Introdução ao Room e ao fluxo

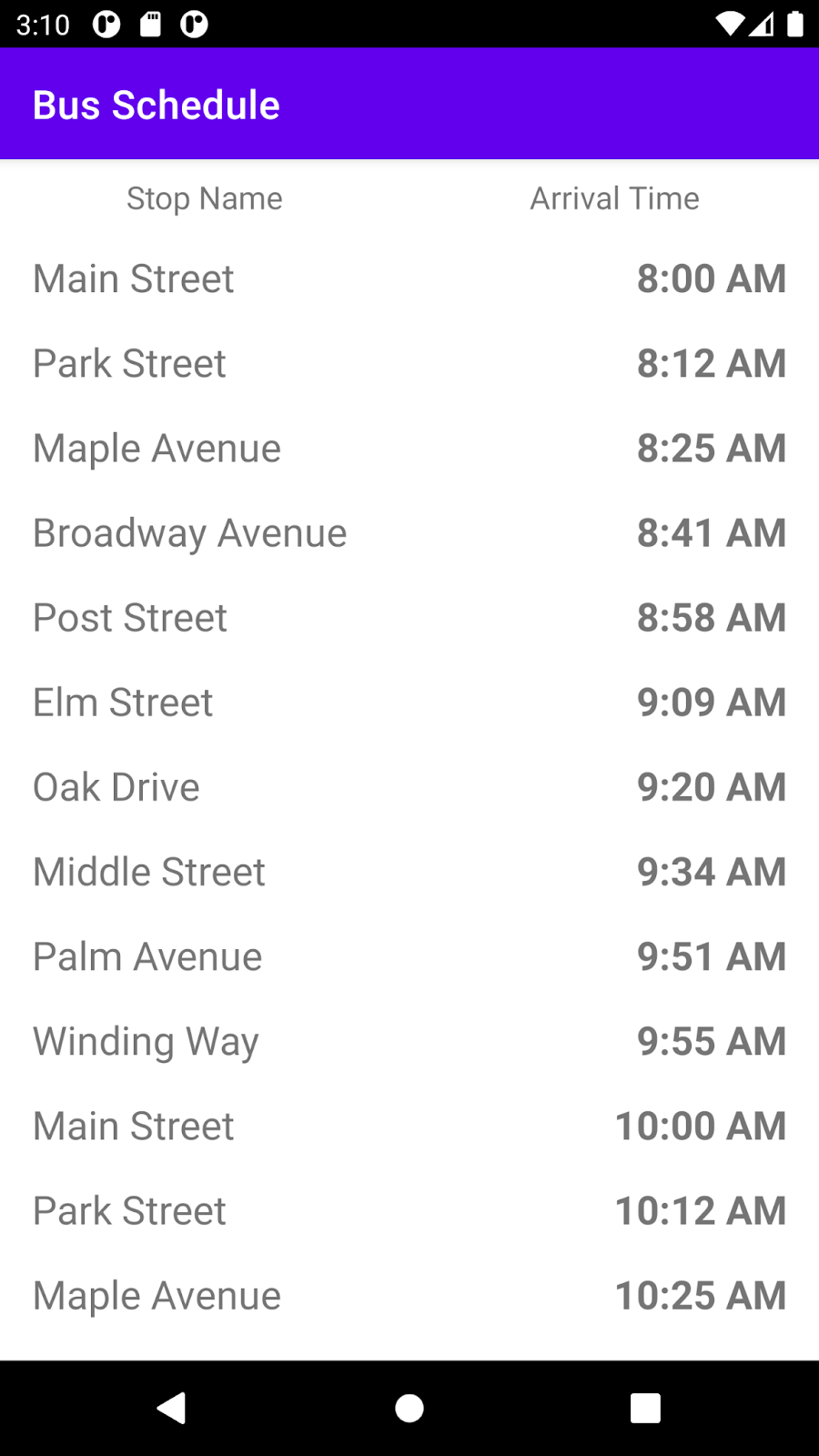
**About this codelab**

account\_circleWritten by Equipe de treinamento do Google Developers

[1. Antes de começar](https://developer.android.com/codelabs/basic-android-kotlin-training-intro-room-flow?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-5-pathway-1%3Fhl%3Dpt-br%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-intro-room-flow#0)

No [codelab anterior](https://developer.android.com/codelabs/basic-android-kotlin-training-sql-basics" \t "_blank), você aprendeu sobre os princípios básicos dos bancos de dados relacionais e sobre como ler e gravar dados usando os comandos SQL: SELECT, INSERT, UPDATE e DELETE. Saber trabalhar com bancos de dados relacionais é uma habilidade fundamental para ter em sua jornada de programação. Entender como os bancos de dados relacionais funcionam também é essencial para implementar a persistência de dados em um app Android. Você começará a fazer isso nesta lição.

Uma maneira fácil de usar um banco de dados em um app Android é com uma biblioteca chamada Room. O Room é uma biblioteca de mapeamento relacional de objeto (ORM, na sigla em inglês) que, como o nome indica, mapeia as tabelas em um banco de dados relacional em busca de objetos que podem ser usados no código Kotlin. Nesta lição, você se concentrará apenas em ler dados. Usando um banco de dados pré-preenchido, você carregará dados de uma tabela de horários de chegada de ônibus e os apresentará em uma RecyclerView.



Nesse processo, você aprenderá os princípios básicos de como usar o Room, incluindo a classe de banco de dados, o DAO, as entidades e os modelos de visualização. Você também conhecerá a classe ListAdapter, outra forma de apresentar dados em uma RecyclerView, e um fluxo, recurso da linguagem Kotlin semelhante ao LiveData, que permitirá que a IU responda a mudanças no banco de dados.

Pré-requisitos

* Familiaridade com a programação orientada a objetos e o uso de classes, objetos e herança em Kotlin.
* Conhecimentos básicos sobre bancos de dados relacionais e SQL, ensinados no [codelab Princípios básicos de SQL](https://developer.android.com/codelabs/basic-android-kotlin-training-sql-basics" \t "_blank).
* Experiência com corrotinas do Kotlin.

O que você aprenderá

Ao final desta lição, você saberá:

* representar tabelas de banco de dados como objetos Kotlin (entidades);
* definir a classe do banco de dados para usar o Room no app e pré-preencher um banco de dados com um arquivo;
* definir a classe DAO e usar consultas SQL para acessar o banco de dados com o código Kotlin;
* definir um modelo de visualização para permitir que a IU interaja com o DAO;
* usar um ListAdapter com uma visualização de reciclagem;
* os conceitos básicos do fluxo do Kotlin e como usá-lo para fazer com que a IU responda a mudanças nos dados.

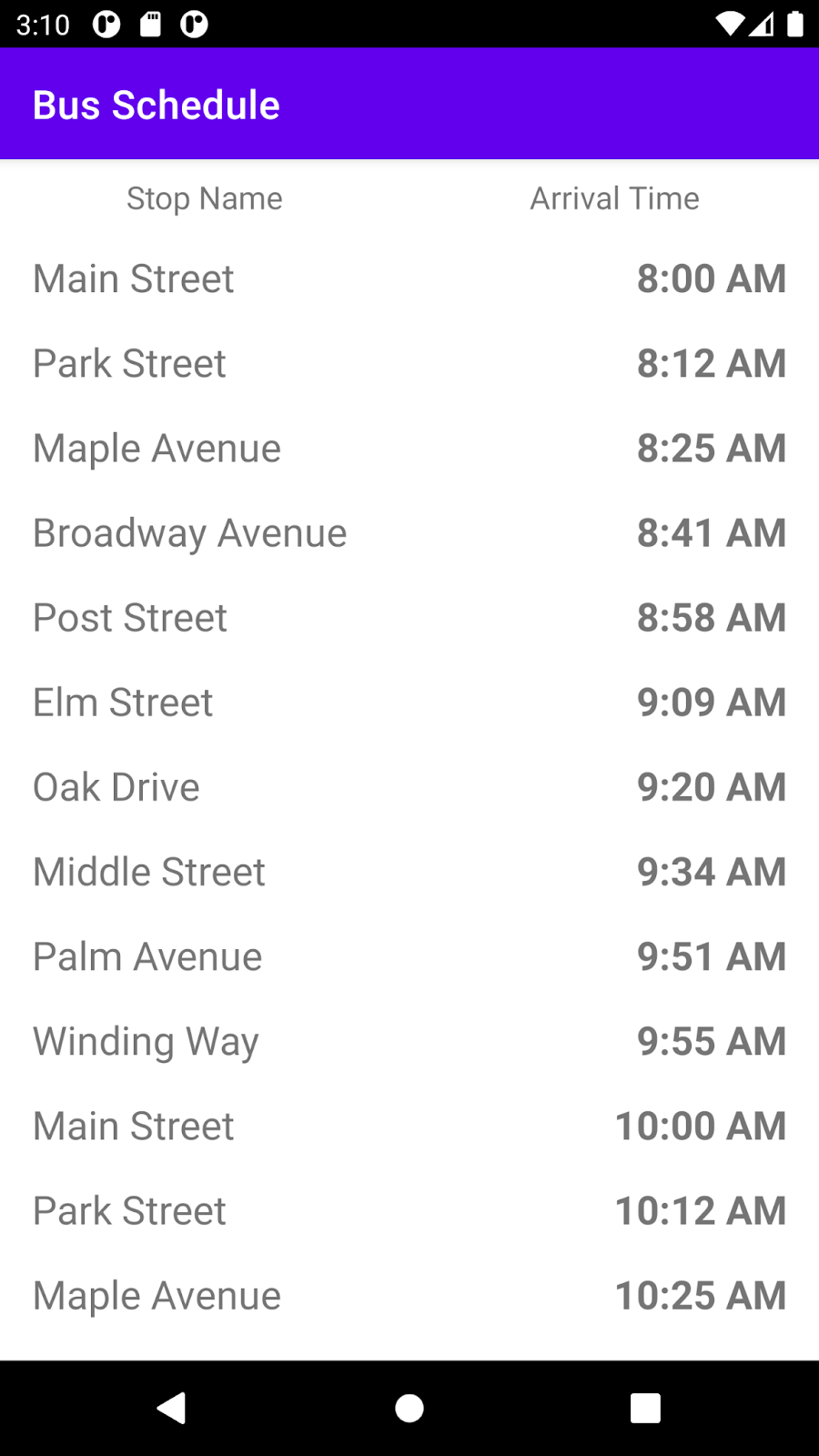
O que você criará

* Você lerá dados de um banco de dados pré-preenchido com o Room e os apresentará em uma visualização de reciclagem em um app simples de horários de ônibus.

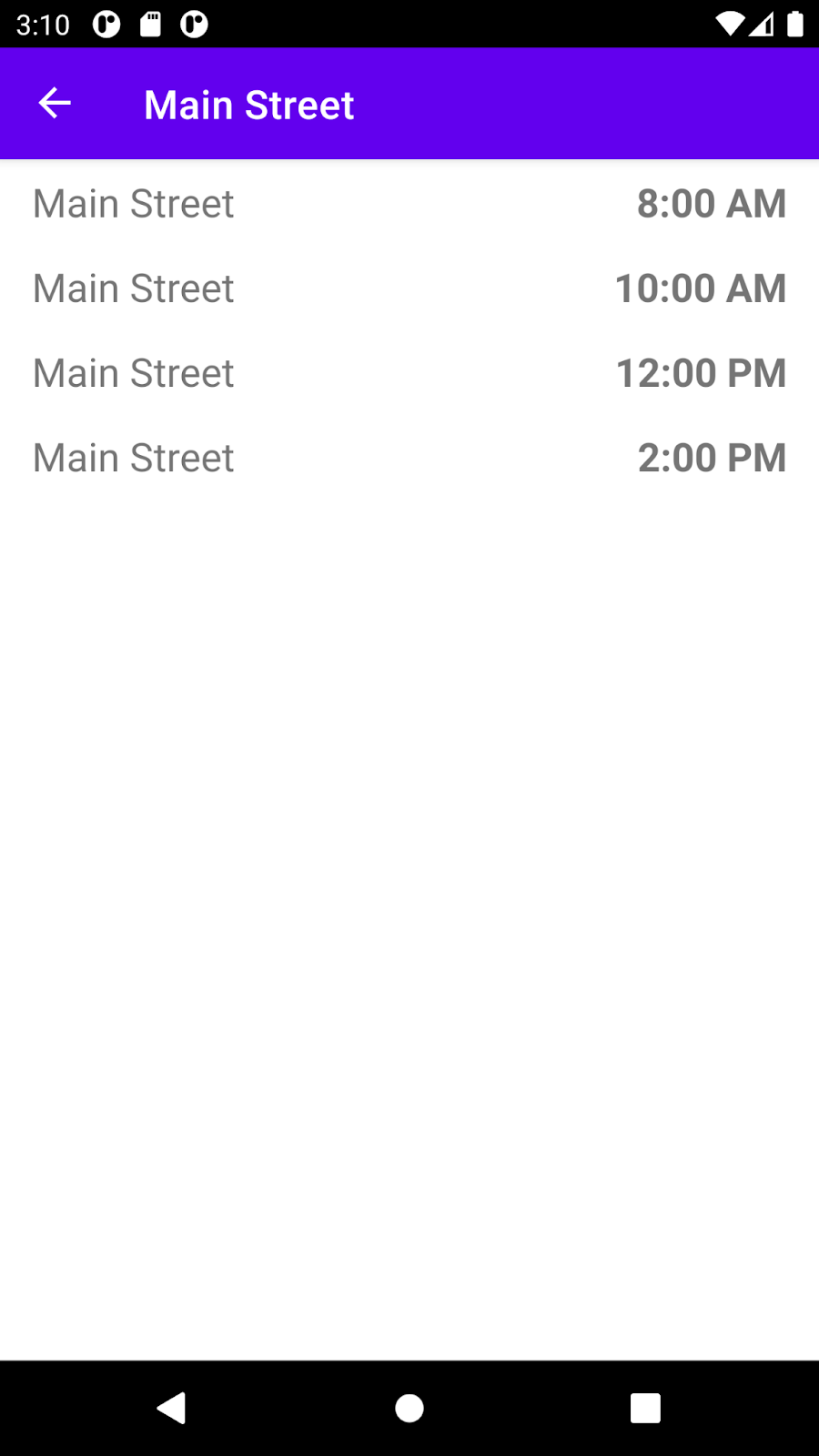
[**Back**](https://developer.android.com/codelabs/basic-android-kotlin-training-intro-room-flow?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-5-pathway-1%3Fhl%3Dpt-br%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-intro-room-flow)

[2. Primeiros passos](https://developer.android.com/codelabs/basic-android-kotlin-training-intro-room-flow?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-5-pathway-1%3Fhl%3Dpt-br%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-intro-room-flow#1)

O nome do app com que você trabalhará neste codelab é Bus Schedule. Ele apresenta uma lista de pontos de ônibus e horários de chegada, do horário mais próximo ao mais tarde.



Ao tocar em uma linha na primeira tela, o app exibe uma nova tela, que mostra somente os próximos horários de chegada para o ponto de ônibus selecionado.



As informações sobre os pontos de ônibus vêm de um banco de dados pré-empacotado com o app. Entretanto, no estado atual, nada será exibido quando o app for executado pela primeira vez. Você integrará o Room para que o app exiba o banco de dados pré-preenchido com os tempos de chegada.

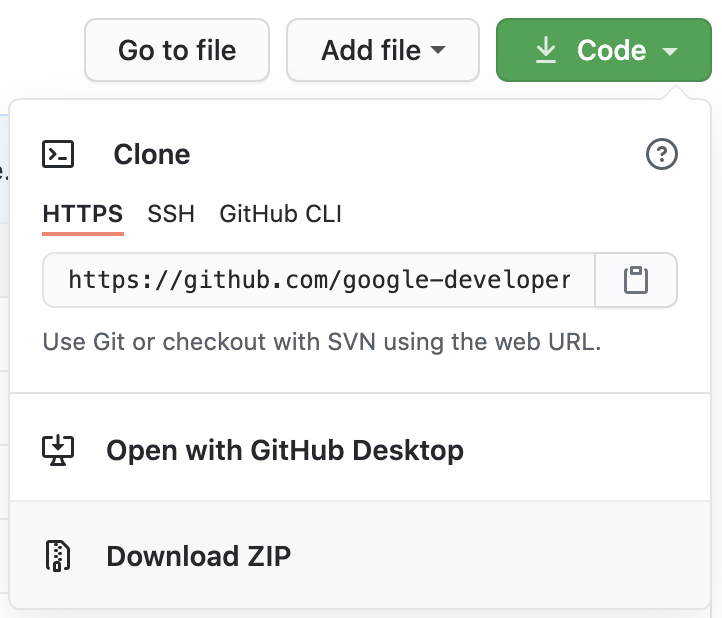
**URL do código inicial:**<https://github.com/google-developer-training/android-basics-kotlin-bus-schedule-app/tree/starter> (link em inglês)

**Ramificação: starter**

Para encontrar o código deste codelab e abri-lo no Android Studio, faça o seguinte.

Buscar o código

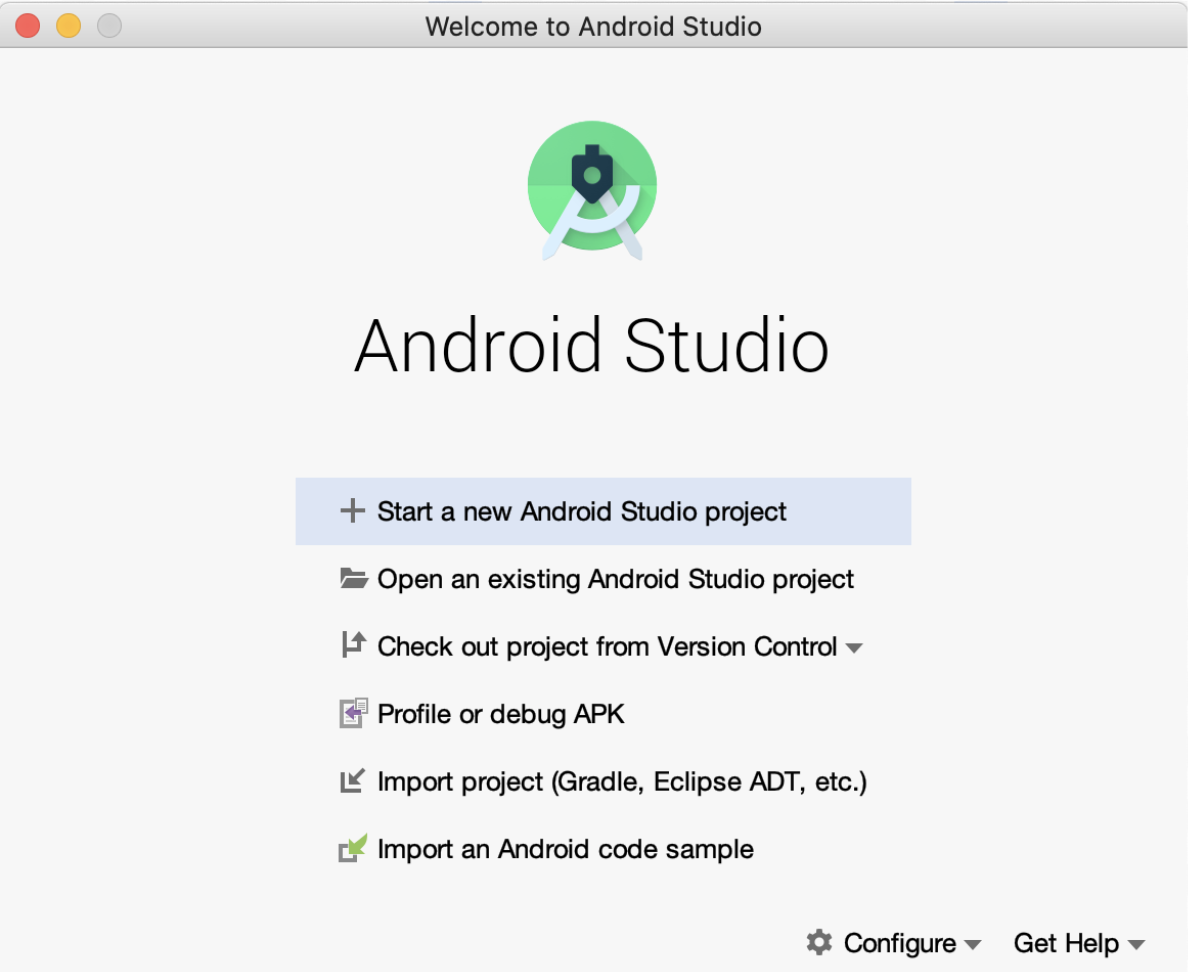
1. Clique no URL fornecido. Isso abrirá a página do GitHub referente ao projeto em um navegador.
2. Na página do GitHub do projeto, clique no botão **Code**, que exibirá uma caixa de diálogo.



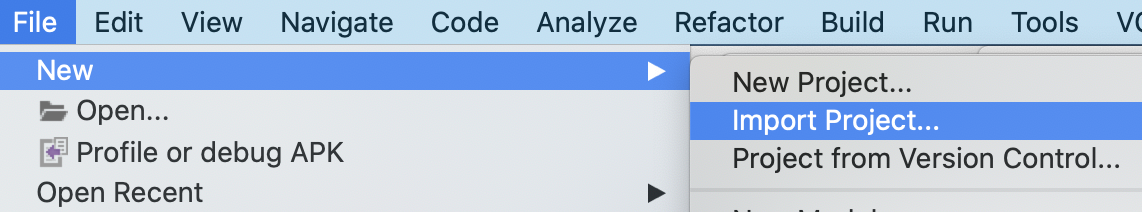
1. Na caixa de diálogo, clique no botão **Download ZIP** para salvar o projeto no seu computador. Aguarde a conclusão do download.
2. Localize o arquivo no computador, que provavelmente está na pasta **Downloads**.
3. Clique duas vezes no arquivo ZIP para descompactá-lo. Isso criará uma nova pasta com os arquivos do projeto.

Abrir o projeto no Android Studio

1. Inicie o Android Studio.
2. Na janela **Welcome to Android Studio**, clique em **Open an existing Android Studio project**.



Observação: caso o Android Studio já esteja aberto, selecione a opção **File** > **New** > **Import Project**.



1. Na caixa de diálogo **Import Project**, vá até a pasta do projeto descompactada, que provavelmente está na pasta **Downloads**.
2. Clique duas vezes nessa pasta do projeto.
3. Aguarde o Android Studio abrir o projeto.
4. Clique no botão **Run**  para criar e executar o app. Confira se ele é compilado da forma esperada.
5. Procure os arquivos do projeto na janela de ferramentas **Project** para ver como o app está configurado.

[**Back**](https://developer.android.com/codelabs/basic-android-kotlin-training-intro-room-flow?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-5-pathway-1%3Fhl%3Dpt-br%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-intro-room-flow)

[3. Adicionar dependência do Room](https://developer.android.com/codelabs/basic-android-kotlin-training-intro-room-flow?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-5-pathway-1%3Fhl%3Dpt-br%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-intro-room-flow#2)

Como acontece com qualquer outra biblioteca, primeiro você precisará adicionar as dependências necessárias para usar o Room no app Bus Schedule. Para isso, será necessário fazer duas pequenas mudanças, uma em cada arquivo do Gradle.

1. No arquivo build.gradle do projeto, defina a room\_version no bloco ext.

ext {  
   kotlin\_version = "1.3.72"  
   nav\_version = "2.3.1"  
   room\_version = '2.3.0'  
}

1. No arquivo build.gradle do app, adicione as seguintes dependências ao final da lista de dependências.

implementation "androidx.room:room-runtime:$room\_version"  
kapt "androidx.room:room-compiler:$room\_version"  
  
// optional - Kotlin Extensions and Coroutines support for Room  
implementation "androidx.room:room-ktx:$room\_version"

1. Sincronize as mudanças e crie o projeto para verificar se as dependências foram adicionadas corretamente.

Nas próximas páginas, você verá uma apresentação dos componentes necessários para integrar o Room em um app: os modelos, o DAO, os modelos de visualização e a classe de banco de dados.

[**Back**](https://developer.android.com/codelabs/basic-android-kotlin-training-intro-room-flow?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-5-pathway-1%3Fhl%3Dpt-br%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-intro-room-flow)

[4. Criar entidade](https://developer.android.com/codelabs/basic-android-kotlin-training-intro-room-flow?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-5-pathway-1%3Fhl%3Dpt-br%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-intro-room-flow#3)

Ao aprender sobre bancos de dados relacionais no codelab anterior, você viu como os dados eram organizados em tabelas formadas por várias colunas, cada uma representando uma propriedade específica de um tipo de dado específico. Da mesma forma como as classes do Kotlin oferecem um modelo para cada objeto, uma tabela em um banco de dados fornece um modelo para cada item ou linha da tabela. Assim, uma classe do Kotlin pode ser usada para representar cada tabela do banco de dados.

Ao trabalhar com o Room, cada tabela é representada por uma classe. Em uma biblioteca de ORM (mapeamento relacional de objeto, na sigla em ingês) como o Room, essas classes costumam ser chamadas de *classes de modelo* ou *entidades*.

O banco de dados do app Bus Schedule é formado por uma única tabela, com informações básicas sobre os horários de chegada dos ônibus.

* id: um número inteiro que fornece um identificador exclusivo que serve como chave primária
* stop\_name: uma string
* arrival\_time: um número inteiro

Os tipos SQL usados no banco de dados são INTEGER para Int e TEXT para String. No entanto, ao trabalhar com o Room, você só precisa se preocupar com os tipos do Kotlin ao definir as classes de modelo. O mapeamento dos tipos de dados na classe de modelo para aqueles que serão usados no banco de dados é feito automaticamente.

Para criar uma entidade para a tabela de horários, crie um novo arquivo chamado **Schedule.kt** no pacote **database.schedule** e defina uma classe de dados chamada Schedule.

data class Schedule(  
)

Conforme abordado na lição sobre os princípios básicos de SQL, as tabelas de dados precisam ter uma chave primária para identificar cada linha de forma exclusiva. A primeira propriedade a ser adicionada à classe Schedule é um número inteiro para representar um ID exclusivo. Adicione uma nova propriedade e marque-a com a anotação @PrimaryKey. Isso instrui o Room a tratar essa propriedade como a chave primária quando novas linhas forem inseridas.

@PrimaryKey val id: Int

Adicione uma coluna para o nome do ponto de ônibus. A coluna precisa ser do tipo String. Para acrescentar novas colunas, será necessário adicionar uma anotação @ColumnInfo para especificar um nome para a coluna. Normalmente, os nomes das colunas SQL têm palavras separadas por um sublinhado, diferentemente do formato lowerCamelCase usado pelas propriedades do Kotlin. O valor dessa coluna não pode ser nulo. Por isso, marque-o com a anotação @NonNull.

@NonNull @ColumnInfo(name = "stop\_name") val stopName: String,

**Observação**: no SQL, as colunas podem ter valores nulos por padrão. Para que não sejam nulos, é necessário marcá-las de forma explícita. Essa característica é o oposto do funcionamento do Kotlin, em que, por padrão, os valores não podem ser nulos.

Os horários de chegada são representados no banco de dados usando números inteiros. Trata-se de um [carimbo de data/hora Unix](https://www.unixtimestamp.com/) (link em inglês), que pode ser convertido em uma data utilizável. Embora versões diferentes do SQL ofereçam formas distintas de converter datas, para os objetivos deste codelab, somente as funções de formato de data do Kotlin serão usadas. Adicione a seguinte coluna @NonNull à classe de modelo.

@NonNull @ColumnInfo(name = "arrival\_time") val arrivalTime: Int

Por fim, para que o Room reconheça essa classe como algo que pode ser usado para definir tabelas do banco de dados, é necessário adicionar uma anotação à classe. Adicione @Entity a uma linha separada, antes do nome da classe. Agora, a classe da entidade "schedule" ficará assim:

@Entity  
data class Schedule(  
   @PrimaryKey val id: Int,  
   @NonNull @ColumnInfo(name = "stop\_name") val stopName: String,  
   @NonNull @ColumnInfo(name = "arrival\_time") val arrivalTime: Int  
)

[**Back**](https://developer.android.com/codelabs/basic-android-kotlin-training-intro-room-flow?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-5-pathway-1%3Fhl%3Dpt-br%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-intro-room-flow)

[5. Definir o DAO](https://developer.android.com/codelabs/basic-android-kotlin-training-intro-room-flow?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-5-pathway-1%3Fhl%3Dpt-br%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-intro-room-flow#4)

A próxima classe que será adicionada para integrar o Room é o DAO. DAO significa objeto de acesso a dados, na sigla em inglês, e é uma classe do Kotlin que fornece acesso aos dados. Mais especificamente, o DAO é o local em que as funções de leitura e processamento de dados são incluídas. Chamar uma função no DAO equivale a executar um comando SQL no banco de dados. Funções DAO, como as que serão definidas neste app, geralmente especificam um comando SQL para que seja possível definir exatamente o que a função fará. O conhecimento sobre SQL adquirido no codelab anterior será útil ao definir o DAO.

1. Adicione uma classe DAO à entidade "Schedule". No pacote **database.schedule**, crie um novo arquivo com o nome ScheduleDao.kt e defina uma interface com o nome ScheduleDao. Assim como na classe Schedule, é necessário adicionar uma anotação, desta vez @Dao, para que a interface passe a ser utilizável com o Room.

@Dao  
interface ScheduleDao {  
}

**Observação**: segundo as convenções de nomenclatura para código do Kotlin, apenas a primeira letra dos acrônimos é grafada como maiúscula. Como DAO é um acrônimo, o nome fica **ScheduleDao**, e não **ScheduleDAO**.

1. O app tem duas telas, e cada uma delas precisará de uma consulta diferente. A primeira tela exibe todas as paradas de ônibus em ordem crescente de horário de chegada. Nesse caso de uso, a consulta precisa apenas abranger todas as colunas e incluir uma cláusula ORDER BY adequada. A consulta é especificada como uma string transmitida para uma anotação @Query. Defina uma função getAll() que retorna uma lista de objetos Schedule, incluindo a anotação @Query, conforme mostrado.

@Query("SELECT \* FROM schedule ORDER BY arrival\_time ASC")  
fun getAll(): List<Schedule>

1. Na segunda consulta, você também selecionará todas as colunas da tabela de horários. No entanto, somente os resultados que correspondam ao nome da parada selecionada serão retornados. Para isso, é necessário adicionar uma cláusula WHERE. É possível referenciar valores do Kotlin na consulta inserindo dois pontos (:) antes dela. Por exemplo, :stopName para o parâmetro da função. Como antes, os resultados são organizados em ordem crescente por horário de chegada. Defina uma função getByStopName() que use um parâmetro String chamado stopName e retorne uma List de objetos Schedule com uma anotação @Query, conforme mostrado.

@Query("SELECT \* FROM schedule WHERE stop\_name = :stopName ORDER BY arrival\_time ASC")  
fun getByStopName(stopName: String): List<Schedule>

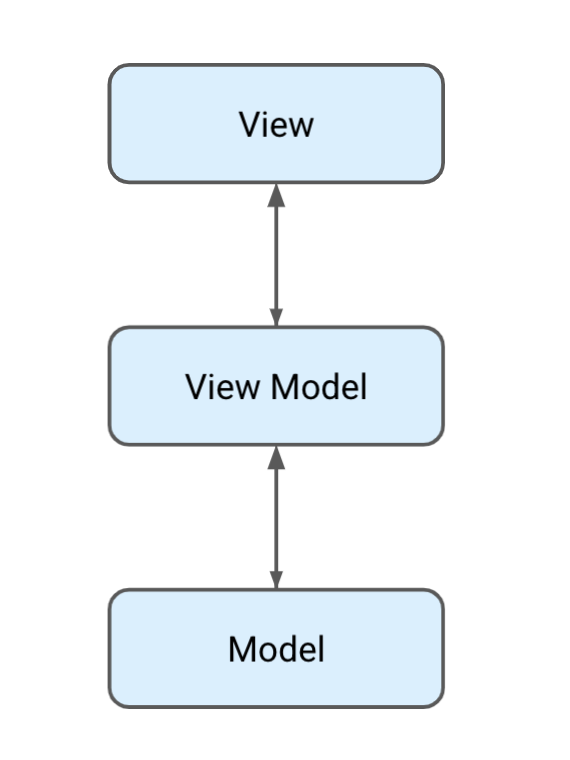
[**Back**](https://developer.android.com/codelabs/basic-android-kotlin-training-intro-room-flow?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-5-pathway-1%3Fhl%3Dpt-br%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-intro-room-flow)

[6. Definir o ViewModel](https://developer.android.com/codelabs/basic-android-kotlin-training-intro-room-flow?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-5-pathway-1%3Fhl%3Dpt-br%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-intro-room-flow#5)

Agora que o DAO foi configurado, tecnicamente você já tem tudo que é necessário para começar a acessar o banco de dados nos fragmentos. No entanto, embora isso funcione na teoria, geralmente não é considerado uma prática recomendada. Isso porque, em apps mais complexos, é provável que existam várias telas que acessam apenas uma parte específica dos dados. Embora o ScheduleDao seja relativamente simples, é fácil imaginar como isso poderia fugir do controle ao trabalhar com duas ou mais telas diferentes. Por exemplo, um DAO pode ser parecido com o exemplo a seguir.

@Dao  
interface ScheduleDao {  
  
    @Query(...)  
    getForScreenOne() ...  
  
    @Query(...)  
    getForScreenTwo() ...  
  
    @Query(...)  
    getForScreenThree()  
  
}

Embora o código de Screen 1 possa acessar getForScreenOne(), não existe um bom motivo para ele acessar os outros métodos. Em vez disso, é recomendável separar a parte do DAO que é exposta à visualização em uma classe separada, chamada de *modelo de visualização*. Esse é um padrão de arquitetura comum em apps para dispositivos móveis. O uso de um modelo de visualização ajuda a definir uma separação clara entre o código da IU do app e o modelo de dados dele. Isso também ajuda a testar cada parte do código de forma independente. Esse assunto será mais explorado ao longo de sua jornada de desenvolvimento para Android.



Usando um modelo de visualização, é possível aproveitar a classe ViewModel. A ViewModel é usada para armazenar dados relacionados à IU de um app. Ela também reconhece o ciclo de vida, o que significa que responde a eventos de ciclo de vida da mesma forma que uma atividade ou um fragmento. Se eventos de ciclo de vida, como a rotação da tela, fizerem com que uma atividade ou um fragmento seja destruído e recriado, o ViewModel associado não precisará ser recriado. Como não é possível fazer isso acessando diretamente uma classe DAO, é recomendável usar a subclasse ViewModel para fazer a separação entre a responsabilidade de carregar dados e a atividade ou o fragmento.

**Observação**: o Bus Schedule é um app relativamente simples e inclui apenas duas telas de conteúdo majoritariamente idêntico. Para fins de estudo, criaremos uma única classe de modelo de visualização que pode ser usada por ambas as telas. No entanto, em um app maior, é recomendável usar um modelo de visualização separado para cada fragmento.

1. Para criar uma classe de modelo de visualização, crie um novo arquivo chamado **BusScheduleViewModel.kt** em um novo pacote chamado **viewmodels**. Defina uma classe para o modelo de visualização. Ela precisa usar um único parâmetro do tipo ScheduleDao.

class BusScheduleViewModel(private val scheduleDao: ScheduleDao): ViewModel() {

1. Como esse modelo de visualização será usado por ambas as telas, será necessário adicionar um método para buscar todos os horários, bem como os horários filtrados por ponto de ônibus. Para fazer isso, chame os métodos correspondentes de ScheduleDao.

fun fullSchedule(): List<Schedule> = scheduleDao.getAll()  
  
fun scheduleForStopName(name: String): List<Schedule> = scheduleDao.getByStopName(name)

Embora o modelo de visualização já tenha sido definido, não é possível instanciar somente um BusScheduleViewModel diretamente e esperar que tudo funcione. Como o objetivo do BusScheduleViewModel da classe ViewModel é ter reconhecimento de ciclo de vida, ele precisa ser instanciado por um objeto que possa responder a eventos de ciclo de vida. Se você instanciá-lo diretamente em um dos fragmentos, o objeto de fragmento terá que processar todo o gerenciamento de memória, o que está fora do escopo do código do app. Em vez disso, você pode criar uma classe, chamada factory, que instanciará objetos de modelo de visualização para você.

1. Para criar uma factory, abaixo da classe de modelo de visualização, crie uma nova classe BusScheduleViewModelFactory, herdada de ViewModelProvider.Factory.

class BusScheduleViewModelFactory(  
   private val scheduleDao: ScheduleDao  
) : ViewModelProvider.Factory {  
}

1. Você precisará usar um código boilerplate para instanciar corretamente um modelo de visualização. Em vez de inicializar a classe diretamente, modifique o método create() que retorna uma BusScheduleViewModelFactory e faz algumas verificações de erros. Implemente create() dentro da classe BusScheduleViewModelFactory, conforme mostrado a seguir:

override fun <T : ViewModel> create(modelClass: Class<T>): T {  
       if (modelClass.isAssignableFrom(BusScheduleViewModel::class.java)) {  
           @Suppress("UNCHECKED\_CAST")  
           return BusScheduleViewModel(scheduleDao) as T  
       }  
       throw IllegalArgumentException("Unknown ViewModel class")  
   }

Agora é possível instanciar um objeto BusScheduleViewModelFactory com BusScheduleViewModelFactory.create(), para que o modelo de visualização possa reconhecer o ciclo de vida sem que isso precise ser processado diretamente pelo fragmento.

[**Back**](https://developer.android.com/codelabs/basic-android-kotlin-training-intro-room-flow?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-5-pathway-1%3Fhl%3Dpt-br%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-intro-room-flow)

[7. Criar uma classe de banco de dados e pré-preencher o banco de dados](https://developer.android.com/codelabs/basic-android-kotlin-training-intro-room-flow?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-5-pathway-1%3Fhl%3Dpt-br%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-intro-room-flow#6)

Agora que você já definiu os modelos, o DAO e um modelo de visualização para que os fragmentos acessem o DAO, ainda é necessário informar ao Room o que fazer com todas essas classes. É aí que entra a classe AppDatabase. Um app Android que usa o Room, como é o caso do seu, tem uma subclasse da RoomDatabase e algumas responsabilidades essenciais. No app, o AppDatabase precisa:

1. especificar quais entidades estão definidas no banco de dados;
2. fornecer acesso a uma única instância de cada classe DAO;
3. executar qualquer outra configuração, como pré-preencher o banco de dados.

Você pode estar se perguntando por que o Room não consegue encontrar todas as entidades e objetos DAO. Isso ocorre porque é provável que o app tenha vários bancos de dados ou inúmeros cenários. Dessa forma, a biblioteca não consegue presumir a intenção do desenvolvedor. A classe AppDatabase oferece controle total sobre os modelos, classes DAO e qualquer configuração do banco de dados que você queira executar.

1. Para adicionar uma classe AppDatabase, crie um novo arquivo chamado AppDatabase.kt no pacote **database** e defina uma nova classe abstrata AppDatabase, herdada de RoomDatabase.

abstract class AppDatabase: RoomDatabase() {  
}

1. A classe do banco de dados possibilita que outras acessem facilmente as classes DAO. Adicione uma função abstrata que retorne um ScheduleDao.

abstract fun scheduleDao(): ScheduleDao

1. Ao usar uma classe AppDatabase, é importante garantir que exista apenas uma instância do banco de dados, para evitar disputas ou outros possíveis problemas. A instância fica armazenada no objeto complementar. Você também precisará de um método que retorne a instância existente ou crie o banco de dados pela primeira vez. Isso é definido no objeto complementar. Adicione o companion object a seguir logo abaixo da função scheduleDao().

companion object {  
}

Em companion object, adicione uma propriedade com o nome INSTANCE do tipo AppDatabase. Inicialmente, esse valor é definido como null, portanto, o tipo é marcado com ?. Também é marcado com uma anotação @Volatile. Embora os detalhes sobre quando usar uma propriedade volátil sejam um pouco avançados para esta lição, você precisará usá-la na instância AppDatabase para evitar possíveis bugs.

@Volatile  
private var INSTANCE: AppDatabase? = null

Abaixo da propriedade INSTANCE, defina uma função para retornar a instância AppDatabase.

fun getDatabase(context: Context): AppDatabase {  
    return INSTANCE ?: synchronized(this) {  
        val instance = Room.databaseBuilder(  
            context,  
            AppDatabase::class.java,  
            "app\_database")  
            .createFromAsset("database/bus\_schedule.db")  
            .build()  
        INSTANCE = instance  
  
        instance  
    }  
}

Na implementação de getDatabase(), use o operador Elvis para retornar a instância atual do banco de dados, se já existir, ou criar o banco de dados pela primeira vez, se necessário. Como os dados são preenchidos automaticamente neste app, Você também chama createFromAsset() para carregar os dados existentes. O arquivo bus\_schedule.db pode ser encontrado no pacote assets.database do projeto.

1. Assim como as classes de modelo e o DAO, a classe do banco de dados requer uma anotação com algumas informações específicas. Todos os tipos de entidade estão listados em uma matriz, e cada tipo pode ser acessado usando ClassName::class. O banco de dados também recebe um número de versão, que será definido como 1. Adicione a anotação @Database da seguinte maneira:

@Database(entities = arrayOf(Schedule::class), version = 1)

**Observação**: o número da versão aumenta todas as vezes que você faz uma mudança de esquema. O app verifica se essa versão corresponde à versão do banco de dados para determinar se, e como, uma migração será realizada.

Agora que a classe AppDatabase foi criada, há mais um passo para que ela passe a ser utilizável. Será necessário fornecer uma subclasse personalizada da classe Application e criar uma propriedade lazy que contenha o resultado de getDatabase().

1. Adicione um novo arquivo chamado BusScheduleApplication.kt e crie uma classe BusScheduleApplication, herdada de Application.

class BusScheduleApplication : Application() {  
}

1. Adicione uma propriedade de banco de dados do tipo AppDatabase. A propriedade precisa ser lenta e retornar o resultado da chamada de getDatabase() na classe AppDatabase.

class BusScheduleApplication : Application() {  
   val database: AppDatabase by lazy { AppDatabase.getDatabase(this) }

1. Por fim, para garantir que a classe BusScheduleApplication seja usada, em vez da classe base padrão Application, será necessário fazer uma pequena mudança no manifesto. Em AndroidMainifest.xml, defina a propriedade android:name como com.example.busschedule.BusScheduleApplication.

<application  
    android:name="com.example.busschedule.BusScheduleApplication"  
    ...

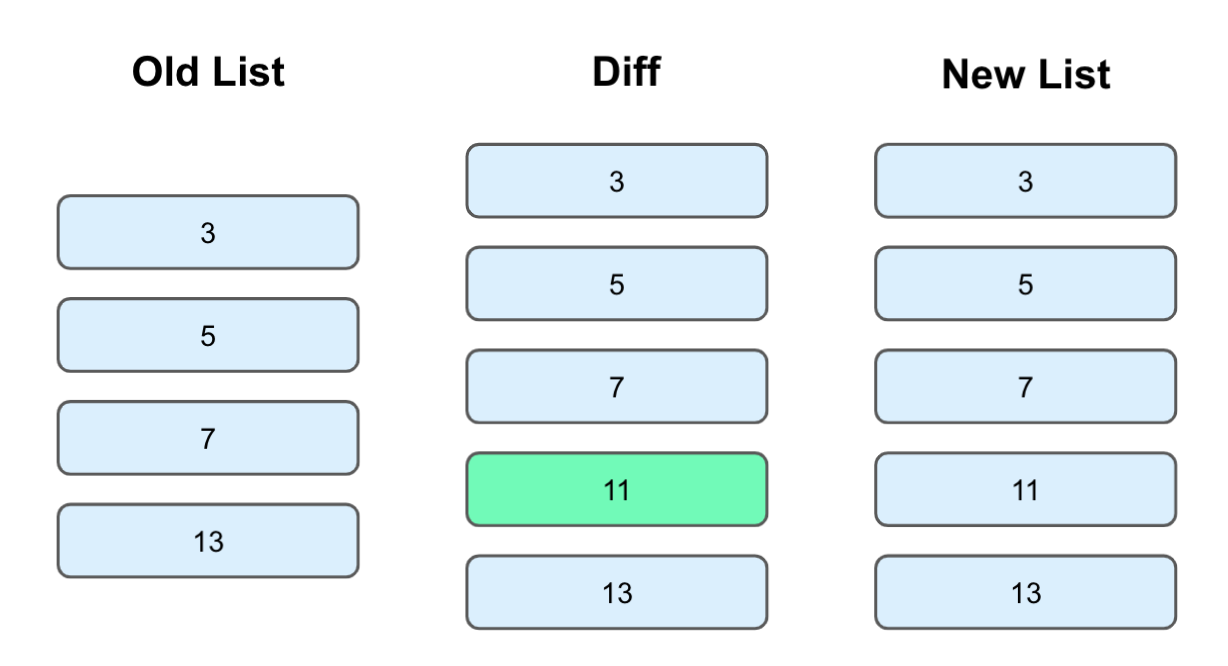
Agora o modelo do app está configurado. Você já pode começar a usar os dados do Room na IU. Nas próximas páginas, você criará um ListAdapter para que a RecyclerView do app exiba os dados dos horários de ônibus e responda a mudanças de dados de forma dinâmica.

[**Back**](https://developer.android.com/codelabs/basic-android-kotlin-training-intro-room-flow?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-5-pathway-1%3Fhl%3Dpt-br%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-intro-room-flow)

[8. Criar o ListAdapter](https://developer.android.com/codelabs/basic-android-kotlin-training-intro-room-flow?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-5-pathway-1%3Fhl%3Dpt-br%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-intro-room-flow#7)

É hora de colocar todo esse trabalho em uso e conectar o modelo à visualização. Anteriormente, ao usar uma RecyclerView, você utilizaria um RecyclerView.RecyclerView para exibir uma lista estática de dados. Embora isso certamente funcione para um app como o Bus Schedule, é comum processar mudanças em tempo real ao trabalhar com um banco de dados. Mesmo que o conteúdo de apenas um item mude, toda a visualização de reciclagem será atualizada. Isso não funcionará para a maioria dos apps que usam a persistência.

Uma alternativa que pode ser usada para uma lista que muda de forma dinâmica é chamada de ListAdapter. O ListAdapter usa [AsyncListDiffer](https://developer.android.com/reference/androidx/recyclerview/widget/AsyncListDiffer" \t "_blank) para determinar as diferenças entre uma lista de dados antiga e uma nova. Em seguida, a visualização de reciclagem será atualizada somente com base nas diferenças entre as duas listas. Como resultado, a visualização da reciclagem será mais eficiente ao processar dados atualizados com frequência, como ocorre frequentemente em aplicativos com banco de dados.



Como a IU é idêntica nas duas telas, você só precisará criar um ListAdapter que possa ser usado por ambas as telas.

1. Crie um novo arquivo BusStopAdapter.kt e uma classe BusStopAdapter, conforme mostrado abaixo. A classe estende um ListAdapter genérico que usa uma lista de objetos Schedule e uma classe BusStopViewHolder para a IU. Para o BusStopViewHolder, você também precisa transmitir um tipo DiffCallback, que será definido em breve. A própria classe BusStopAdapter também usa um parâmetro, onItemClicked(). Essa função será usada para processar a navegação quando um item for selecionado na primeira tela. Porém, na segunda tela, apenas uma função vazia será transmitida.

class BusStopAdapter(private val onItemClicked: (Schedule) -> Unit) : ListAdapter<Schedule, BusStopAdapter.BusStopViewHolder>(DiffCallback) {  
}

1. De forma semelhante a um adaptador de visualização de reciclagem, você precisará de um fixador de visualização para acessar as visualizações criadas a partir do arquivo de layout no código. O layout das células já foi criado. Basta criar uma classe BusStopViewHolder, conforme mostrado abaixo, e implementar a função bind() para definir o texto de stopNameTextView como o nome do ponto de ônibus e o texto da arrivalTimeTextView como a data formatada.

class BusStopViewHolder(private var binding: BusStopItemBinding): RecyclerView.ViewHolder(binding.root) {  
    @SuppressLint("SimpleDateFormat")  
    fun bind(schedule: Schedule) {  
        binding.stopNameTextView.text = schedule.stopName  
        binding.arrivalTimeTextView.text = SimpleDateFormat(  
            "h:mm a").format(Date(schedule.arrivalTime.toLong() \* 1000)  
        )  
    }  
}

1. Modifique e implemente o onCreateViewHolder(), infle o layout e instrua o onClickListener() a chamar onItemClicked() para o item na posição atual.

override fun onCreateViewHolder(parent: ViewGroup, viewType: Int): BusStopViewHolder {  
   val viewHolder = BusStopViewHolder(  
       BusStopItemBinding.inflate(  
           LayoutInflater.from( parent.context),  
           parent,  
           false  
       )  
   )  
   viewHolder.itemView.setOnClickListener {  
       val position = viewHolder.adapterPosition  
       onItemClicked(getItem(position))  
   }  
   return viewHolder  
}

1. Modifique e implemente o onBindViewHolder() e vincule a visualização na posição especificada.

override fun onBindViewHolder(holder: BusStopViewHolder, position: Int) {  
   holder.bind(getItem(position))  
}

1. Você se lembra da classe DiffCallback que foi especificada para o ListAdapter? Ela é apenas um objeto que ajuda o ListAdapter a determinar quais itens nas listas novas e antigas são diferentes ao atualizar a lista. Há dois métodos: areItemsTheSame() verifica se o objeto (ou a linha no banco de dados no seu caso) é igual apenas verificando o ID. O areContentsTheSame() verifica se todas as propriedades, não apenas o ID, são iguais. Esses métodos permitem que o ListAdapter determine quais itens foram inseridos, atualizados e excluídos para que a IU possa ser atualizada de acordo.

Adicione um objeto complementar e implemente DiffCallback, conforme mostrado a seguir.

companion object {  
   private val DiffCallback = object : DiffUtil.ItemCallback<Schedule>() {  
       override fun areItemsTheSame(oldItem: Schedule, newItem: Schedule): Boolean {  
           return oldItem.id == newItem.id  
       }  
  
       override fun areContentsTheSame(oldItem: Schedule, newItem: Schedule): Boolean {  
           return oldItem == newItem  
       }  
   }  
}

A configuração do adaptador está pronta. Ele será usado nas duas telas do app.

1. Primeiro, em FullScheduleFragment.kt, é necessário ter uma referência ao modelo de visualização.

private val viewModel: BusScheduleViewModel by activityViewModels {  
   BusScheduleViewModelFactory(  
       (activity?.application as BusScheduleApplication).database.scheduleDao()  
   )  
}

1. Em seguida, em onViewCreated(), adicione o código a seguir para configurar a visualização de reciclagem e atribuir o gerenciador de layout.

recyclerView = binding.recyclerView  
recyclerView.layoutManager = LinearLayoutManager(requireContext())

1. Depois, atribua a propriedade do adaptador. A ação transmitida usará o stopName para navegar pela próxima tela selecionada, de modo que a lista de pontos de ônibus possa ser filtrada.

val busStopAdapter = BusStopAdapter({  
   val action = FullScheduleFragmentDirections.actionFullScheduleFragmentToStopScheduleFragment(  
       stopName = it.stopName  
   )  
   view.findNavController().navigate(action)  
})  
recyclerView.adapter = busStopAdapter

1. Por fim, para atualizar uma visualização em lista, chame submitList(), transmitindo a lista de pontos de ônibus do modelo de visualização.

busStopAdapter.submitList(viewModel.fullSchedule())

1. Faça o mesmo no StopScheduleFragment. Primeiro, encontre uma referência ao modelo de visualização.

private val viewModel: BusScheduleViewModel by activityViewModels {  
   BusScheduleViewModelFactory(  
       (activity?.application as BusScheduleApplication).database.scheduleDao()  
   )  
}

1. Em seguida, configure a visualização de reciclagem em onViewCreated(). Desta vez, basta transmitir um bloco vazio (função) com {}. O objetivo é que nada aconteça ao tocar nas linhas dessa tela.

recyclerView = binding.recyclerView  
recyclerView.layoutManager = LinearLayoutManager(requireContext())  
val busStopAdapter = BusStopAdapter({})  
recyclerView.adapter = busStopAdapter  
busStopAdapter.submitList(viewModel.scheduleForStopName())

1. Agora que você configurou o adaptador, a integração do Room ao app Bus Schedule está pronta. Execute o app, e uma lista de horários de chegada será exibida. Toque em uma linha para acessar a tela de detalhes.

|  |  |
| --- | --- |
|  |  |

[**Back**](https://developer.android.com/codelabs/basic-android-kotlin-training-intro-room-flow?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-5-pathway-1%3Fhl%3Dpt-br%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-intro-room-flow)

[9. Responder a mudanças de dados usando o fluxo](https://developer.android.com/codelabs/basic-android-kotlin-training-intro-room-flow?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-5-pathway-1%3Fhl%3Dpt-br%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-intro-room-flow#8)

Embora a visualização em lista esteja configurada para processar as mudanças de dados de forma eficiente sempre que submitList() for chamado, o app ainda não processa atualizações dinâmicas. Para ver como isso ocorre, abra o Database Inspector e execute a consulta a seguir para inserir um novo item na tabela de horários.

INSERT INTO schedule  
VALUES (null, 'Winding Way', 1617202500)

Você perceberá que nada acontece no emulador. O usuário presumirá que os dados não mudaram. Será necessário executar o app novamente para ver as mudanças.

O problema é que List<Schedule> é retornado de cada uma das funções DAO apenas uma vez. Mesmo que os dados subjacentes sejam atualizados, submitList() não será chamado para atualizar a IU. Assim, da perspectiva do usuário, não haverá mudanças.

Para corrigir isso, use um recurso do Kotlin chamado *fluxo assíncrono*, frequentemente conhecido apenas como *fluxo*. Ele permitirá que o DAO emita dados de forma contínua com base no banco de dados. Se um item for inserido, atualizado ou excluído, o resultado será enviado de volta ao fragmento. Usando uma função chamada collect(),, é possível chamar submitList() com o novo valor emitido pelo fluxo, para que o ListAdapter possa atualizar a IU com base nos novos dados.

1. Para usar o fluxo no Bus Schedule, abra ScheduleDao.kt. Para converter as funções DAO para retornar um Flow, basta mudar o tipo de retorno da função getAll() para Flow<List<Schedule>>.

fun getAll(): Flow<List<Schedule>>

1. Da mesma forma, atualize o valor de retorno da função getByStopName().

fun getByStopName(stopName: String): Flow<List<Schedule>>

1. As funções do modelo de visualização que acessam o DAO também precisam ser atualizadas. Atualize os valores de retorno de Flow<List<Schedule>> para fullSchedule() e scheduleForStopName().

class BusScheduleViewModel(private val scheduleDao: ScheduleDao): ViewModel() {  
  
   fun fullSchedule(): Flow<List<Schedule>> = scheduleDao.getAll()  
  
   fun scheduleForStopName(name: String): Flow<List<Schedule>> = scheduleDao.getByStopName(name)  
}

1. Por fim, em FullScheduleFragment.kt, o busStopAdapter precisa ser atualizado quando collect() é chamado nos resultados da consulta. Como o método fullSchedule() é uma função de suspensão, ele precisa ser chamado em uma corrotina. Substitua a linha:

busStopAdapter.submitList(viewModel.fullSchedule())

Pelo seguinte código que usa o fluxo retornado de fullSchedule():

lifecycle.coroutineScope.launch {  
   viewModel.fullSchedule().collect() {  
       busStopAdapter.submitList(it)  
   }  
}

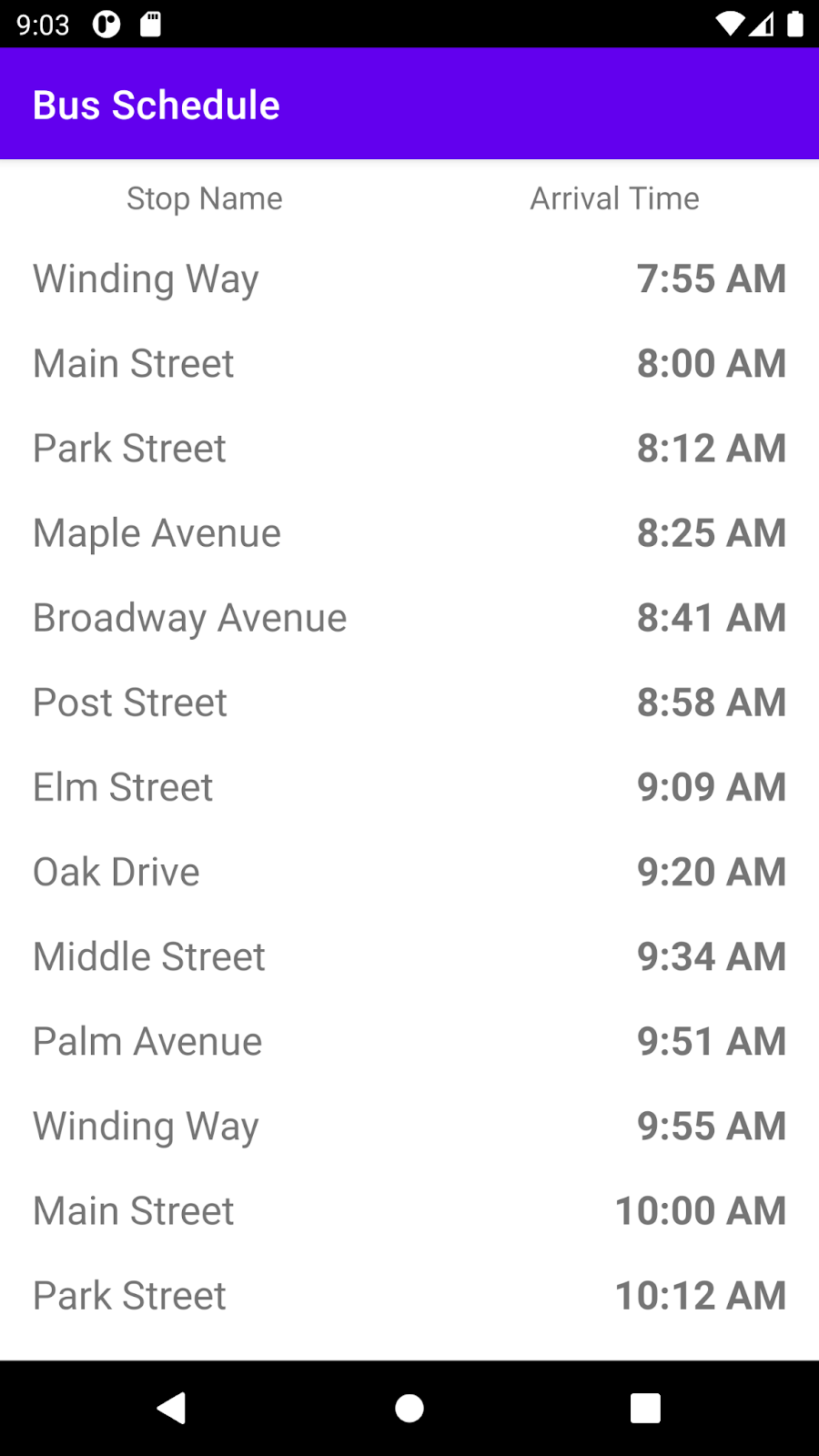
1. Faça o mesmo em StopScheduleFragment, mas substitua a chamada por scheduleForStopName() usando o código a seguir.

lifecycle.coroutineScope.launch {  
   viewModel.scheduleForStopName(stopName).collect() {  
       busStopAdapter.submitList(it)  
   }  
}

1. Depois de fazer as mudanças acima, execute novamente o app para verificar se as mudanças nos dados agora são processadas em tempo real. Quando o app estiver em execução, retorne ao Database Inspector e envie a seguinte consulta para inserir um novo horário de chegada antes das 8h.

INSERT INTO schedule  
VALUES (null, 'Winding Way', 1617202500)

O novo item aparecerá no topo da lista.



O app Bus Schedule está pronto. Ótimo trabalho até aqui! Agora você tem uma base sólida para trabalhar com o Room. No próximo módulo, você estudará mais sobre o Room com um novo app de exemplo e aprenderá a salvar dados criados pelo usuário em um dispositivo.

[**Back**](https://developer.android.com/codelabs/basic-android-kotlin-training-intro-room-flow?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-5-pathway-1%3Fhl%3Dpt-br%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-intro-room-flow)

[10. Código da solução](https://developer.android.com/codelabs/basic-android-kotlin-training-intro-room-flow?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-5-pathway-1%3Fhl%3Dpt-br%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-intro-room-flow#9)

O código da solução para este codelab está no projeto e no módulo mostrados abaixo.

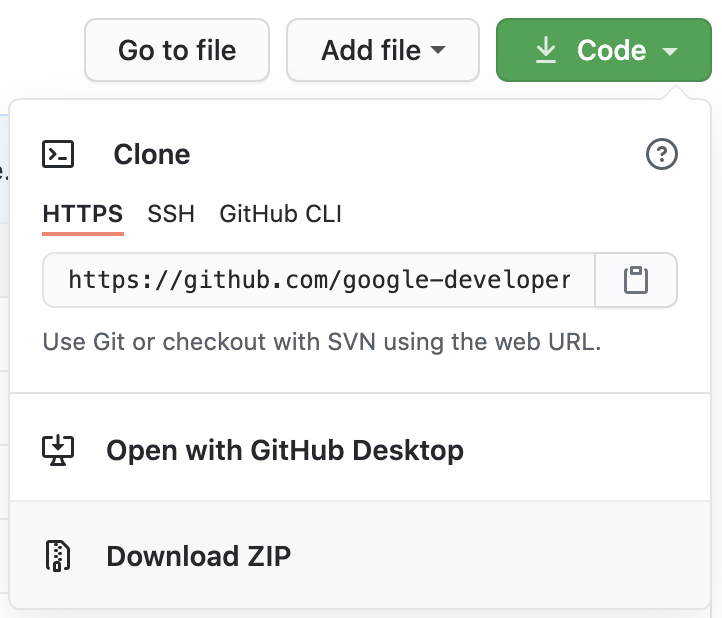
**URL do código da solução:**[https://github.com/google-developer-training/android-basic-kotlin-bus-schedule-app](https://github.com/google-developer-training/android-basics-kotlin-bus-schedule-app) (link em inglês)

**Nome da ramificação: main**

Para encontrar o código deste codelab e abri-lo no Android Studio, faça o seguinte.

Buscar o código

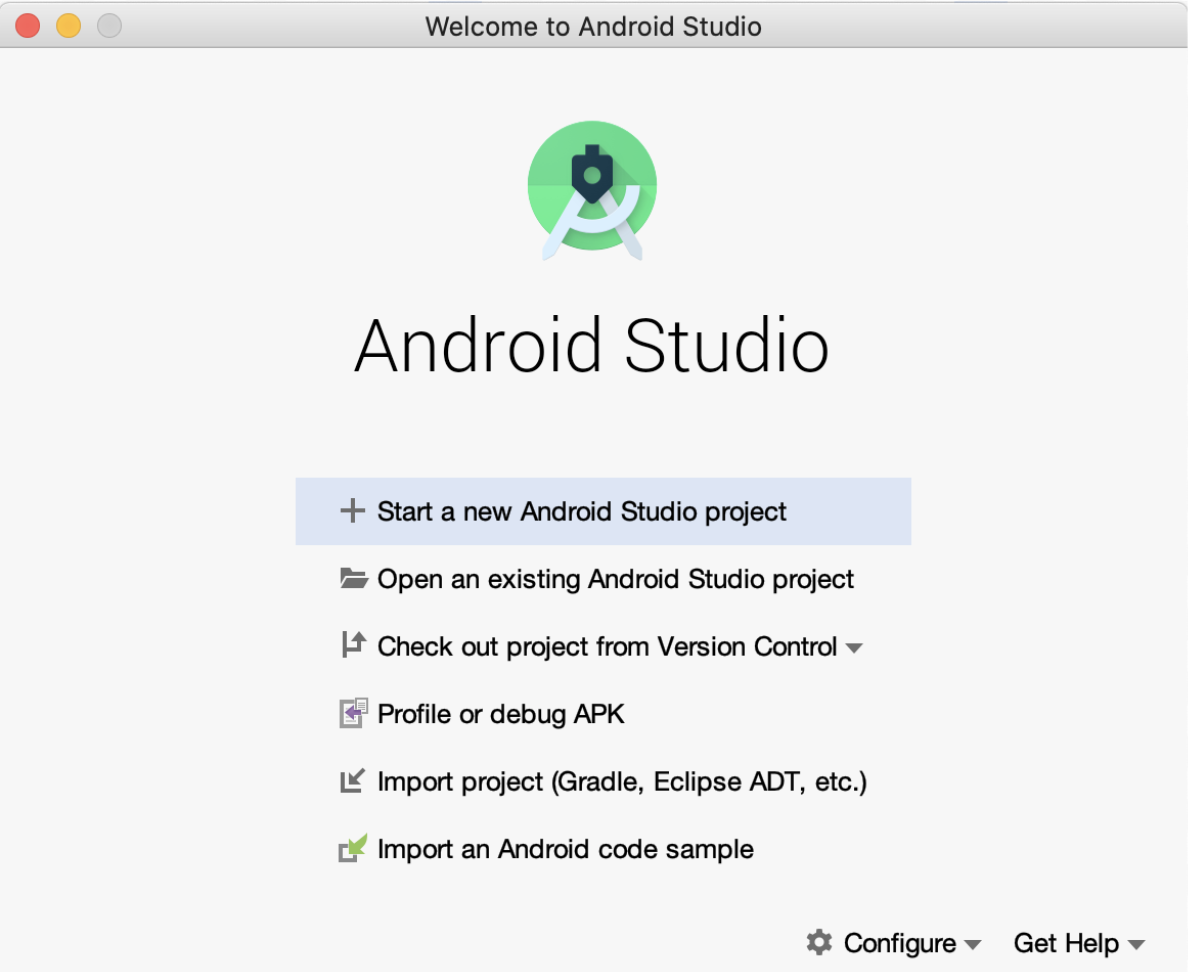
1. Clique no URL fornecido. Isso abrirá a página do GitHub referente ao projeto em um navegador.
2. Na página do GitHub do projeto, clique no botão **Code**, que exibirá uma caixa de diálogo.



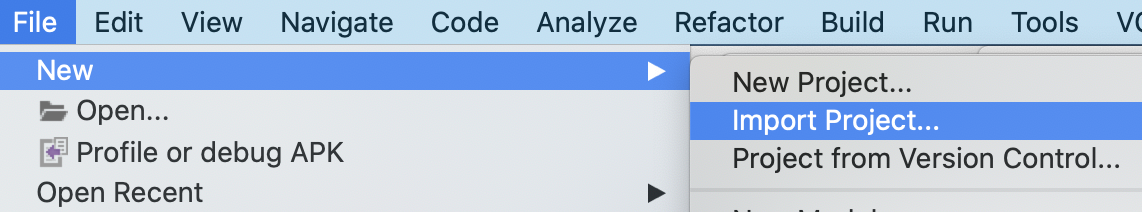
1. Na caixa de diálogo, clique no botão **Download ZIP** para salvar o projeto no seu computador. Aguarde a conclusão do download.
2. Localize o arquivo no computador, que provavelmente está na pasta **Downloads**.
3. Clique duas vezes no arquivo ZIP para descompactá-lo. Isso criará uma nova pasta com os arquivos do projeto.

Abrir o projeto no Android Studio

1. Inicie o Android Studio.
2. Na janela **Welcome to Android Studio**, clique em **Open an existing Android Studio project**.



Observação: caso o Android Studio já esteja aberto, selecione a opção **File** > **New** > **Import Project**.



1. Na caixa de diálogo **Import Project**, vá até a pasta do projeto descompactada, que provavelmente está na pasta **Downloads**.
2. Clique duas vezes nessa pasta do projeto.
3. Aguarde o Android Studio abrir o projeto.
4. Clique no botão **Run**  para criar e executar o app. Confira se ele é compilado da forma esperada.
5. Procure os arquivos do projeto na janela de ferramentas **Project** para ver como o app está configurado.

[**Back**](https://developer.android.com/codelabs/basic-android-kotlin-training-intro-room-flow?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-5-pathway-1%3Fhl%3Dpt-br%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-intro-room-flow)

[11. Parabéns](https://developer.android.com/codelabs/basic-android-kotlin-training-intro-room-flow?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-5-pathway-1%3Fhl%3Dpt-br%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-intro-room-flow#10)

Resumindo:

* As tabelas em um banco de dados SQL são representadas no Room por classes do Kotlin chamadas entidades.
* O DAO fornece métodos correspondentes aos comandos SQL, que interagem com o banco de dados.
* O ViewModel é um componente com reconhecimento de ciclo de vida usado para separar os dados do app da visualização.
* A classe AppDatabase informa o Room quais entidades usar, fornece acesso ao DAO e executa todas as configurações ao criar o banco de dados.
* O ListAdapter é um adaptador usado com a RecyclerView, ideal para processar listas atualizadas de forma dinâmica.
* Fluxo é um recurso do Kotlin para retornar um stream de dados e pode ser usado com o Room para garantir que a IU e o banco de dados estejam sincronizados.

Saiba mais

* [ViewModel](https://developer.android.com/topic/libraries/architecture/viewmodel)
* [ViewModelProvider.Factory](https://developer.android.com/reference/androidx/lifecycle/ViewModelProvider.Factory)
* [RoomDataBase](https://developer.android.com/reference/android/arch/persistence/room/RoomDatabase)
* [@Volatile annotation](https://kotlinlang.org/api/latest/jvm/stdlib/kotlin.jvm/-volatile/) (link em inglês)
* [ListAdapter](https://developer.android.com/reference/androidx/recyclerview/widget/ListAdapter)
* [AsyncListDiffer](https://developer.android.com/reference/androidx/recyclerview/widget/AsyncListDiffer)

[**Back**](https://developer.android.com/codelabs/basic-android-kotlin-training-intro-room-flow?continue=https%3A%2F%2Fdeveloper.android.com%2Fcourses%2Fpathways%2Fandroid-basics-kotlin-unit-5-pathway-1%3Fhl%3Dpt-br%23codelab-https%3A%2F%2Fdeveloper.android.com%2Fcodelabs%2Fbasic-android-kotlin-training-intro-room-flow)