ANDROID JETPACK

MAKE BETTER APPS

Speaker notes

+

TODAY'S SESSION

- Jetpack Overview
- Adding Jetpack to Your App
- Foundation
- Architecture Components
- More Architecture Components
- Behavior
- UI
- Questions

Speaker notes

+

Speaker notes

- Web/mobile dev
- Android lead at BCycle
- Former Android lead at Kohl's
- Alexa and Google Assistant developer

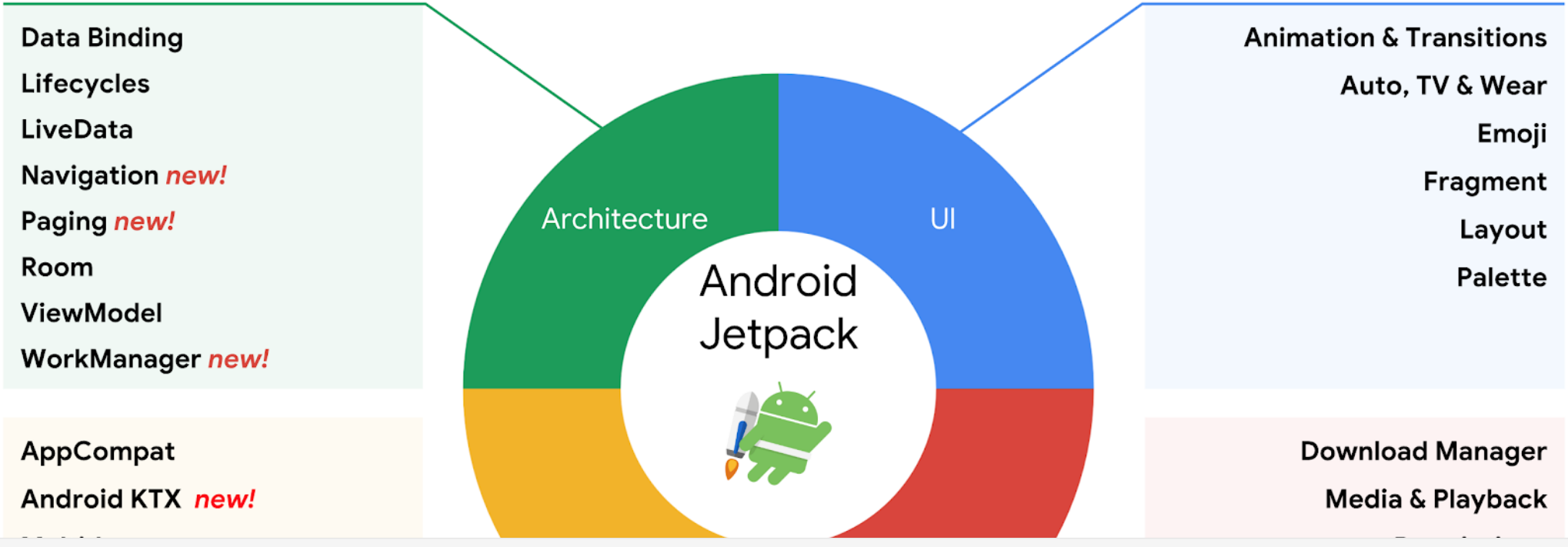
SKYLINE

Speaker notes

- Tech consultants in WI
- Microsoft partner
- Web knowledge as well
- Voice assistants

Speaker notes

- Lots to cover
- I move around
- I'm Italian, my hands move a lot



Speaker notes

- Components + Best Practices
  - Components to make Android dev easier
  - Help follow best practices
- Remove boilerplate code
- Simplify complex tasks
- Located in the AndroidX package
  - Pull in only what you need

Speaker notes

- A first-class language on Android
- Now the default when starting in Android Studio
- Easy to add to existing projects
  - Add a new Kotlin Activity or class/file
  - Android Studio automatically adds Kotlin support to Gradle files
- Android Studio has a built-in converter from Java to Kotlin

Why Kotlin?

Concise

Drastically reduce the amount of boilerplate code.

See example

Safe

Avoid entire classes of errors such as null pointer exceptions.

See example

Interoperable

Leverage existing libraries for the JVM, Android, and the browser.

See example

Tool-friendly

Choose any Java IDE or build from the command line.

See example

Speaker notes

- Way less code
- Null safety
- Extension methods
  - Android KTX
  - KotlinX

play.kotlinlang.org

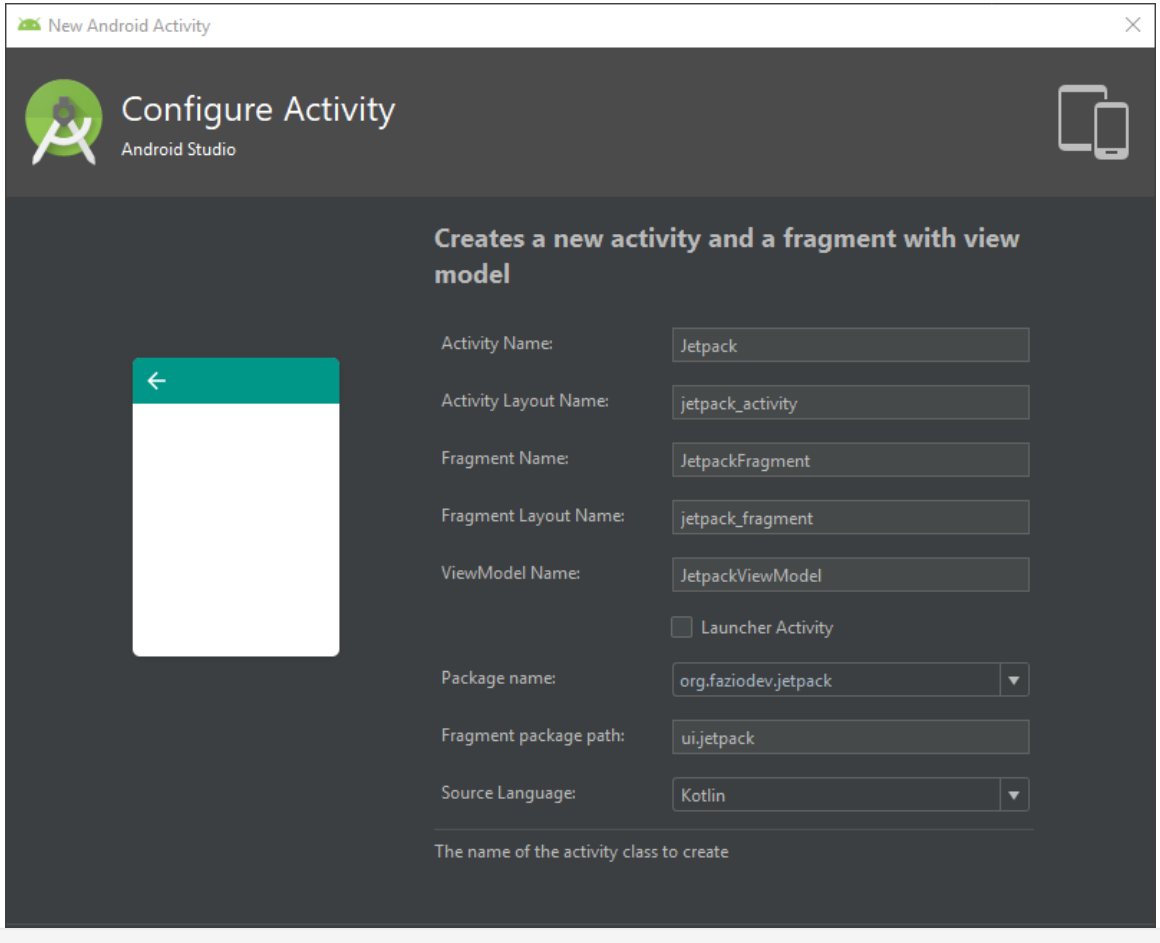
Kotlin Playground is an online sandbox to explore Kotlin programming language. Browse code samples directly in the browser

```
*
 You can edit, run, and share this code.
 play.kotlinlang.org
/


 List<Player>.topTen(
 filterPredicate: (Player) -> Boolean,
 sortPredicate: (Player) -> Int,
```

Speaker notes

+

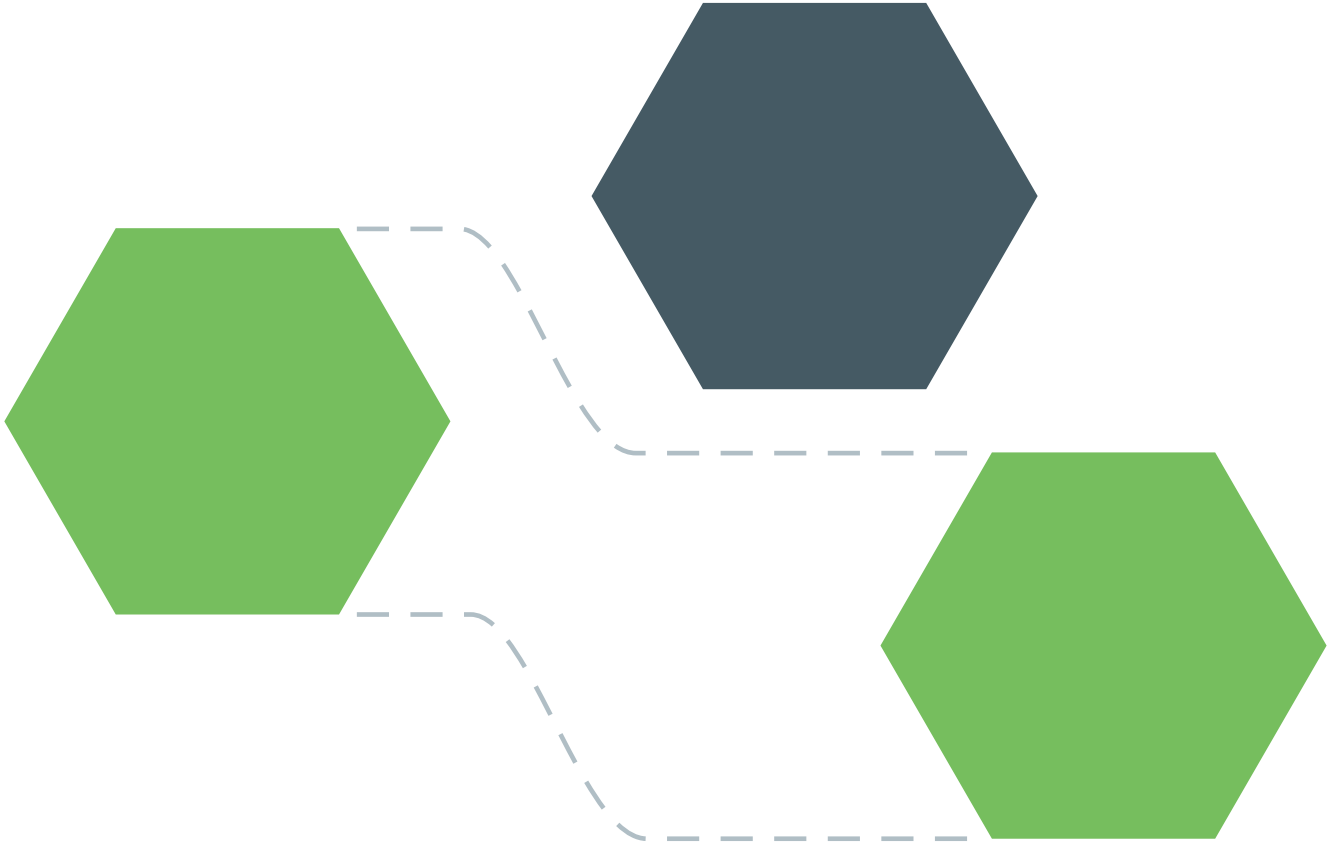
ADDING JETPACK



Speaker notes

- New > Activity > Fragment + ViewModel
  - Sets up a single-Activity architecture
- Update Gradle file to include Jetpack libraries

FOUNDATION



Speaker notes

- Foundation components provide cross-cutting functionality like backwards compatibility, testing and Kotlin language support.

APPCOMPAT

Speaker notes

- More than 10% of devices still running a four+ year old OS
  - Graph from end of October
- All moved to the AndroidX library
  - Existing versions still work
  - All new dev will be in AndroidX
  - Migration
    - Inside Android Studio: Refactor > Migrate
    - `android.useAndroidX=true`
    - `android.enableJetifier=true`
    - https://developer.android.com/jetpack/androidx/migrate
- Target API level 28 (Android 9.0)

ANDROID KTX

```kotlin
// Kotlin
view.viewTreeObserver.addOnPreDrawListener(
    object : ViewTreeObserver.OnPreDrawListener {
        override fun onPreDraw(): Boolean {
            viewTreeObserver.removeOnPreDrawListener(this)
            actionToBeTriggered()
            return true
        }
    }
)
```

Speaker notes

- Kotlin Extensions
- Add `google()` repository
- Add `'androidx.core:core-ktx:1.0.0'` dependency
- Extensions for:
  - Animation
  - SQLite interaction (but use Room instead)
  - Fragment transactions (but use Navigation instead)
  - Many, many more

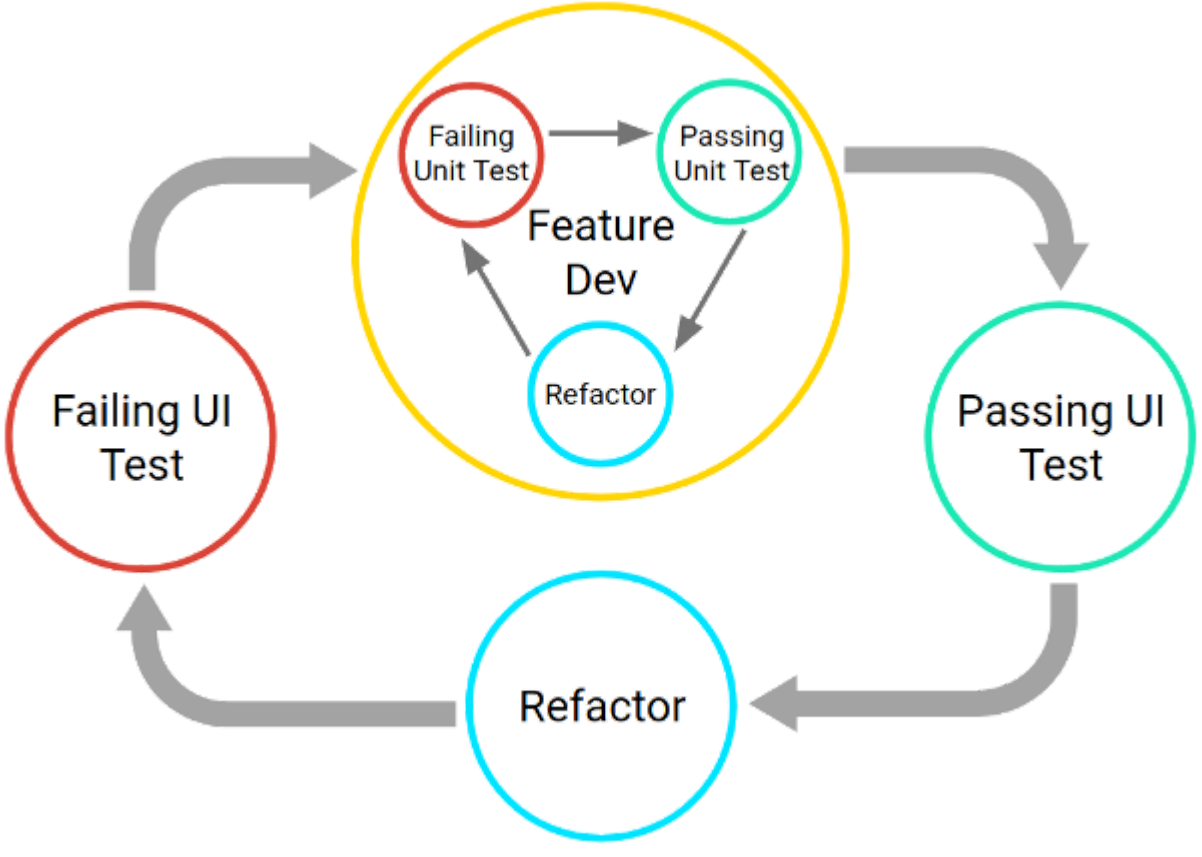
MULTIDEX

```
android {
    defaultConfig {
        ...
        minSdkVersion 21
        targetSdkVersion 28
        multiDexEnabled true
    }
    ...
}

// Required if your minSdkVersion is 20 or lower
dependencies {
  compile 'com.android.support:multidex:1.0.3'
}
```

Speaker notes

- App + referenced libraries exceed 64K methods (65,536 methods)

TESTING



Speaker notes

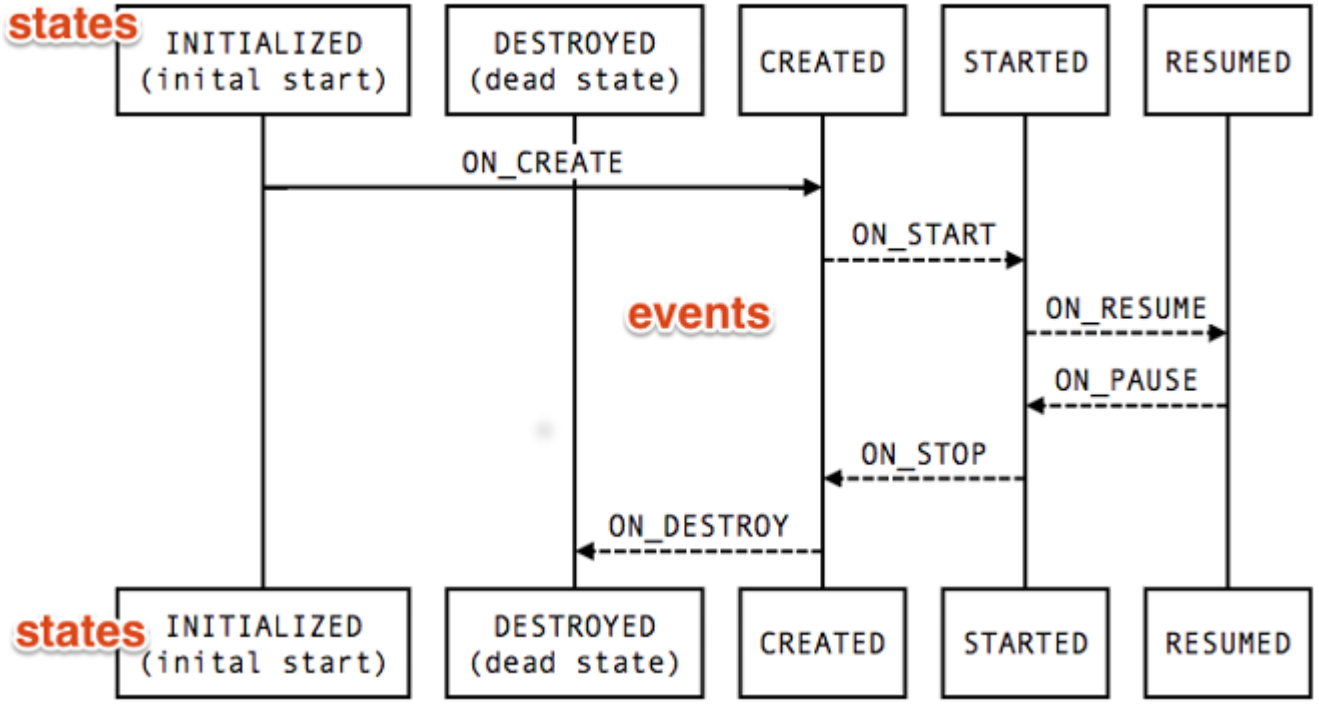
- Write unit tests with JUnit
- Write Android UI tests with Espresso
- Best practices, samples, info on the Jetpack site

ARCHITECTURE COMPONENTS

Speaker notes

- Architecture components help you design robust, testable and maintainable apps.
- Collection of libraries

LIFECYCLES



Speaker notes

- Components go through lifecycle changes based on OS interaction
- Lifecycle-aware components respond to lifecycle status changes
- Use the `android.arch.lifecycle` package

- `Lifecycle`: Contains `Event` and `State`
- `LifecycleOwner`: Contains `getLifecycle()`
- `LifecycleObserver`: Supports `@OnLifecycleEvent(...)` annotation
- `Lifecycle`:
  - `State`: Current state, like `CREATED`
  - `Event`: Change in state, like `ON_START`
- Using a `LifecycleObserver` moves logic out of an Activity's `OnStart(...)`, etc methods
- `LifecycleOwner`
  - Interface with one method, `getLifecycle()`
  - Fragments/Activities implement `LifecycleOwner` starting with Support Library 26.1.0
- `LifecycleObserver`:
  - `@OnLifecycleEvent(Lifecycle.Event.ON_RESUME)`
  - `@OnLifecycleEvent(Lifecycle.Event.ON_PAUSE)`
  - Observer can be added to a `LifecycleOwner`
    - `lifecycle.addObserver(lifeCycleObserver)`
- Able to do this manually, but usually just want to go with lifecycle-aware components
  - `ViewModel`
  - `LiveData`

ViewModel

```
class MyViewModel : ViewModel() {
    private lateinit var users: MutableLiveData<List<User>>

    fun getUsers(): LiveData<List<User>> {
        if (!::users.isInitialized) {
            users = MutableLiveData()
            loadUsers()
        }
        return users
    }

    private fun loadUsers() {
```

Speaker notes

- Sourcing data to your views
- Extend the `ViewModel` abstract class
- Generally used with `LiveData`
- Keeps data logic out of UI layer (separation of concerns)
  - Easier to understand
  - Easier to test
- ViewModels must never reference a view, lifecycle, or any class with an `Activity` context reference

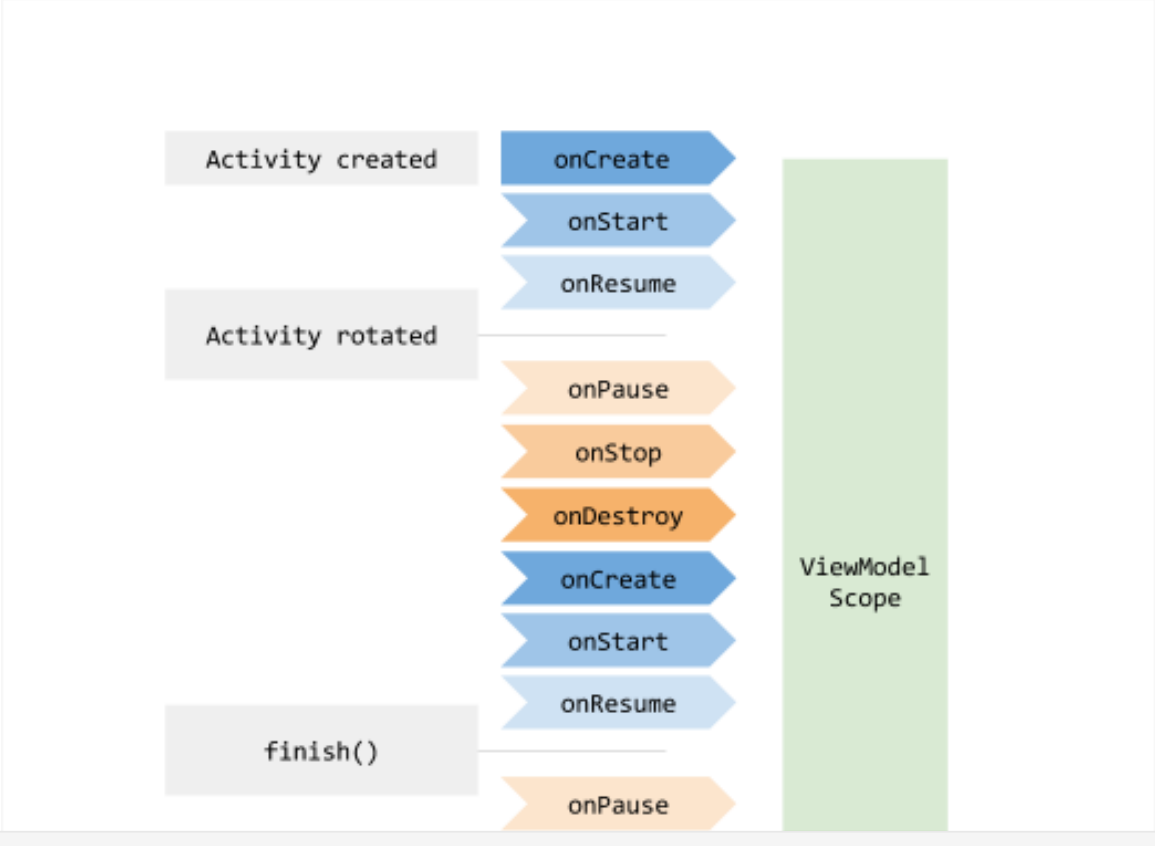
USING A `ViewModel`

```
class MyActivity : AppCompatActivity() {

    override fun onCreate(savedInstanceState: Bundle?) {
        // Create a ViewModel the first time the system calls an activ
        // Re-created activities receive the same MyViewModel instance

        val model = ViewModelProviders.of(this).get(MyViewModel::class
        model.getUsers().observe(this, Observer<List<User>>{ users ->
            // update UI
        })
    }
}
```

Speaker notes

- Observe a `ViewModel`'s `LiveData` and react accordingly
- Can also bind to a `ViewModel` in XML

LIFECYCLE OF A `ViewModel`



Speaker notes

- Lifecycle-aware; saves data through Activity/Fragment lifecycles
  - Re-creating UI doesn't lose data
  - Activities can always get the same `ViewModel` instance
    - ViewModels live until `onCleared()` is called

GETTING THE SAME ViewModel

```
class MasterFragment : Fragment() {
    private lateinit var model: SharedViewModel
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        model = activity?.run {
            ViewModelProviders.of(this).get(SharedViewModel::class.java)
        } ?: throw Exception("Invalid Activity")
    }
}

class DetailFragment : Fragment() {
    private lateinit var model: SharedViewModel
    override fun onCreate(savedInstanceState: Bundle?) {
        super.onCreate(savedInstanceState)
        model = activity?.run {
            ViewModelProviders.of(this).get(SharedViewModel::class.java)
        } ?: throw Exception("Invalid Activity")
    }
```

Speaker notes

+

LIVEDATA

```
public class CurrentWeekViewModel extends ViewModel {
    private MutableLiveData<String> currentWeekHeader;
    public MutableLiveData<String> getCurrentWeekHeader() {
        if(currentWeekHeader == null)
        currentWeekHeader = new MutableLiveData<String>();

        return currentWeekHeader;
    }
}
```

```
final CurrentWeekViewModel vm
    = ViewModelProviders.of(this).get(CurrentWeekViewModel::class);
```

Speaker notes

- Lifecycle-aware observable
- LiveData
  - Create instance with `get...` and `set...`
  - Create observer
  - Attach observer to `LiveData`
  - Just use bindings?

LIVEDATA

ADVANTAGES

- Only updates components when active
- No more manual lifecycle handling
- Ensures your UI matches your data state

Speaker notes

- LiveData updates the UI when the data changes instead of the UI updating on each load
- Bound to lifecycle objects
- Observers clean up after themselves
- No LiveData updating for inactive components
- No need to manually handle lifecycle events
- Updates are ready as soon as an UI component is ready
  - LiveData is updated, but it doesn't alert the UI until it's ready
- On device rotate, for example, latest data is available without having to reload
- Wrap LiveData to share around the app
- Use LiveData with ViewModels

LIVEDATA

```
class CurrentWeekViewModel : ViewModel() {
    val currentWeekHeader: MutableLiveData<String>
        = MutableLiveData<String>()
}
```

```
override fun onCreateView(
    inflater: LayoutInflater, container: ViewGroup?, state: Bundle?):
    val vm = ViewModelProviders
        .of(this)
        .get(CurrentWeekViewModel::class.java)

    val headerObserver = Observer<String> { newValue ->
        headerTextView.text = newValue
    }

    vm.currentWeekHeader.observe(this, headerObserver)
}
```

Speaker notes

- Store LiveData in ViewModels, not Activities/Fragments
- `LiveData` can use `Transformations` to map data before sending to observers

DATA BINDING

```java
// Java
final TextView headerTextView = findViewById(R.id.week_header_text);
headerTextView.setText(viewModel.getCurrentWeekHeader());
```

```kotlin
// Kotlin
findViewById<TextView>(R.id.week_header_text).apply {
    text = viewModel.currentWeekHeader
}
```

```xml
<TextView
    android:id="@+id/week_header_text"
    android:layout_height="wrap_content"
    android:layout_width="0dp"
    android:text="@{currentWeekViewModel.currentWeekHeader}"
    ...
/>
```

Speaker notes

- Bind UI components to data in XML

ENABLE DATA BINDING

```
android {
    ...
    dataBinding {
        enabled = true
    }
}
```

```
val binding = FragmentWeekBinding.inflate(inflater, container, false)
```

Speaker notes

- Must configure data binding for modules that rely on libraries using data binding
- Can be used on apps targeting API level 14 or higher (support library)
- Can inflate UI from the generated binding class
- Tie a view model to a binding object
  - Can also tie other objects
- Binding class is generated automatically
  - Layout name + `Binding`
  - `fragment_week.xml` -> `FragmentWeekBinding`

DATA BINDING

```
<layout xmlns:android="http://schemas.android.com/apk/res/android"
        xmlns:app="http://schemas.android.com/apk/res-auto">
    <data>
        <variable
            name="currentWeekViewModel"
            type="org.faziodev.timetracker.viewmodels.CurrentWeekViewM
            />
    </data>
    <ConstraintLayout... /> <!-- UI layout's root element -->
</layout>
```

Speaker notes

- Add the `<data>` section with one or more `<variable>` tags

DATA BINDING

```
<TextView
    android:id="@+id/week_header_text"
    android:text="@{viewModel.currentWeekHeader}"
    ... />
```

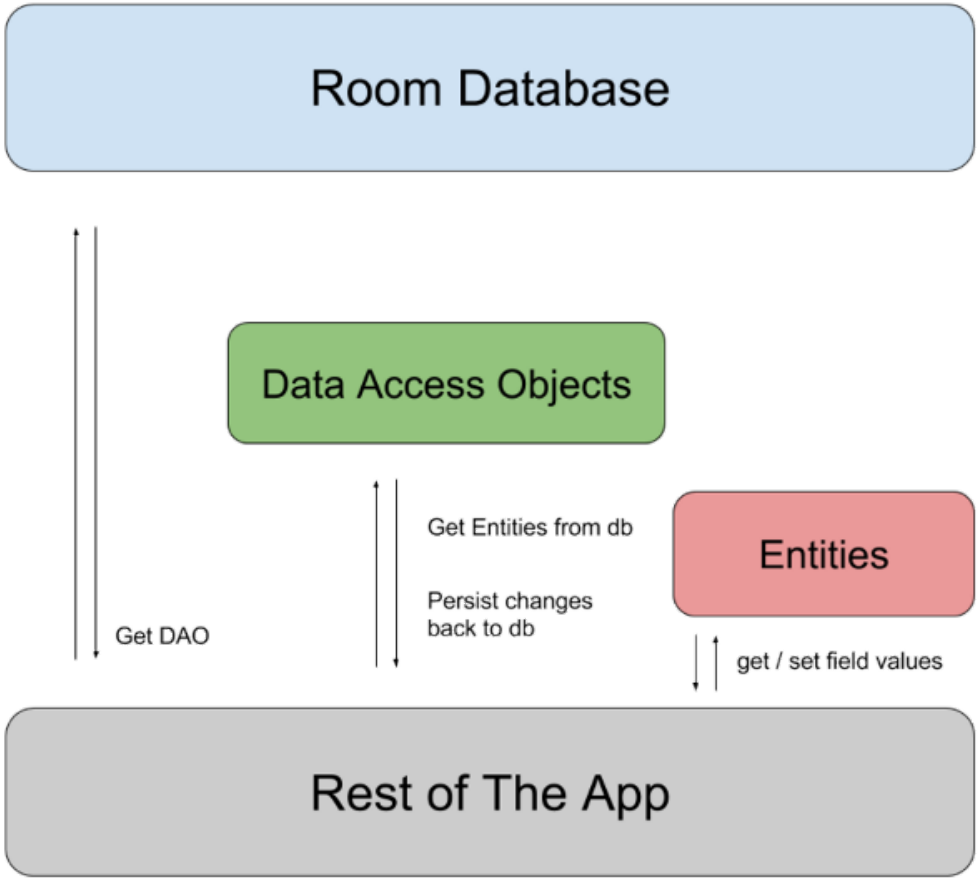
```
<com.google.android.material.floatingactionbutton.FloatingActionButton
    android:id="@+id/week_summary_fab"
    android:onClick="@{ () -> viewModel.onFabClicked() }"
    ... />
```

```
<TextView
    android:id="@+id/week_pace_text"
```

Speaker notes

- Can use most literals, operators, method calls, and more in binding statements
  - Access resources and formats (like strings) within expressions
  - Can even import types
- Two-way binding using a =
  - Can add a converter if the value doesn't match what's displayed
  - Observable fires when the value changes on the UI
- Can add Custom binding adapters if needed

ROOM



https://developer.android.com/training/data-storage/room/

Speaker notes

- Abstraction layer over SQLite

@Entity and @Dao

```
@Entity
data class User(
    @PrimaryKey(autoGenerate = true) var uid: Int,
    @ColumnInfo(name = "first_name") @NonNull var firstName: String,
    @ColumnInfo(name = "last_name") var lastName: String?
)
```

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

Speaker notes

- `@Entity`
  - Table in a database
  - Requires a @PrimaryKey
  - Able to customize the name
    - Otherwise, named after class name
- `@Dao`
  - Methods used to access the DB
  - Queries are verified at compile-time

@Database and RoomDatabase

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

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

Speaker notes

- Database lists entities, views
- Retrieve DAOs from Database
  - Abstract getter method for each `@Dao`

@DatabaseView

```
@DatabaseView("SELECT user.id, user.name, user.departmentId," +
        "department.name AS departmentName FROM user " +
        "INNER JOIN department ON user.departmentId = department.id")
data class UserDetail(
    var id: Long,
    var name: String?,
    var departmentId: Long,
    var departmentName: String?
)
```

```
@Database(entities = arrayOf(User::class),
          views = arrayOf(UserDetail::class), version = 1)
abstract class AppDatabase : RoomDatabase() {
```

Speaker notes

- Add DB views to access particular queries
- Access via `Database.views()`
- Database version is used with migrations
  - Able to use `Migration` class to update DB without losing data
    - Specify SQL to execute for a migration
    - Runs in `Migration`'s `migrate(...)` method

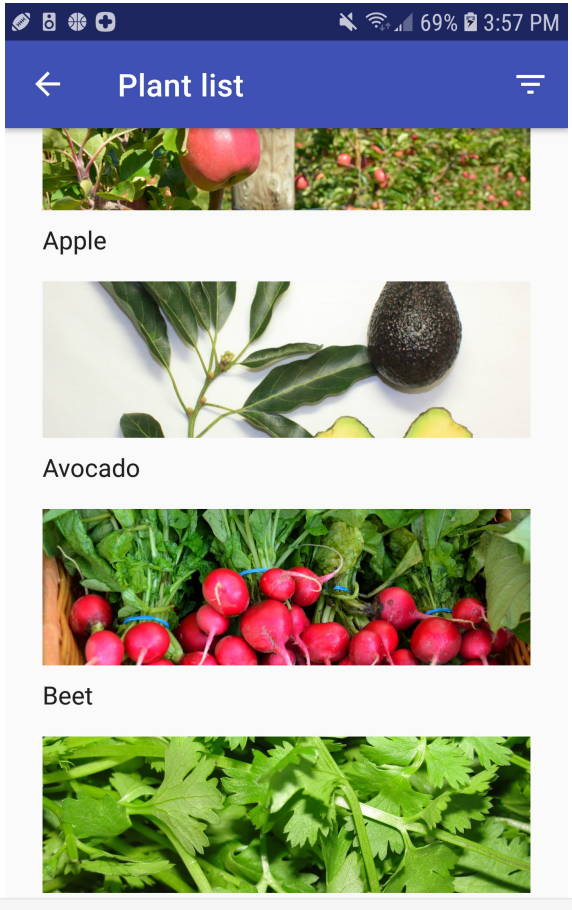
DATABASE MIGRATIONS

```
val MIGRATION_1_2 = object : Migration(1, 2) {
    override fun migrate(database: SupportSQLiteDatabase) {
        database.execSQL(
            "CREATE TABLE `Fruit` (`id` INTEGER, `name` TEXT, " +
                "PRIMARY KEY(`id`))")
    }
}

val MIGRATION_2_3 = object : Migration(2, 3) {
    override fun migrate(database: SupportSQLiteDatabase) {
        database.execSQL("ALTER TABLE Book ADD COLUMN pub_year INTEGER
    }
}
```

Speaker notes

- Database version is used with migrations
  - Able to use `Migration` class to update DB without losing data
    - Specify SQL to execute for a migration
    - Runs in `Migration`'s `migrate(...)` method
- Migrations specify a `startVersion` and `endVersion`
- Docs have tips on testing migrations

PAGING



Speaker notes

- Load data gradually and gracefully in a `RecyclerView`
  - Uses less data/network bandwidth
  - Quickly respond to user input while loading data

PagedList and PagedAdapter

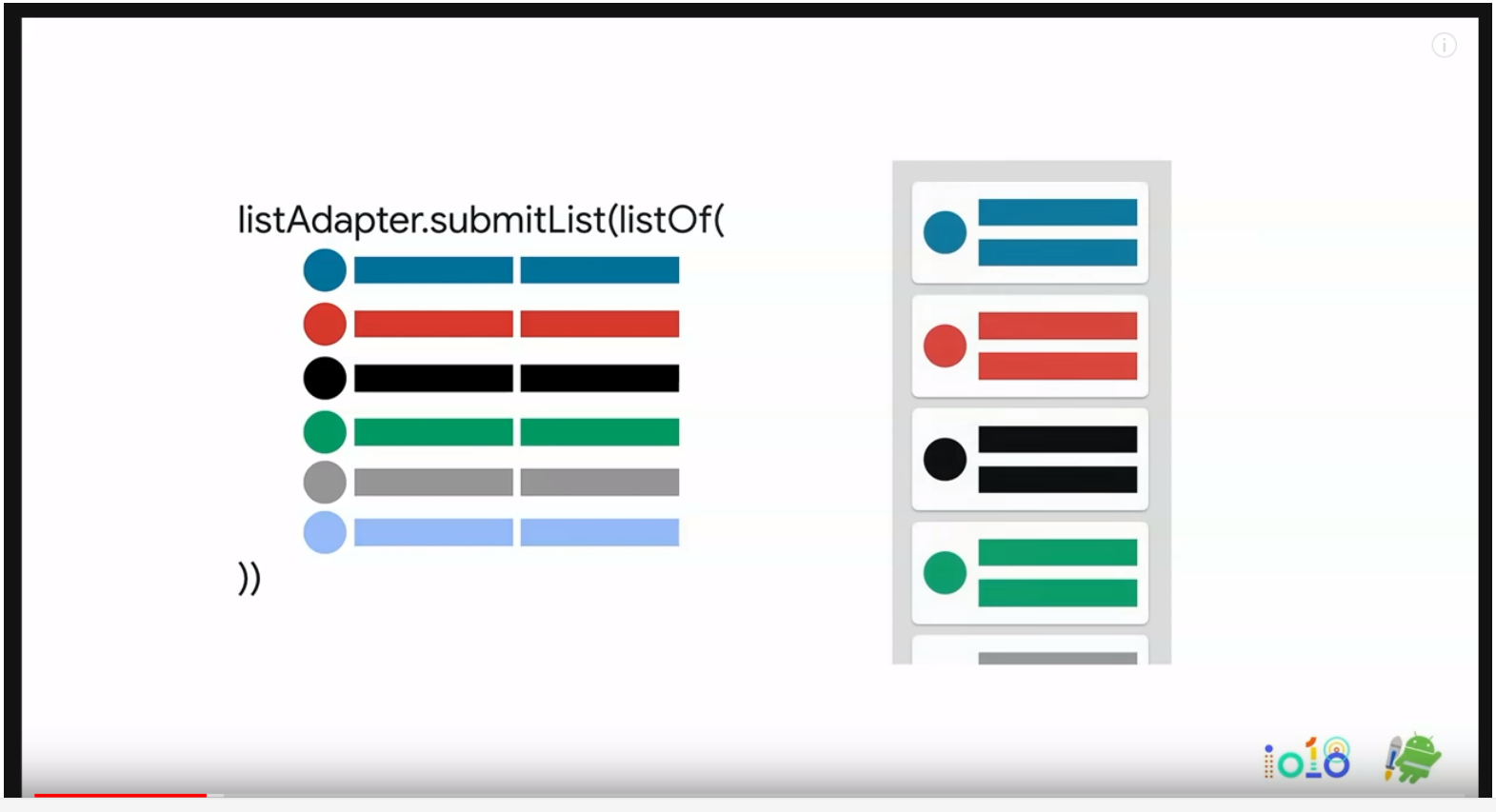
```
// The Int type argument corresponds to a PositionalDataSource object.
val myConcertDataSource : DataSource.Factory<Int, Concert> =
        concertDao.concertsByDate()

val concertList = LivePagedListBuilder(
    myConcertDataSource, /* page size */ 20).build()
```

```
private val adapter = ConcertAdapter()
private lateinit var viewModel: ConcertViewModel
```

Speaker notes

- `PagedList`
  - Loads an up-to-date snapshot of your data
  - Loaded from `DataSource`
  - `LivePagedListBuilder` returns `LiveData<PagedList>`
- `PagedListAdapter`
  - Sends data to a `RecyclerView`
- Can fetch data from server, DB, or both
  - Network calls suggest using Retrofit (from Square)
  - DB calls should use Room

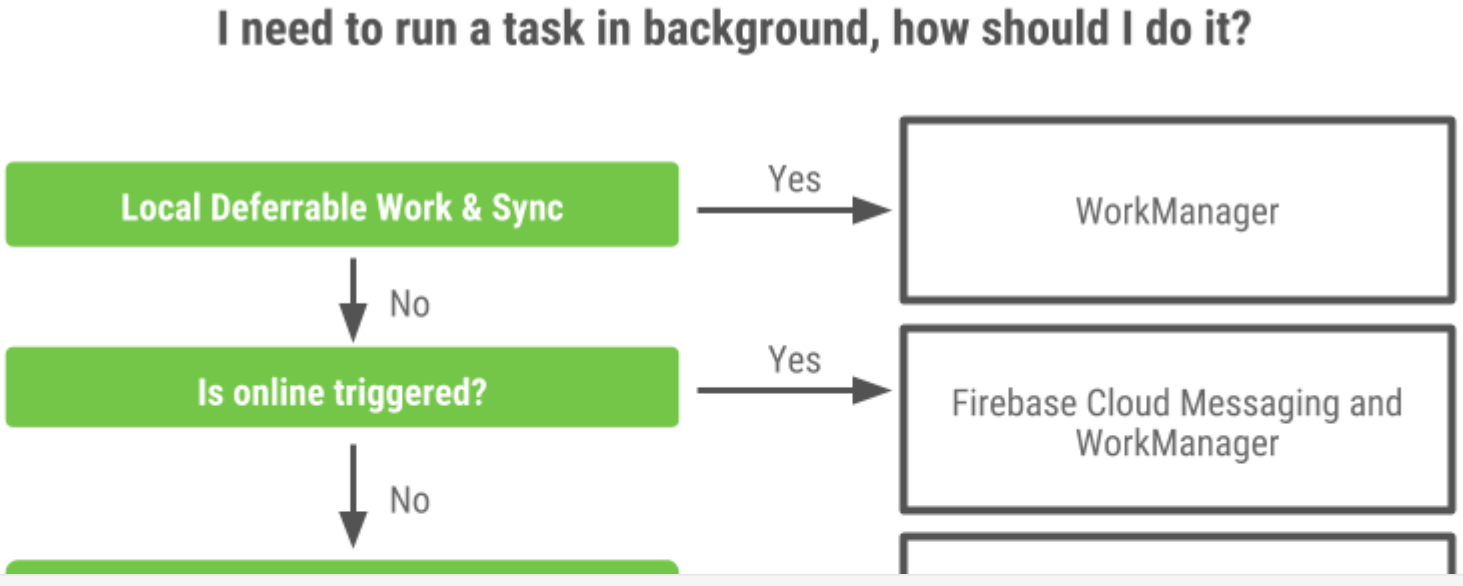
PAGING



Speaker notes

- "Android Jetpack: Manage infinite lists with RecyclerView and Paging"
- Google I/O '18
- Almost 33 minute video
- Out of the scope of this talk

WORKMANAGER



Speaker notes

- Runs deferrable, async tasks
- Determines best way to run based on device API level and app state
  - If app's running, a new thread can run the task
  - If the app is *not* running, a background task is scheduled
    - Could use `JobScheduler`, `Firebase JobDispatcher`, or `AlarmManager`
      - Depends on Device API level and included dependencies
- Intended for tasks that need to run even if the app quits
  - Use `ThreadPools` for tasks that can be stopped when app closes

Worker and WorkRequest

```
class CompressWorker(context : Context, params : WorkerParameters)
    : Worker(context, params) {
    override fun doWork(): Result {
        doWork()
        // Indicate success or failure with your return value:
        // Result.retry() tells WorkManager to try again
        // Result.failure() says not to try again.
        return Result.success()
    }
}
```

Speaker notes

- `Worker` specifies what task you need to perform.
  - WorkManager APIs include an abstract `Worker` class
    - Extend this class and perform the work there
- `WorkRequest` is an individual task
  - Specifies which `Worker` should perform the task
  - Details can be added to `WorkRequests`
    - Circumstances when the request should run (`Constraints`)
  - Abstract class
    - Use `OneTimeWorkRequest` or `PeriodicWorkRequest`

WorkManager and WorkInfo

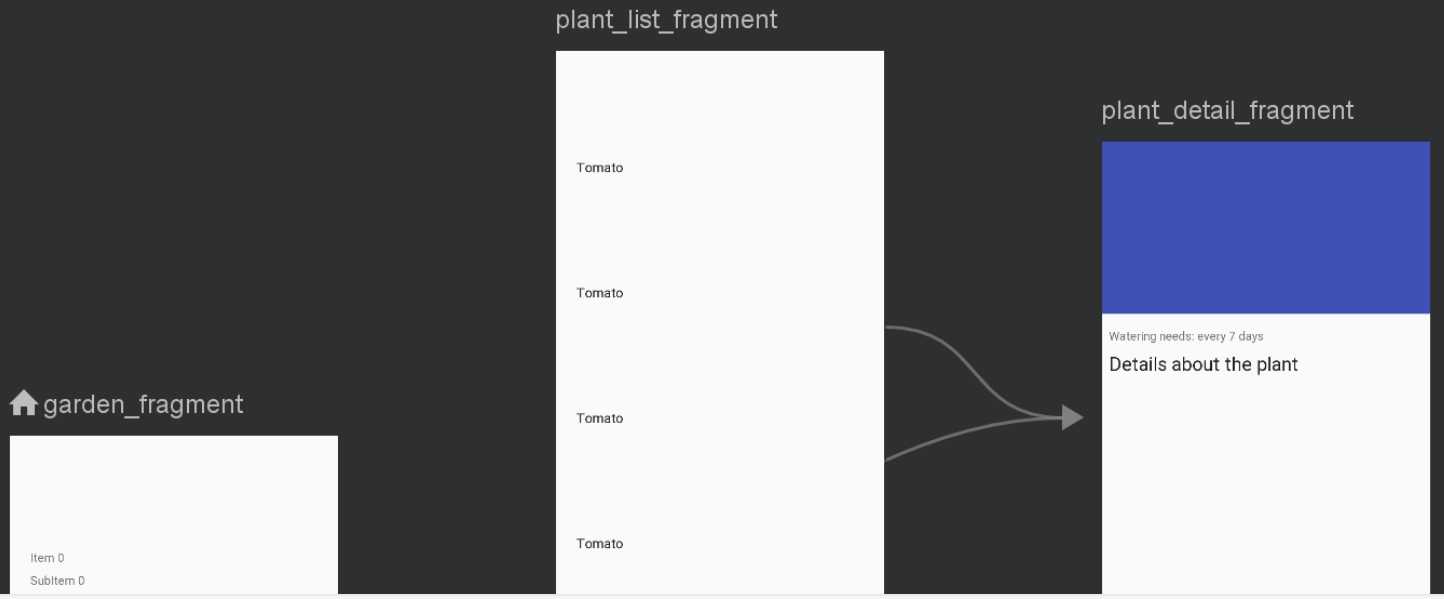
```
// Can also use PeriodicWorkRequest
val compressionWork = OneTimeWorkRequestBuilder<CompressWorker>().buil
WorkManager.getInstance().enqueue(compressionWork)
```

```
WorkManager.getInstance().getWorkInfoByIdLiveData(compressionWork.id)
    .observe(lifecycleOwner, Observer { workInfo ->
        // Do something with the status
        if (workInfo != null && workInfo.state.isFinished) {
            // ...
        }
    })
```

Speaker notes

- `WorkManager`: Enqueues/manages work requests
  - Pass `WorkRequest` to `WorkManager`
  - Scheduled in a way to not overload a device
- `WorkInfo`: Info about a particular task
  - `WorkManager` has `LiveData` for each `WorkRequest` object

NAVIGATION



Speaker notes

- Navigation Architecture Component
  - Handle moving between destinations
    - Fragments
    - Activities
    - Navigation (sub)graphs
    - Custom destination types
  - Destinations are connected via Actions
  - Destinations + Actions == Navigation Graph

NAVIGATION

BENEFITS

- Handle fragment transactions
- Handle Up/Back actions correctly by default
- Providing standardized resources for animations and transactions
- Treating deep linking as a first-class operation
- Use nav drawers and bottom navs with minimal work
- Type safety when passing info during navigation

Speaker notes

+

NAVIGATION

PRINCIPLES

- Fixed start destination
- Nav state should be a stack of destinations
- Up button never exits the app
- Up and back are identical in root

Speaker notes

- One spot where the app starts from the launcher
  - Same screen the user sees before exiting the app when pressing back
- Start destination at the bottom, interact with the top of the stack
- Up button in the nav bar should *never* exit the app
  - When entering with a deep link, should go to start destination
- Up and system back button should be the same unless in the start destination
- Up/back should be able to get the user back to the start destination
- Deep linking removes an existing navigation stack
  - Replaced with the deep link's navigation stack

nav_graph.xml

```
<?xml version="1.0" encoding="utf-8"?>
<navigation xmlns:android="http://schemas.android.com/apk/res/android"
    xmlns:app="http://schemas.android.com/apk/res-auto"
    xmlns:tools="http://schemas.android.com/tools"
    android:id="@+id/nav_graph"
    app:startDestination="@id/weekFragment">

    <fragment
        android:id="@+id/weekFragment"
        android:name="org.faziodev.timetracker.WeekFragment"
        android:label="fragment_week"
        tools:layout="@layout/fragment_week" />
    <fragment
        android:id="@+id/dashboardFragment"
        android:name="org.faziodev.timetracker.DashboardFragment"
        android:label="fragment_dashboard"
        tools:layout="@layout/fragment_dashboard" />
    <fragment
```

Speaker notes

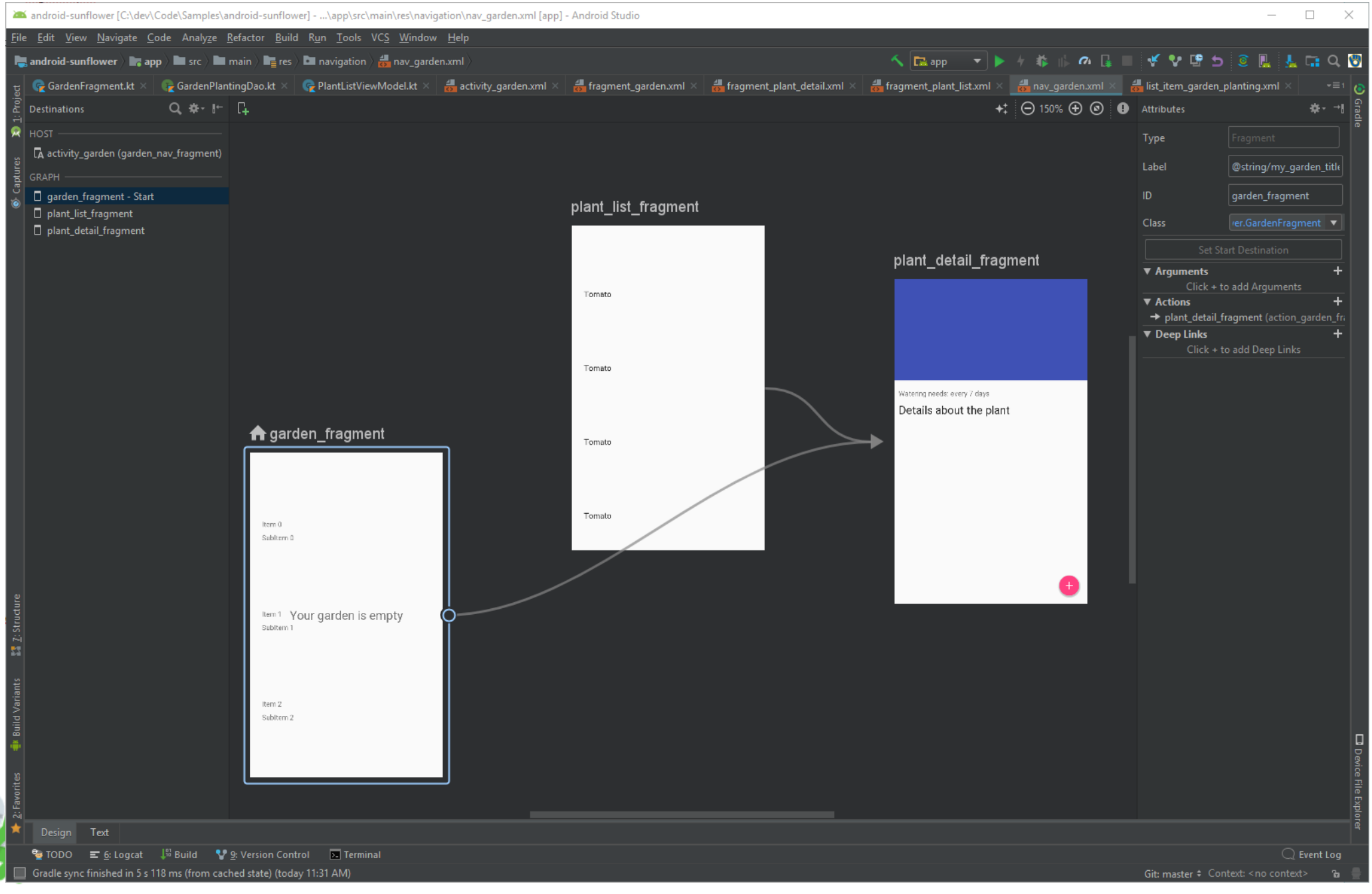
+

NAVIGATION

```
override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)

    val binding: ActivityMainBinding
        = DataBindingUtil.setContentView(this, R.layout.activity_main)

    this.navController
        = Navigation.findNavController(this, R.id.mainFragment)

    binding.bottomNav.setupWithNavController(this.navController)
}
```

Speaker notes

- setupWithNavController can be used on:
  - Bottom nav
  - Navigation drawer
- Match the menu item IDs with the destination IDs


android-sunflower [C:\dev\Code\Samples\android-sunflower] - ...\app\src\main\res\navigation\nav_garden.xml [app] - Android Studio
File Edit View Navigate Code Analyze Refactor Build Run Tools VCS Window Help
android-sunflower app src main res navigation nav_garden.xml
GardenFragment.kt GardenPlantingDao.kt PlantListViewModel.kt activity_garden.xml fragment_garden.xml fragment_plant_detail.xml fragment_plant_list.xml nav_garden.xml list_item_garden_planting.xml
Destinations
HOST
activity_garden (garden_nav_fragment)
GRAPH
garden_fragment - Start
plant_list_fragment
plant_detail_fragment
150%
plant_list_fragment
Tomato
Tomato
Tomato
Tomato
plant_detail_fragment
Watering needs: every 7 days
Details about the plant
garden_fragment
Item 0
SubItem 0
Item 1
Your garden is empty
SubItem 1
Item 2
SubItem 2
Attributes
Type Fragment
Label @string/my_garden_title
ID garden_fragment
Class er.GardenFragment
Set Start Destination
Arguments
Click + to add Arguments
Actions
plant_detail_fragment (action_garden_fr
Deep Links
Click + to add Deep Links
Design Text
TODO 6: Logcat Build 9: Version Control Terminal
Event Log
Gradle sync finished in 5 s 118 ms (from cached state) (today 11:31 AM)
Git: master Context: <no context>

Speaker notes

- Settings > Experimental > Enable Navigation Editor

BEHAVIOR

Speaker notes

- Behavior components help your app integrate with standard Android services like notifications, permissions, sharing and the Assistant.
- Help apps work with devices

BEHAVIOR COMPONENTS

- Download Manager
- Media & Playback
- Notifications

Speaker notes

- Download Manager
  - Schedule and manage large downloads
- Media & Playback
  - Backwards-compatible APIs for media playback and routing (including Google Cast)
  - Shows best practices
    - Client/server set up for audio, single-activity design for videos
- Notifications
  - Backwards-compatible API with support for Wear and Auto
  - More best practices!
- Permissions
  - Compatibility APIs for checking/requesting app permissions
  - Implementation info
  - Even more best practices

PREFERENCES

```
<androidx.preference.PreferenceScreen
    xmlns:app="http://schemas.android.com/apk/res-auto">

    <SwitchPreferenceCompat
        app:key="notifications"
        app:title="Enable message notifications"/>

    <Preference
        app:key="feedback"
        app:title="Send feedback"
        app:summary="Report technical issues or suggest new features"/

</androidx.preference.PreferenceScreen>
```

Speaker notes

- Create interactive settings screens
- AndroidX Preference Library
  - Manages UI and storage interaction for you
  - Create preference hierarchy in XML (or code)
  - Inflate inside a `PreferenceFragment`
  - Add to any Activity as you would a normal fragment

SHARING

```
<menu xmlns:android="http://schemas.android.com/apk/res/android">
    <item
            android:id="@+id/menu_item_share"
            android:showAsAction="ifRoom"
            android:title="Share"
            android:actionProviderClass=
                "android.widget.ShareActionProvider" />
    ...
</menu>
```

Speaker notes

- `ShareActionProvider`
  - Add it to the menu
  - Set up Share `Intent`

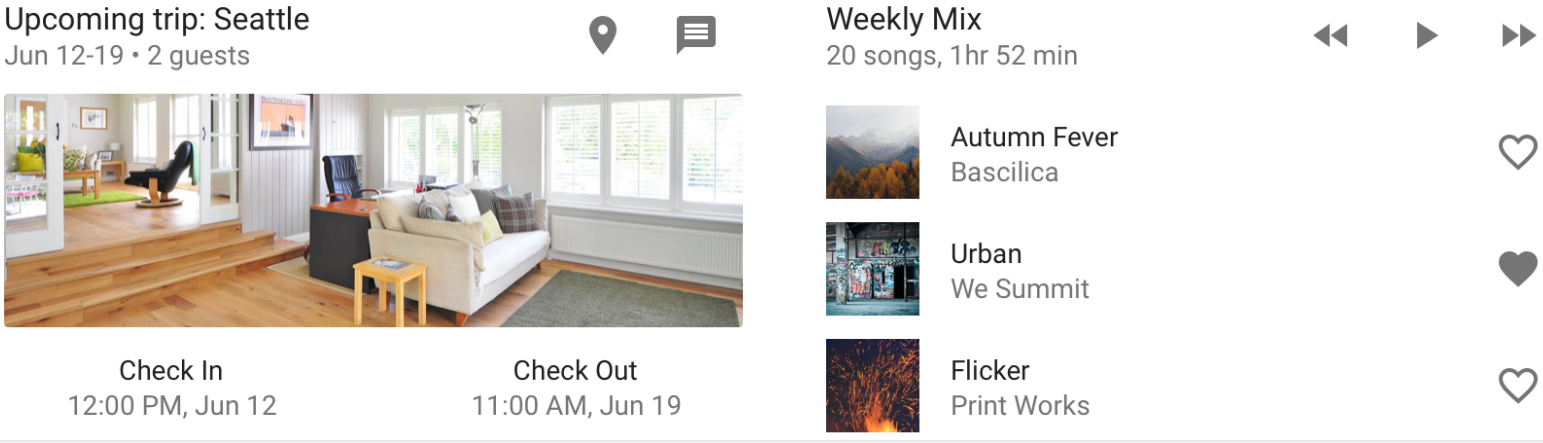
SHARING

```
private var mShareActionProvider: ShareActionProvider? = null
...

override fun onCreateOptionsMenu(menu: Menu): Boolean {
    menuInflater.inflate(R.menu.share_menu, menu)

    menu.findItem(R.id.menu_item_share).also { menuItem ->
        // Fetch and store ShareActionProvider
        mShareActionProvider = menuItem.actionProvider as? ShareActionProvider
    }

    // Return true to display menu
    return true
}

// Call to update the share intent
private fun setShareIntent(shareIntent: Intent) {
```

Speaker notes

- `ShareActionProvider`
  - Add it to the menu
  - Set up Share `Intent`

SLICES



Speaker notes

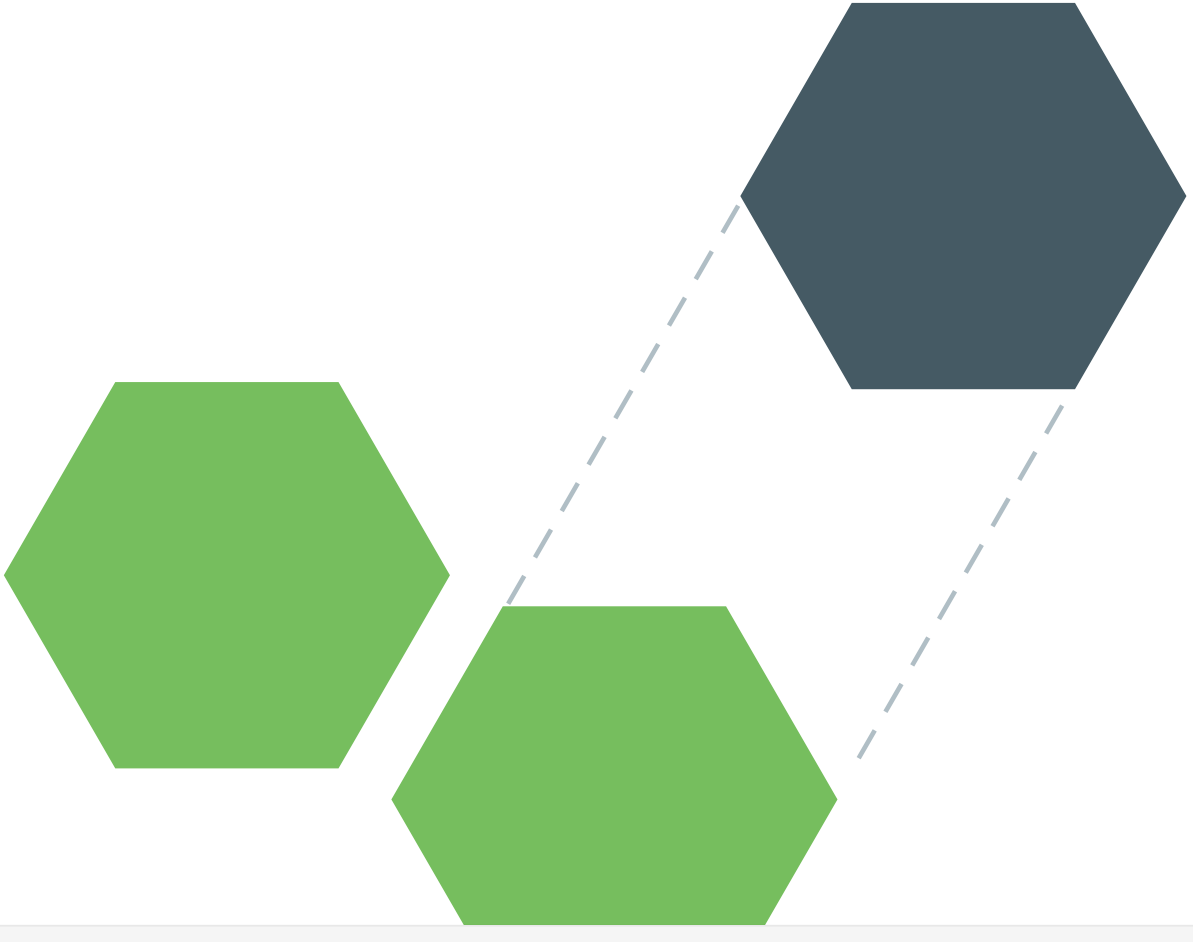
- Show content from your app in Google Search and the Google Assistant
- Use templates to set up/style the UI
- Supports deep linking, `LiveData`, scrolling content, inline actions
- Backwards compatible through KitKat (API level 19)
-

Slices viewer available for testing

-

Slices will start appearing soon for users, but you can start building today.

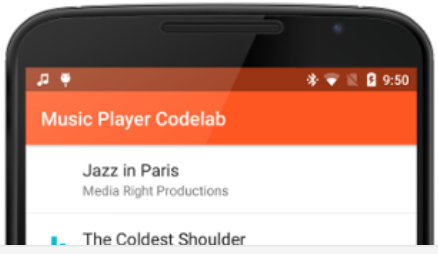
UI



Speaker notes

- UI components provide widgets and helpers to make your app not only easy, but delightful to use.

UI COMPONENTS



Speaker notes

- Animation & Transitions
- Fragments
- Layout
- Android Auto
- Android TV
-

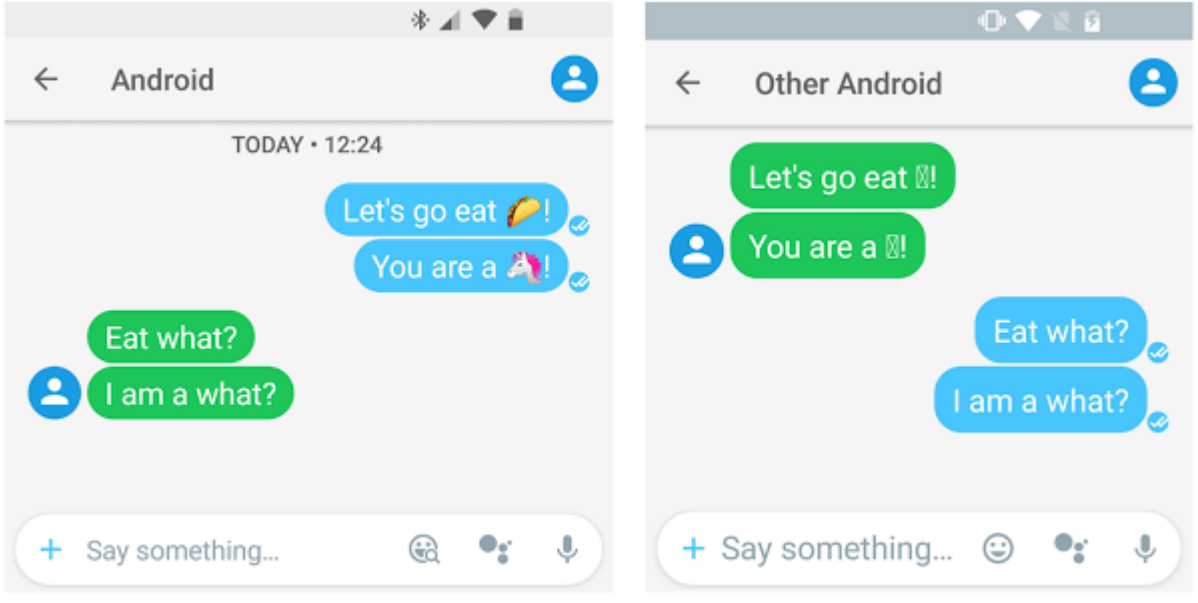
Wear OS by Google

-

First three help shape and form your app

- Latter three are other form factors than a phone/tablet

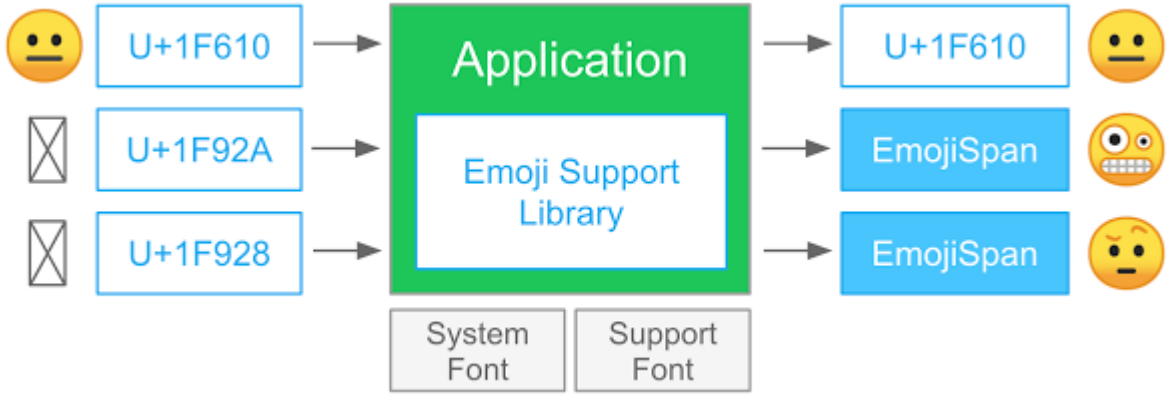
EMOJI



Speaker notes

- `EmojiCompat` support library
  - Gives up-to-date emoji font on older platforms
  - Backwards compatible
- Downloadable or bundled fonts
- Identifies a `CharSequence` to replace with an `EmojiSpan`

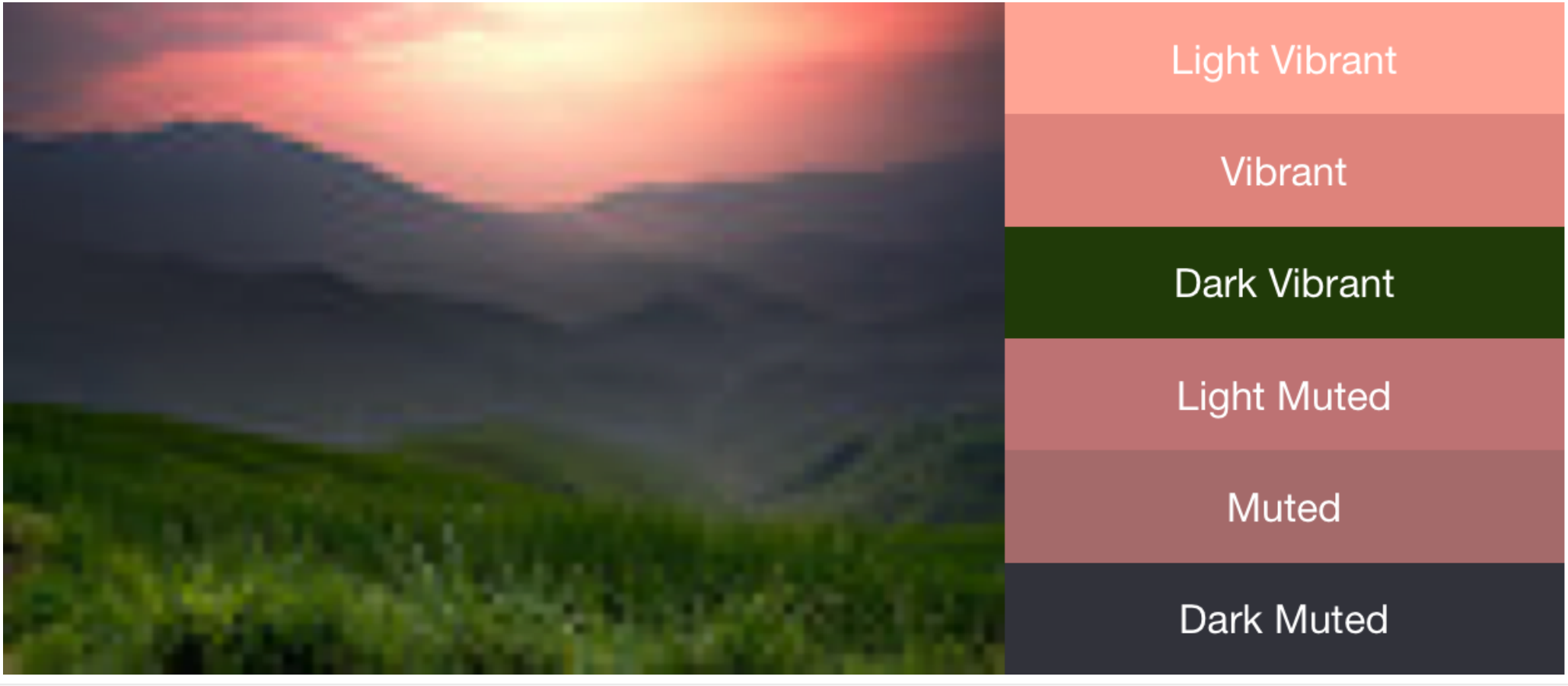
EMOJI



Speaker notes

- Identifies a `CharSequence` to replace with an `EmojiSpan`

PALETTE



Speaker notes

- Android support library 24.0.0 or greater
- Extracts colors from images
- Ex: Album art colors when playing a song

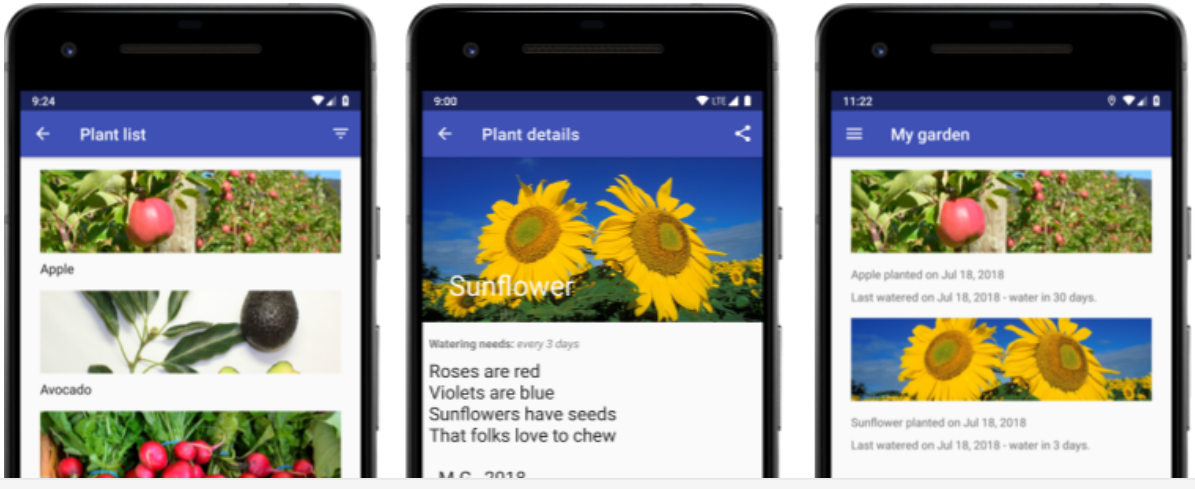
PALETTE

```
// Generate palette synchronously and return it
fun createPaletteSync(bitmap: Bitmap): Palette
    = Palette.from(bitmap).generate()

// Generate palette asynchronously and use it on a different
// thread using onGenerated()
fun createPaletteAsync(bitmap: Bitmap) {
    Palette.from(bitmap).generate { palette ->
        // Use generated instance
    }
}
```

Speaker notes

- Android support library 24.0.0 or greater
- Extracts colors from images
- Ex: Album art colors when playing a song

ANDROID SUNFLOWER



Speaker notes

- Gardening app illustrating how to use Jetpack
- Uses most of what's included in Jetpack
  - Data Binding
  - ViewModels
  - Navigation
  - Room
  - WorkManager
- Also uses
  - Glide
  - Kotlin Coroutines

USEFUL LINKS

- Jetpack Home: https://g.co/jetpack
- Getting Started with Jetpack: https://goo.gl/bGnL7N
- Introducing Android Jetpack: https://youtu.be/LmkKFCfmnhQ
- Jetpack on YouTube: https://l.faziodev.org/jetpack-youtube
- Paging on YouTube: https://l.faziodev.org/paging-youtube
- Android Sunflower: https://l.faziodev.org/jetpack
- Room with a View: https://l.faziodev.org/android-rwav-kotlin

Speaker notes

+

QUESTIONS?

Speaker notes

- "What questions do you have?"

THANKS!

Speaker notes

+