provém uma **API reativa** similar ao RxJava

# Kotlin Flow

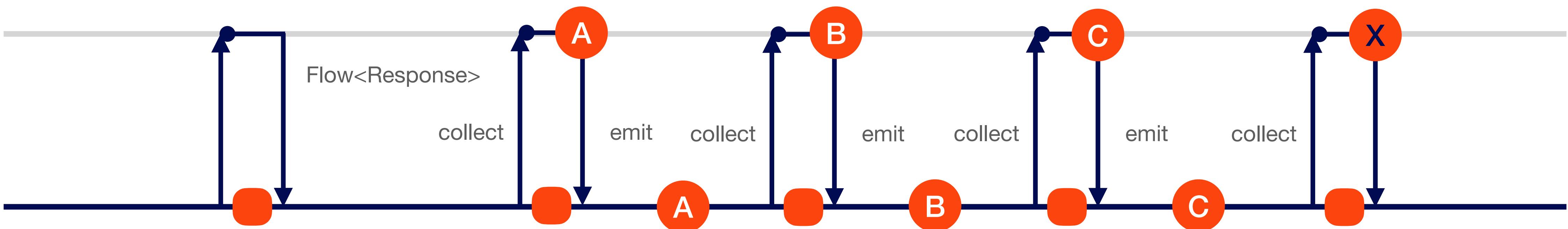
```
suspend fun foo(): List<Response> = buildList {  
    add(compute("A"))  
    add(compute("B"))  
    add(compute("C"))  
}
```



```
fun main() = runBlocking {  
    val list = foo()  
    for (x in list) println(x)  
}
```

# Kotlin Flow

```
fun foo(): Flow<Response> = flow {
    emit(compute("A"))
    emit(compute("B"))
    emit(compute("C"))
}
```



```
fun main() = runBlocking {
    val flow = foo()
    flow.collect {x -> println(x)}
}
```

# Kotlin Flow

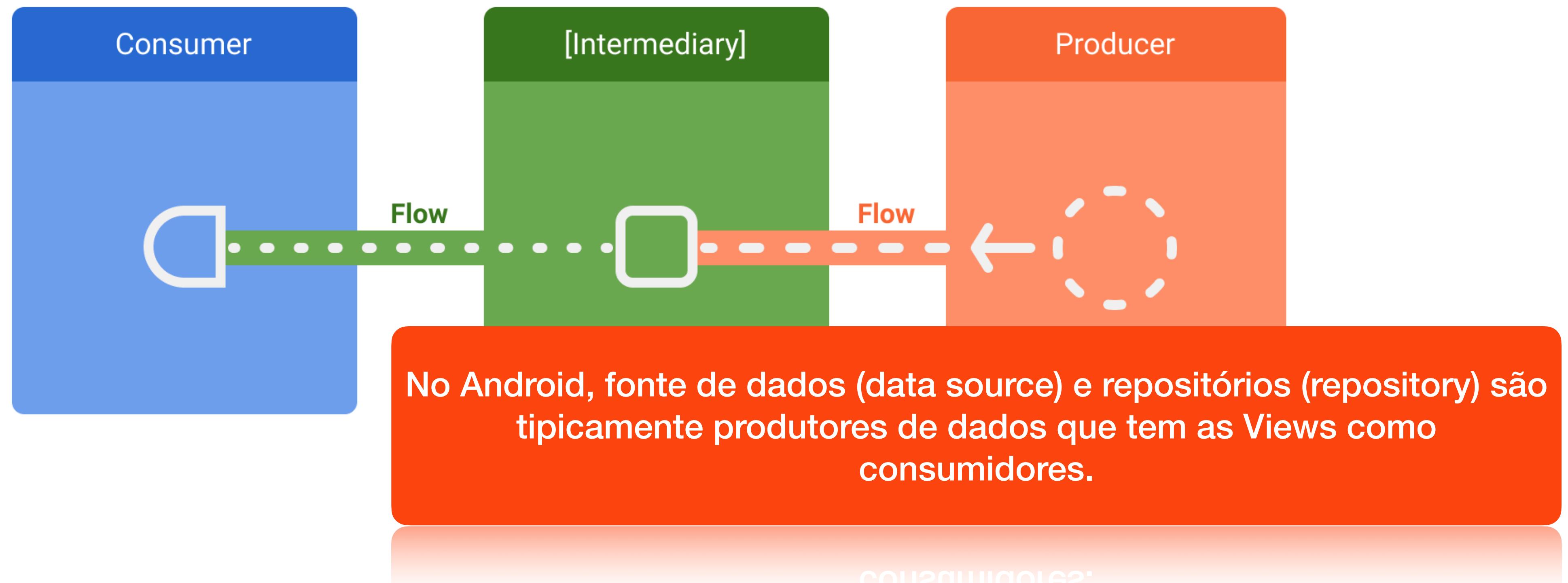
## Kotlin Flow é reativo

- Implementação da especificação de *reactive stream*
  - Construído tendo **Coroutines** como base
- Gerenciamento assíncrono de streams de dados
- **Frio (Cold)**, não começar a emitir enquanto não houver um observer
- Null safety
- Suporte ao Kotlin multiplataforma

# Kotlin Flow

## Flow no Android

- Consomem os valores a partir da stream
- Podem modificar os valores emitidos ou a própria stream
- Produz os dados que são adicionados a stream
- Graças as Coroutines, a eles podem produzir dados assincronamente



# Kotlin Flow

## Por que usar Flow?

- Apesar de existirem outras implementações da *reactive stream specification*, como a RxJava, devido ao fato delas serem **JVM-specific**, elas não podem ser usadas em **projetos Kotlin multi-platform**
  - Flow é parte do **Kotlin**, então é uma escolha ideal para esse tipo de projeto
- Devido ao fator de operarem tendo **Coroutines como base**, **Kotlin Flow** possui **menos operadores** e que ainda assim são simples
  - Herdam todas outra características de Coroutines, como a *structured concurrency* e *cancellation*

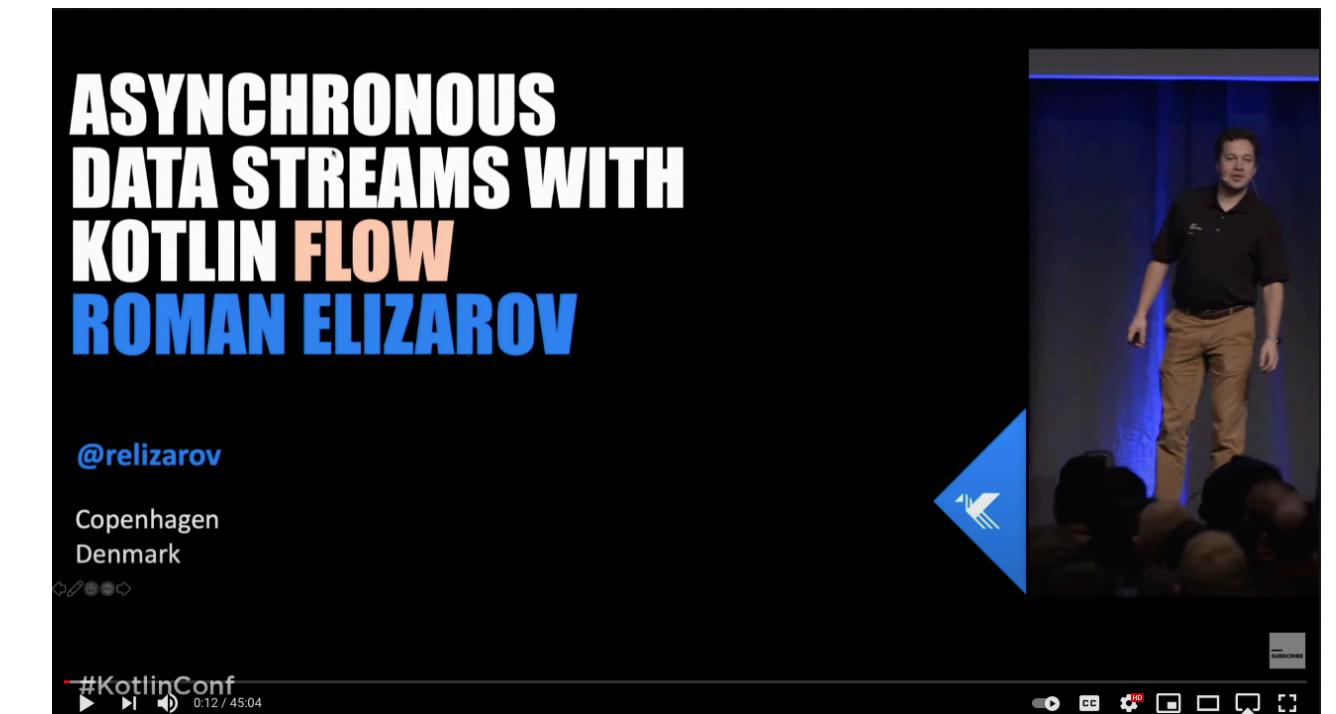
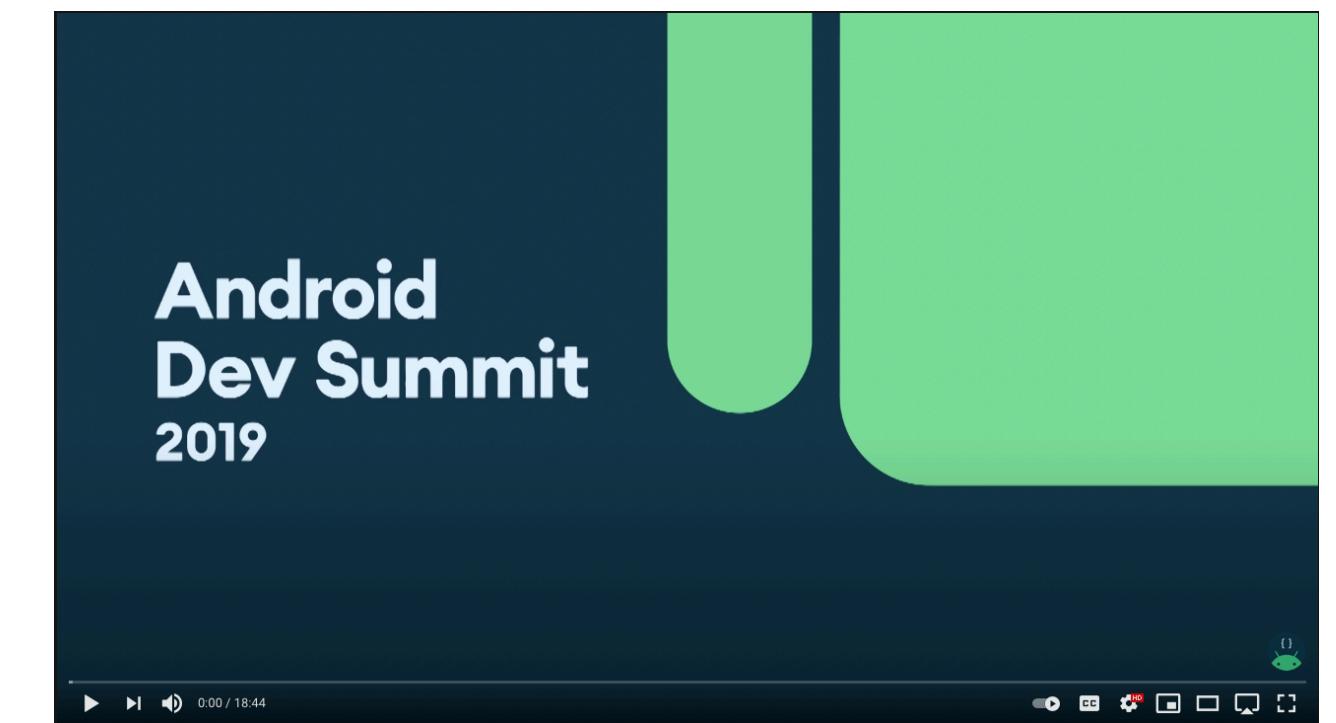
# Kotlin Flow

## Por que usar Flow?

- É possível substituir o uso de LiveData (*platform-dependent*) ao usar o StateFlow/SharedFlow em todas as camadas de uma app (view, model, storage)
- LiveData é dependente do ciclo de vida do Android Lifecycle e portanto não é o ideal usá-la fora do contexto de Views/ViewModels

# Kotlin Flow

- Mais informações
  - [Go with the Kotlin Flow](#)
  - [Kotlin Flow for Android: Getting Started](#)
  - [Reactive Streams on Kotlin: SharedFlow and StateFlow](#)
  - [Migrating from LiveData to Kotlin's Flow](#)
  - [Documentação oficial](#)

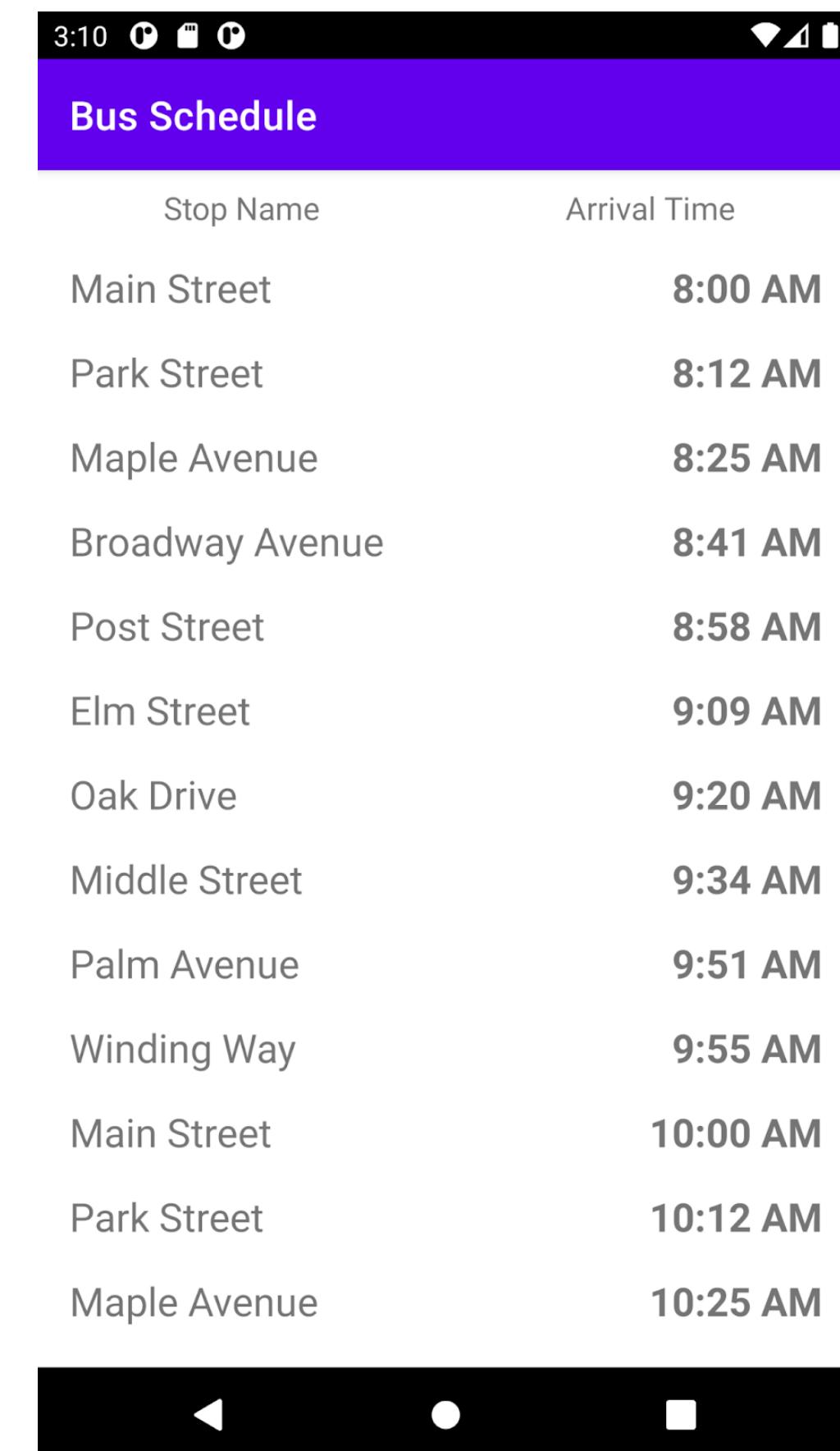


# Room e Flow

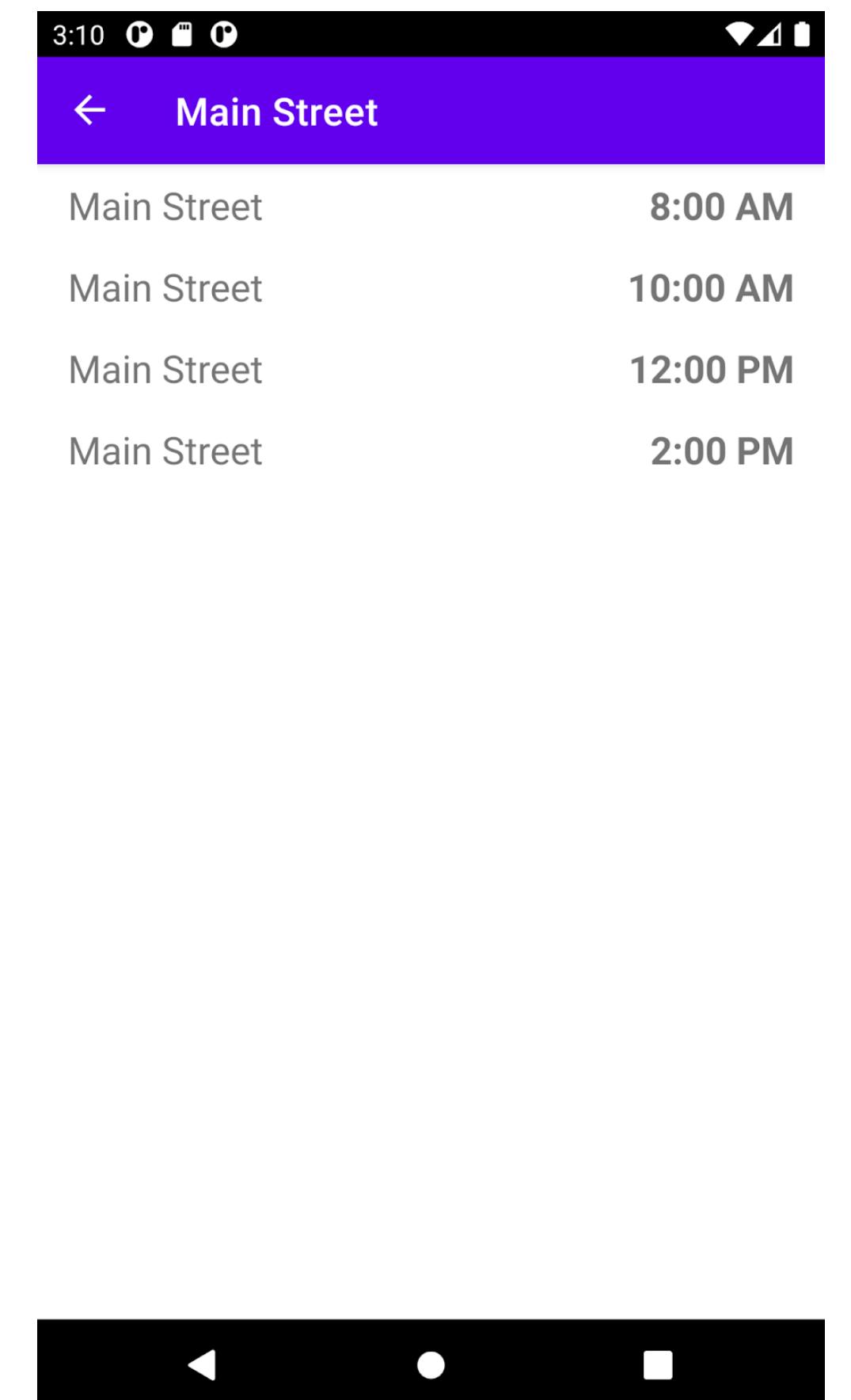


# Room e Flow

- Codelabs:
  - Introdução ao Room e Flow



Stop Name	Arrival Time
Main Street	8:00 AM
Park Street	8:12 AM
Maple Avenue	8:25 AM
Broadway Avenue	8:41 AM
Post Street	8:58 AM
Elm Street	9:09 AM
Oak Drive	9:20 AM
Middle Street	9:34 AM
Palm Avenue	9:51 AM
Winding Way	9:55 AM
Main Street	10:00 AM
Park Street	10:12 AM
Maple Avenue	10:25 AM



Stop Name	Arrival Time
Main Street	8:00 AM
Main Street	10:00 AM
Main Street	12:00 PM
Main Street	2:00 PM

Por hoje é só