Appendix A: Hands-On Converting RxJava to Coroutines

This appendix offers a hands-on tutorial, akin to those from *Exploring Android*. In this tutorial, we will convert an app that uses RxJava to have it use coroutines instead.

To be able to follow along in this tutorial, it will help to read the first few chapters of this book, plus have basic familiarity with how RxJava works. You do not need to be an RxJava expert, let alone a coroutines expert, though.

You can download the code for the initial project and follow the instructions in this tutorial to replace RxJava with coroutines. Or, you can download the code for the project after the changes to see the end result.

WARNING: The resulting coroutines-based app will be using a few pre-release artifacts. Consider this a "preview of coming attractions", more than a recommendation to use pre-release artifacts in production.

About the App

The `WeatherWorkshop` app will display the weather conditions from the US National Weather Service KDCA weather station. This weather station covers the Washington, DC area.

When you launch the app, it will immediately attempt to contact the National Weather Service's Web service API to get the current conditions at KDCA (an "observation"). Once it gets them, the app will save those conditions in a database, plus display them in a row of a list:

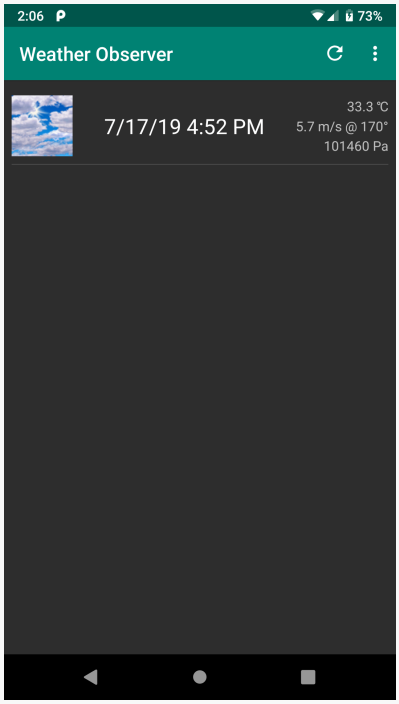


Figure 6: Weather Workshop Sample App, Showing Weather Info

The "refresh" toolbar button will attempt to get a fresh set of current conditions. However, the weather station only reports conditions every hour or so. If you wind up getting the same set of current conditions as before, the "new" set is not added to the database or the list. If, however, you request the current conditions substantially later than the most-recent entry, the new conditions will be added to the database and list.

The overflow menu contains a "clear" option that clears the database and the list. You can then use "refresh" to request the current conditions.

Step #1: Reviewing What We Have

First, let's spend a bit reviewing the current code, so you can see how it all integrates and how it leverages RxJava.

ObservationRemoteDataSource

This app uses Retrofit for accessing the National Weather Service's Web service. The Retrofit logic is encapsulated in an `ObservationRemoteDataSource`:

```
package com.commonsware.coroutines.weather

import io.reactivex.Single
import okhttp3.OkHttpClient
import retrofit2.Retrofit
import retrofit2.adapter.rxjava2.RxJava2CallAdapterFactory
import retrofit2.converter.moshi.MoshiConverterFactory
import retrofit2.http.GET
import retrofit2.http.Path

private const val STATION = "KDCA"
private const val API_ENDPOINT = "https://api.weather.gov"

class ObservationRemoteDataSource(ok: OkHttpClient) {
  private val retrofit = Retrofit.Builder()
    .client(ok)
    .baseUrl(API_ENDPOINT)
    .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
    .addConverterFactory(MoshiConverterFactory.create())
    .build()
  private val api = retrofit.create(ObservationApi::class.java)

  fun getCurrentObservation() = api.getCurrentObservation(STATION)
}

private interface ObservationApi {
  @GET("/stations/{stationId}/observations/current")
  fun getCurrentObservation(@Path("stationId") stationId: String): Single<ObservationResponse>
}

data class ObservationResponse(
  val id: String,
  val properties: ObservationProperties
)

data class ObservationProperties(
  val timestamp: String,
  val icon: String,
  val temperature: ObservationValue,
  val windDirection: ObservationValue,
  val windSpeed: ObservationValue,
  val barometricPressure: ObservationValue
)

data class ObservationValue(
  val value: Double?,
  val unitCode: String
)
```

Most of `ObservationRemoteDataSource` could work with any weather station; we hard-code `"KDCA"` to keep the app simpler.

We have just one function in our `ObservationApi`: `getCurrentObservation()`. We are using Retrofit's RxJava extension, and `getCurrentObservation()` is set up to return an RxJava `Single`, to deliver us a single response from the Web service. That response is in the form of an `ObservationResponse`, which is a simple Kotlin class that models the JSON response that we will get back from the Web service.

Of note in that response:

- `id` in `ObservationResponse` is a server-supplied unique identifier for this set of conditions
- The actual weather conditions are in a `properties` property
- `icon` holds the URL to an icon that indicates the sort of weather that DC is having (sunny, cloudy, rain, snow, sleet, thunderstorm, kaiju attack, etc.)
- The core conditions are represented as `ObservationValue` tuples of a numeric reading and a string indicating the units that the reading is in (e.g., `unit:degC` for a temperature in degrees Celsius)

`ObservationDatabase`

Our local storage of conditions is handled via Room and an `ObservationDatabase`:

```
package com.commonsware.coroutines.weather

import android.content.Context
import androidx.annotation.NonNull
import androidx.room.*
import io.reactivex.Completable
import io.reactivex.Observable

private const val DB_NAME = "weather.db"

@Entity(tableName = "observations")
data class ObservationEntity(
  @PrimaryKey @NonNull val id: String,
  val timestamp: String,
  val icon: String,
  val temperatureCelsius: Double?,
  val windDirectionDegrees: Double?,
  val windSpeedMetersSecond: Double?,
  val barometricPressurePascals: Double?
) {
  fun toModel() = ObservationModel(
    id = id,
    timestamp = timestamp,
```

```
    icon = icon,
    temperatureCelsius = temperatureCelsius,
    windDirectionDegrees = windDirectionDegrees,
    windSpeedMetersSecond = windSpeedMetersSecond,
    barometricPressurePascals = barometricPressurePascals
  )
}

@Dao
interface ObservationStore {
  @Query("SELECT * FROM observations ORDER BY timestamp DESC")
  fun load(): Observable<List<ObservationEntity>>

  @Insert(onConflict = OnConflictStrategy.IGNORE)
  fun save(entity: ObservationEntity): Completable

  @Query("DELETE FROM observations")
  fun clear(): Completable
}

@Database(entities = [ObservationEntity::class], version = 1)
abstract class ObservationDatabase : RoomDatabase() {
  abstract fun observationStore(): ObservationStore

  companion object {
    fun create(context: Context) =
      Room.databaseBuilder(context, ObservationDatabase::class.java, DB_NAME)
        .build()
  }
}
```

The JSON that we get back from the Web service is very hierarchical. The Room representation, by contrast, is a single `observations` table, modeled by an `ObservationEntity`. `ObservationEntity` holds the key pieces of data from the JSON response in individual fields, with the core conditions readings normalized to a single set of units (e.g., `temperatureCelsius`).

From an RxJava standpoint, we are using Room's RxJava extensions, and so our `ObservationStore` DAO has a `load()` function that returns an `Observable` for our select-all query, giving us a `List` of `ObservationEntity`objects. We also have a `save()` function to insert a new reading (if we do not already have one for its ID) and a `clear()` function to delete all current rows from the table. Those are set to return `Completable`, so we can find out when they are done with their work but are not expecting any particular data back.

ObservationRepository

The gateway to ObservationRemoteDataSource and ObservationDatabase is the ObservationRepository. This provides the public API that our UI uses to request weather observations:

```
package com.commonsware.coroutines.weather

import io.reactivex.Completable
import io.reactivex.Observable
import io.reactivex.schedulers.Schedulers

interface IObservationRepository {
  fun load(): Observable<List<ObservationModel>>
  fun refresh(): Completable
  fun clear(): Completable
}

class ObservationRepository(
  private val db: ObservationDatabase,
  private val remote: ObservationRemoteDataSource
) : IObservationRepository {
  override fun load(): Observable<List<ObservationModel>> = db.observationStore()
    .load()
    .map { entities -> entities.map { it.toModel() }
  }

  override fun refresh() = remote.getCurrentObservation()
      .subscribeOn(Schedulers.io())
      .map { convertToEntity(it) }
      .flatMapCompletable { db.observationStore().save(it) }

  override fun clear() = db.observationStore().clear()

  private fun convertToEntity(response: ObservationResponse): ObservationEntity {
    when {
      response.properties.temperature.unitCode != "unit:degC" ->
        throw IllegalStateException(
          "Unexpected temperature unit: ${response.properties.temperature.unitCode}"
        )
      response.properties.windDirection.unitCode != "unit:degree_(angle)" ->
        throw IllegalStateException(
          "Unexpected windDirection unit: ${response.properties.windDirection.unitCode}"
        )
      response.properties.windSpeed.unitCode != "unit:m_s-1" ->
        throw IllegalStateException(
          "Unexpected windSpeed unit: ${response.properties.windSpeed.unitCode}"
        )
      response.properties.barometricPressure.unitCode != "unit:Pa" ->
        throw IllegalStateException(
          "Unexpected barometricPressure unit: ${response.properties.barometricPressure.unitCode}"
        )
    }

    return ObservationEntity(
      id = response.id,
      icon = response.properties.icon,
```

```
      timestamp = response.properties.timestamp,
      temperatureCelsius = response.properties.temperature.value,
      windDirectionDegrees = response.properties.windDirection.value,
      windSpeedMetersSecond = response.properties.windSpeed.value,
      barometricPressurePascals = response.properties.barometricPressure.value
    )
  }
}

data class ObservationModel(
  val id: String,
  val timestamp: String,
  val icon: String,
  val temperatureCelsius: Double?,
  val windDirectionDegrees: Double?,
  val windSpeedMetersSecond: Double?,
  val barometricPressurePascals: Double?
)
```

There are three public functions in that API, which is defined on an `IObservationRepository` interface:

- `load()` calls `load()` on our DAO and use RxJava's `map()` operator to convert each of the `ObservationEntity` objects into a corresponding `ObservationModel`, so our UI layer is isolated from any changes in our database schema that future Room versions might require. `load()` then returns an `Observable` of our list of models.
- `refresh()` calls `getCurrentObservation()` on our `ObservationRemoteDataSource`, uses RxJava's `map()` to convert that to a model, then uses RxJava's `flatMapCompletable()` to wrap a call to `save()` on our DAO. The net result is that `refresh()` returns a `Completable` to indicate when the refresh operation is done. `refresh()` itself does not return fresh data — instead, we are relying upon Room to update the `Observable` passed through `load()` when a refresh results in a new row in our table.
- `clear()` is just a pass-through to the corresponding `clear()` function on our DAO

`KoinApp`

All of this is knitted together by Koin as a service loader/dependency injector framework. Our Koin module — defined in a `KoinApp` subclass of `Application` — sets up the `OkHttpClient` instance, our `ObservationDatabase`, our `ObservationRemoteDataSource`, and our `ObservationRepository`, among other things:

```
package com.commonsware.coroutines.weather
```

```
import android.app.Application
import com.jakewharton.threetenabp.AndroidThreeTen
import okhttp3.OkHttpClient
import org.koin.android.ext.android.startKoin
import org.koin.android.ext.koin.androidContext
import org.koin.androidx.viewmodel.ext.koin.viewModel
import org.koin.dsl.module.module
import org.threeten.bp.format.DateTimeFormatter
import org.threeten.bp.format.FormatStyle

class KoinApp : Application() {
  private val koinModule = module {
    single { OkHttpClient.Builder().build() }
    single { ObservationDatabase.create(androidContext()) }
    factory { ObservationRemoteDataSource(get()) }
    single<IObservationRepository> { ObservationRepository(get(), get()) }
    single { DateTimeFormatter.ofLocalizedDateTime(FormatStyle.SHORT) }
    viewModel { MainMotor(get(), get(), androidContext()) }
  }

  override fun onCreate() {
    super.onCreate()

    AndroidThreeTen.init(this);
    startKoin(this, listOf(koinModule))
  }
}
```

`MainMotor`

The app uses a variation on the model-view-intent (MVI) GUI architecture pattern. The UI (`MainActivity`) observes a stream of view-state objects (`MainViewState`) and calls functions that result in a fresh view-state being delivered to reflect any subsequent changes in the UI content.

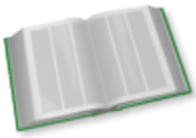
`MainMotor` is what implements those functions, such as `refresh()` and `clear()`, and is what manages the stream of view-state objects:

```
package com.commonsware.coroutines.weather

import android.content.Context
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import io.reactivex.android.schedulers.AndroidSchedulers
import io.reactivex.disposables.CompositeDisposable
```

```
import io.reactivex.rxkotlin.subscribeBy
import org.threeten.bp.OffsetDateTime
import org.threeten.bp.ZoneId
import org.threeten.bp.format.DateTimeFormatter

data class RowState(
  val timestamp: String,
  val icon: String,
  val temp: String?,
  val wind: String?,
  val pressure: String?
) {
  companion object {
    fun fromModel(
      model: ObservationModel,
      formatter: DateTimeFormatter,
      context: Context
    ): RowState {
      val timestampDateTime = OffsetDateTime.parse(
        model.timestamp,
        DateTimeFormatter.ISO_OFFSET_DATE_TIME
      )
      val easternTimeId = ZoneId.of("America/New_York")
      val formattedTimestamp =
        formatter.format(timestampDateTime.atZoneSameInstant(easternTimeId))

      return RowState(
        timestamp = formattedTimestamp,
        icon = model.icon,
        temp = model.temperatureCelsius?.let {
          context.getString(
            R.string.temp,
            it
          )
        },
        wind = model.windSpeedMetersSecond?.let { speed ->
          model.windDirectionDegrees?.let {
            context.getString(
              R.string.wind,
              speed,
              it
            )
          }
        },
        pressure = model.barometricPressurePascals?.let {
          context.getString(
            R.string.pressure,
            it
```

```
          )
        }
      )
    }
  }
}

sealed class MainViewState {
  object Loading : MainViewState()
  data class Content(val observations: List<RowState>) : MainViewState()
  data class Error(val throwable: Throwable) : MainViewState()
}

class MainMotor(
  private val repo: IObservationRepository,
  private val formatter: DateTimeFormatter,
  private val context: Context
) : ViewModel() {
  private val _states =
    MutableLiveData<MainViewState>().apply { value = MainViewState.Loading }
  val states: LiveData<MainViewState> = _states
  private val sub = CompositeDisposable()

  init {
    sub.add(
      repo.load()
        .observeOn(AndroidSchedulers.mainThread())
        .subscribeBy(onNext = { models ->
          _states.value = MainViewState.Content(models.map {
            RowState.fromModel(it, formatter, context)
          })
        }, onError = { _states.value = MainViewState.Error(it) })
    )
  }

  override fun onCleared() {
    sub.dispose()
  }

  fun refresh() {
    sub.add(repo.refresh()
      .observeOn(AndroidSchedulers.mainThread())
      .subscribeBy(onError = { _states.value = MainViewState.Error(it) })
    )
  }

  fun clear() {
    sub.add(repo.clear()
```

```
      .observeOn(AndroidSchedulers.mainThread())
      .subscribeBy(onError = { _states.value = MainViewState.Error(it) })
    )
  }
}
```



You can learn more about the MVI GUI architecture pattern in the "Adding Some Architecture" chapter of *Elements of Android Jetpack*!

`MainMotor` itself is an `AndroidViewModel` from the Jetpack viewmodel system. Our Koin module supports injecting viewmodels, so it is set up to provide an instance of `MainMotor` to `MainActivity` when needed.

When our motor is instantiated, we immediately call `load()` on our repository. We then go through a typical RxJava construction to:

- Arrange to do the I/O on a background thread (`subscribeOn(Schedulers.io())`)
- Arrange to consume the results of that I/O on the main application thread (`observeOn(AndroidSchedulers.mainThread())`)
- Supply lambda expressions for consuming both the success and failure cases
- Add the resulting `Disposable` to a `CompositeDisposable`, which we clear when the viewmodel itself is cleared in `onCleared()`

The net effect is that whenever we get fresh models from the repository-supplied `Observable`, we update a `MutableLiveData` with a `MainViewState` wrapping either the list of models (`MainViewState.Content`) or the exception that we got (`MainViewState.Error`).

The motor's `refresh()` and `clear()` functions call `refresh()` and `clear()` on the repository, with the same sort of RxJava setup to handle threading and any errors. In the case of `refresh()` and `clear()`, though, since we get RxJava `Completable` objects from the repository, there is nothing to do for the success cases — we just wait for Room to deliver fresh models through the `load()` `Observable`.

Note that `MainViewState` actually has three states: `Loading`, `Content`, and `Error`. `Loading` is our initial state, set up when we initialize the `MutableLiveData`. The other states are used by the main code of the motor. The "content" for the `Content` state consists of a list of `RowState` objects, where each `RowState` is a formatted

rendition of the key bits of data from an observation (e.g., formatting the date after converting it to Washington DC's time zone). Note that the `DateTimeFormatter` comes from Koin via the constructor, so we can use a formatter for tests that is locale-independent.

`MainActivity`

Our UI consists of a `RecyclerView` and `ProgressBar` inside of a `ConstraintLayout`:

```
<?xml version="1.0" encoding="utf-8"?>
<androidx.constraintlayout.widget.ConstraintLayout 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:layout_width="match_parent"
  android:layout_height="match_parent"
  tools:context=".MainActivity">

  <androidx.recyclerview.widget.RecyclerView
    android:id="@+id/observations"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:layout_margin="8dp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintLeft_toLeftOf="parent"
    app:layout_constraintRight_toRightOf="parent"
    app:layout_constraintTop_toTopOf="parent">

  </androidx.recyclerview.widget.RecyclerView>

  <ProgressBar
    android:id="@+id/progress"
    style="?android:attr/progressBarStyle"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:layout_marginBottom="8dp"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:layout_marginTop="8dp"
    app:layout_constraintBottom_toBottomOf="parent"
    app:layout_constraintEnd_toEndOf="parent"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent" />

</androidx.constraintlayout.widget.ConstraintLayout>
```

Each row of the `RecyclerView` will show the relevant bits of data from our observations, including an `ImageView` for the icon:

```
<?xml version="1.0" encoding="utf-8"?>
<layout 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">
```

```
<data>

  <variable
    name="state"
    type="com.commonsware.coroutines.weather.RowState" />
</data>

<androidx.constraintlayout.widget.ConstraintLayout
  android:layout_width="match_parent"
  android:layout_height="wrap_content"
  android:paddingBottom="8dp"
  android:paddingTop="8dp">

  <ImageView
    android:id="@+id/icon"
    android:layout_width="0dp"
    android:layout_height="64dp"
    android:contentDescription="@string/icon"
    app:imageUrl="@{state.icon}"
    app:layout_constraintDimensionRatio="1:1"
    app:layout_constraintStart_toStartOf="parent"
    app:layout_constraintTop_toTopOf="parent"
    tools:srcCompat="@tools:sample/backgrounds/scenic" />

  <TextView
    android:id="@+id/timestamp"
    android:layout_width="0dp"
    android:layout_height="0dp"
    android:layout_marginEnd="8dp"
    android:layout_marginStart="8dp"
    android:gravity="center"
    android:text="@{state.timestamp}"
    android:textAppearance="@style/TextAppearance.AppCompat.Large"
    app:autoSizeMaxTextSize="16sp"
    app:autoSizeTextType="uniform"
    app:layout_constraintBottom_toBottomOf="@id/icon"
    app:layout_constraintEnd_toStartOf="@id/barrier"
    app:layout_constraintStart_toEndOf="@id/icon"
    app:layout_constraintTop_toTopOf="@id/icon"
    tools:text="7/5/19 13:52" />

  <TextView
    android:id="@+id/temp"
    android:layout_width="wrap_content"
    android:layout_height="wrap_content"
    android:text="@{state.temp}"
    android:textAppearance="@style/TextAppearance.AppCompat.Small"
```

```
      app:layout_constraintBottom_toTopOf="@id/wind"
      app:layout_constraintEnd_toEndOf="parent"
      app:layout_constraintTop_toTopOf="@id/icon"
      tools:text="20 C" />

    <TextView
      android:id="@+id/wind"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@{state.wind}"
      android:textAppearance="@style/TextAppearance.AppCompat.Small"
      app:layout_constraintBottom_toTopOf="@id/pressure"
      app:layout_constraintEnd_toEndOf="parent"
      app:layout_constraintTop_toBottomOf="@id/temp"
      tools:text="5.1 m/s @ 170°" />

    <TextView
      android:id="@+id/pressure"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      android:text="@{state.pressure}"
      android:textAppearance="@style/TextAppearance.AppCompat.Small"
      app:layout_constraintBottom_toBottomOf="@id/icon"
      app:layout_constraintEnd_toEndOf="parent"
      app:layout_constraintTop_toBottomOf="@id/wind"
      tools:text="10304 Pa" />

    <androidx.constraintlayout.widget.Barrier
      android:id="@+id/barrier"
      android:layout_width="wrap_content"
      android:layout_height="wrap_content"
      app:barrierDirection="start"
      app:constraint_referenced_ids="temp,wind,pressure" />

  </androidx.constraintlayout.widget.ConstraintLayout>
</layout>
```

The row layout uses data binding, with binding expressions to update views based on `RowState` properties. For the `ImageView`, this requires a binding adapter, in this case using Glide:

```
package com.commonsware.coroutines.weather

import android.widget.ImageView
import androidx.databinding.BindingAdapter
import com.bumptech.glide.Glide
```

```
@BindingAdapter("imageUrl")
fun ImageView.loadImage(url: String?) {
  url?.let {
    Glide.with(context)
      .load(it)
      .into(this)
  }
}
```

The `MainActivity` Kotlin file not only contains the activity code, but also the adapter and view-holder for our `RecyclerView`:

```
package com.commonsware.coroutines.weather

import android.os.Bundle
import android.util.Log
import android.view.Menu
import android.view.MenuItem
import android.view.View
import android.view.ViewGroup
import android.widget.Toast
import androidx.appcompat.app.AppCompatActivity
import androidx.lifecycle.Observer
import androidx.recyclerview.widget.*
import com.commonsware.coroutines.weather.databinding.RowBinding
import kotlinx.android.synthetic.main.activity_main.*
import org.koin.android.ext.android.inject

class MainActivity : AppCompatActivity() {
  private val motor: MainMotor by inject()

  override fun onCreate(savedInstanceState: Bundle?) {
    super.onCreate(savedInstanceState)
    setContentView(R.layout.activity_main)

    val adapter = ObservationAdapter()

    observations.layoutManager = LinearLayoutManager(this)
    observations.adapter = adapter
    observations.addItemDecoration(
      DividerItemDecoration(
        this,
        DividerItemDecoration.VERTICAL
      )
    )

    motor.states.observe(this, Observer { state ->
```

```
    when (state) {
      MainViewState.Loading -> progress.visibility = View.VISIBLE
      is MainViewState.Content -> {
        progress.visibility = View.GONE
        adapter.submitList(state.observations)
      }
      is MainViewState.Error -> {
        progress.visibility = View.GONE
        Toast.makeText(
          this@MainActivity, state.throwable.localizedMessage,
          Toast.LENGTH_LONG
        ).show()
        Log.e("Weather", "Exception loading data", state.throwable)
      }
    }
  })

  motor.refresh()
}

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

  return super.onCreateOptionsMenu(menu)
}

override fun onOptionsItemSelected(item: MenuItem): Boolean {
  when (item.itemId) {
    R.id.refresh -> { motor.refresh(); return true }
    R.id.clear -> { motor.clear(); return true }
  }

  return super.onOptionsItemSelected(item)
}

inner class ObservationAdapter :
  ListAdapter<RowState, RowHolder>(RowStateDiffer) {
  override fun onCreateViewHolder(
    parent: ViewGroup,
    viewType: Int
  ) = RowHolder(RowBinding.inflate(layoutInflater, parent, false))

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

class RowHolder(private val binding: RowBinding) :
```

```
    RecyclerView.ViewHolder(binding.root) {

    fun bind(state: RowState) {
      binding.state = state
      binding.executePendingBindings()
    }
  }

  object RowStateDiffer : DiffUtil.ItemCallback<RowState>() {
    override fun areItemsTheSame(
      oldItem: RowState,
      newItem: RowState
    ): Boolean {
      return oldItem === newItem
    }

    override fun areContentsTheSame(
      oldItem: RowState,
      newItem: RowState
    ): Boolean {
      return oldItem == newItem
    }
  }
}
```

The activity gets its motor via Koin (`private val motor: MainMotor by inject()`). The activity observes the `LiveData` and uses that to update the states of the `RecyclerView` and the `ProgressBar`. The `RecyclerView.Adapter` (`ObservationAdapter`) uses `ListAdapter`, so we can just call `submitList()` to supply the new list of `RowState` objects. If we get an `Error` view-state, we log the exception and show a `Toast` with the message.

The activity also sets up the action bar, with refresh and clear options to call `refresh()` and `clear()` on the motor, respectively. We also call `refresh()` at the end of `onCreate()`, to get the initial observation for a fresh install and to make sure that we have the latest observation for any future launches of the activity.

Step #2: Deciding What to Change (and How)

The `Single` response from `ObservationRemoteDataSource`, and the `Completable` responses from `ObservationStore`, are easy to convert into `suspend` functions returning a value (for `Single`) or `Unit` (for `Completable`).

Google has recently upgraded Room to support `Flow` as a return type. We can use

`Flow` to replace our `Observable`.

Once we have made these changes — both to the main app code and to the tests – we can remove RxJava from the project.

Step #3: Adding a Coroutines Dependency

Coroutines are not automatically added to a Kotlin project — you need the `org.jetbrains.kotlinx:kotlinx-coroutines-android` dependency for Android, which in turn will pull in the rest of the coroutines implementation.

In our case, though, we also need to add a dependency to teach Room about coroutines, so it can generate `suspend` functions for relevant DAO functions.

To that end, add this line to the `app` module's `build.gradle` file's `dependencies` closure:

```
    implementation "androidx.room:room-ktx:$room_version"
```

This uses a `room_version` constant that we have set up to synchronize the versions of our various Room dependencies:

```
def room_version = "2.1.0"
```

Step #4: Converting `ObservationRemoteDataSource`

To have Retrofit use a `suspend` function, simply change the function signature of `getCurrentObservation()` in `ObservationApi` from:

```
  fun getCurrentObservation(@Path("stationId") stationId: String): Single<ObservationResponse>
```

to:

```
  suspend fun getCurrentObservation(@Path("stationId") stationId: String): ObservationResponse
```

So now `getCurrentObservation()` directly returns our `ObservationResponse`, with the `suspend` allowing Retrofit to have the network I/O occur on another dispatcher.

This, in turn, will require us to add `suspend` to `getCurrentObservation()` on `ObservationRemoteDataSource`:

```
  suspend fun getCurrentObservation() = api.getCurrentObservation(STATION)
```

Since we are no longer using RxJava with Retrofit, we can remove its call adapter factory from our `Retrofit.Builder` configuration. Delete the `addCallAdapterFactory()` line seen in the original code:

```
  private val retrofit = Retrofit.Builder()
    .client(ok)
    .baseUrl(API_ENDPOINT)
    .addCallAdapterFactory(RxJava2CallAdapterFactory.create())
    .addConverterFactory(MoshiConverterFactory.create())
    .build()
```

...leaving you with:

```
  private val retrofit = Retrofit.Builder()
    .client(ok)
    .baseUrl(API_ENDPOINT)
    .addConverterFactory(MoshiConverterFactory.create())
    .build()
```

You can remove any RxJava imports (e.g., for `Single`).

At this point, `ObservationRemoteDataSource` should look like:

```
package com.commonsware.coroutines.weather

import okhttp3.OkHttpClient
import retrofit2.Retrofit
import retrofit2.converter.moshi.MoshiConverterFactory
import retrofit2.http.GET
import retrofit2.http.Path

private const val STATION = "KDCA"
private const val API_ENDPOINT = "https://api.weather.gov"

class ObservationRemoteDataSource(ok: OkHttpClient) {
  private val retrofit = Retrofit.Builder()
    .client(ok)
    .baseUrl(API_ENDPOINT)
    .addConverterFactory(MoshiConverterFactory.create())
    .build()
  private val api = retrofit.create(ObservationApi::class.java)

  suspend fun getCurrentObservation() = api.getCurrentObservation(STATION)
}

private interface ObservationApi {
  @GET("/stations/{stationId}/observations/current")
  suspend fun getCurrentObservation(@Path("stationId") stationId: String): ObservationResponse
}
```

```
data class ObservationResponse(
  val id: String,
  val properties: ObservationProperties
)

data class ObservationProperties(
  val timestamp: String,
  val icon: String,
  val temperature: ObservationValue,
  val windDirection: ObservationValue,
  val windSpeed: ObservationValue,
  val barometricPressure: ObservationValue
)

data class ObservationValue(
  val value: Double?,
  val unitCode: String
)
```

Step #5: Altering `ObservationDatabase`

Changing our database access is a matter of updating two function declarations on `ObservationStore`, replacing a `Completable` return type with `suspend`. So, change:

```
  @Insert(onConflict = OnConflictStrategy.IGNORE)
  fun save(entity: ObservationEntity): Completable

  @Query("DELETE FROM observations")
  fun clear(): Completable
```

to:

```
  @Insert(onConflict = OnConflictStrategy.IGNORE)
  suspend fun save(entity: ObservationEntity)

  @Query("DELETE FROM observations")
  suspend fun clear()
```

Step #6: Adjusting `ObservationRepository`

Our `ObservationRepository` now needs revised `refresh()` and `clear()` functions, to use the new `suspend` functions from the data sources.

First, add `suspend` to `refresh()` and `clear()` — and remove the `Completable` return type — from the `IObservationRepository` interface that `ObservationRepository` implements:

```
interface IObservationRepository {
  fun load(): Observable<List<ObservationModel>>
  suspend fun refresh()
  suspend fun clear()
}
```

In terms of `ObservationRepository` itself, fixing `clear()` is easy — just take this:

```
  override fun clear() = db.observationStore().clear()
```

...and add the `suspend` keyword, giving you this:

```
  override suspend fun clear() = db.observationStore().clear()
```

`refresh()` is a bit more involved. The current code is:

```
  override fun refresh() = remote.getCurrentObservation()
      .subscribeOn(Schedulers.io())
      .map { convertToEntity(it) }
      .flatMapCompletable { db.observationStore().save(it) }
```

That takes our `Single` response from the repository and applies two RxJava operators to it:

- `map()` to convert the `ObservationResponse` into an `ObservationEntity`, and
- `flatMapCompletable()` to `save()` the entity in Room as part of this RxJava chain

Change that to:

```
  override suspend fun refresh() {
    db.observationStore().save(convertToEntity(remote.getCurrentObservation()))
  }
```

Now, we have a more natural imperative-style syntax, getting our response, converting it to an entity, and passing the entity to `save()`. The `suspend` keyword is required, since we are calling `suspend` functions, as we needed with `refresh()`.

Step #7: Modifying `MainMotor`

The last step is to update `MainMotor` to call the new `suspend` functions from the repository.

However, to do that, we need a `CoroutineScope`. The ideal scope is `viewModelScope`, which is tied to the life of the `ViewModel`. However, for that, we need another dependency. Add this line to the `dependencies` closure of `app/build.gradle`:

```
    implementation "androidx.lifecycle:lifecycle-viewmodel-ktx:2.1.0-rc01"
```

Then, replace the existing `refresh()` and `clear()` functions with:

```
  fun refresh() {
    viewModelScope.launch(Dispatchers.Main) {
      try {
        repo.refresh()
      } catch (t: Throwable) {
        _states.value = MainViewState.Error(t)
      }
    }
  }

  fun clear() {
    viewModelScope.launch(Dispatchers.Main) {
      try {
        repo.clear()
      } catch (t: Throwable) {
        _states.value = MainViewState.Error(t)
      }
    }
  }
```

Exceptions get thrown normally with `suspend` functions, so we can just use ordinary `try/catch` structures to catch them and publish an error state for our UI.

At this point, if you run the app, it should work as it did before, just with somewhat less RxJava.

Step #8: Upgrading to `Flow` Support

Room `2.2.0-alpha02`, released in early August 2019, added support for `Flow` as a return type for `@Dao` functions. Right now, though, the project is using Room `2.1.0`.

So, adjust the value of `room_version` to be `2.2.0-alpha02`:

```
def room_version = "2.2.0-alpha02"
```

Step #9: Amending `ObservationStore` for `Flow`

Now we can change the return type of `load()` to use `Flow` instead of `Observable`:

```
  @Query("SELECT * FROM observations ORDER BY timestamp DESC")
  fun load(): Flow<List<ObservationEntity>>
```

And you can get rid of the RxJava-related `import` statements, both `Observable` and ones lingering from before (e.g., `Completable`).

The resulting `ObservationStore` should look like:

```
@Dao
interface ObservationStore {
  @Query("SELECT * FROM observations ORDER BY timestamp DESC")
  fun load(): Flow<List<ObservationEntity>>

  @Insert(onConflict = OnConflictStrategy.IGNORE)
  suspend fun save(entity: ObservationEntity)

  @Query("DELETE FROM observations")
  suspend fun clear()
}
```

...and `ObservationDatabase` overall should look like:

```
package com.commonsware.coroutines.weather

import android.content.Context
import androidx.annotation.NonNull
import androidx.room.*
import kotlinx.coroutines.flow.Flow

private const val DB_NAME = "weather.db"

@Entity(tableName = "observations")
data class ObservationEntity(
  @PrimaryKey @NonNull val id: String,
  val timestamp: String,
  val icon: String,
  val temperatureCelsius: Double?,
  val windDirectionDegrees: Double?,
  val windSpeedMetersSecond: Double?,
  val barometricPressurePascals: Double?
) {
```

```
  fun toModel() = ObservationModel(
    id = id,
    timestamp = timestamp,
    icon = icon,
    temperatureCelsius = temperatureCelsius,
    windDirectionDegrees = windDirectionDegrees,
    windSpeedMetersSecond = windSpeedMetersSecond,
    barometricPressurePascals = barometricPressurePascals
  )
}

@Dao
interface ObservationStore {
  @Query("SELECT * FROM observations ORDER BY timestamp DESC")
  fun load(): Flow<List<ObservationEntity>>

  @Insert(onConflict = OnConflictStrategy.IGNORE)
  suspend fun save(entity: ObservationEntity)

  @Query("DELETE FROM observations")
  suspend fun clear()
}

@Database(entities = [ObservationEntity::class], version = 1)
abstract class ObservationDatabase : RoomDatabase() {
  abstract fun observationStore(): ObservationStore

  companion object {
    fun create(context: Context) =
      Room.databaseBuilder(context, ObservationDatabase::class.java, DB_NAME)
        .build()
  }
}
```

Step #10: Updating `ObservationRepository` for `Flow`

`ObservationRepository` had been working with an `Observable` from `ObservationStore` – now we need to adjust it to work with a `Flow` instead. Fortunately, this is a very minor change.

Modify the `IObservationRepository` interface to use `Flow` for the return value for `load()`:

```
interface IObservationRepository {
  fun load(): Flow<List<ObservationModel>>
```

```
  suspend fun refresh()
  suspend fun clear()
}
```

Then, on `ObservationRepository` itself, replace the current `load()` function with:

```
  override fun load(): Flow<List<ObservationModel>> = db.observationStore()
    .load()
    .map { entities -> entities.map { it.toModel() }
  }
```

Note that you will need to add an `import` for the `map()` extension function on `Flow`, since even though syntactically it is the same as what we had before, the `map()` implementation is now an extension function:

```
import kotlinx.coroutines.flow.map
```

This just:

- Switches the return type from `Observable` to `Flow`
- Uses the `Flow.map()` extension function instead of the `map()` method on `Observable`

You can remove any RxJava-related `import` statements (e.g., `Observable`, `Single`).

At this point, `ObservationRepository` should look like:

```
package com.commonsware.coroutines.weather

import kotlinx.coroutines.flow.Flow
import kotlinx.coroutines.flow.map

interface IObservationRepository {
  fun load(): Flow<List<ObservationModel>>
  suspend fun refresh()
  suspend fun clear()
}

class ObservationRepository(
  private val db: ObservationDatabase,
  private val remote: ObservationRemoteDataSource
) : IObservationRepository {
  override fun load(): Flow<List<ObservationModel>> = db.observationStore()
    .load()
    .map { entities -> entities.map { it.toModel() }
  }

  override suspend fun refresh() {
    db.observationStore().save(convertToEntity(remote.getCurrentObservation()))
  }
```

```
  override suspend fun clear() = db.observationStore().clear()

  private fun convertToEntity(response: ObservationResponse): ObservationEntity {
    when {
      response.properties.temperature.unitCode != "unit:degC" ->
        throw IllegalStateException(
          "Unexpected temperature unit: ${response.properties.temperature.unitCode}"
        )
      response.properties.windDirection.unitCode != "unit:degree_(angle)" ->
        throw IllegalStateException(
          "Unexpected windDirection unit: ${response.properties.windDirection.unitCode}"
        )
      response.properties.windSpeed.unitCode != "unit:m_s-1" ->
        throw IllegalStateException(
          "Unexpected windSpeed unit: ${response.properties.windSpeed.unitCode}"
        )
      response.properties.barometricPressure.unitCode != "unit:Pa" ->
        throw IllegalStateException(
          "Unexpected barometricPressure unit: ${response.properties.barometricPressure.unitCode}"
        )
    }

    return ObservationEntity(
      id = response.id,
      icon = response.properties.icon,
      timestamp = response.properties.timestamp,
      temperatureCelsius = response.properties.temperature.value,
      windDirectionDegrees = response.properties.windDirection.value,
      windSpeedMetersSecond = response.properties.windSpeed.value,
      barometricPressurePascals = response.properties.barometricPressure.value
    )
  }
}

data class ObservationModel(
  val id: String,
  val timestamp: String,
  val icon: String,
  val temperatureCelsius: Double?,
  val windDirectionDegrees: Double?,
  val windSpeedMetersSecond: Double?,
  val barometricPressurePascals: Double?
)
```

Step #11: Collecting our `Flow` in `MainMotor`

Finally, `MainMotor` now needs to switch from observing an `Observable` to collecting a `Flow`.

Replace the `init` block in `MainMotor` with this:

```
  init {
    viewModelScope.launch(Dispatchers.Main) {
      try {
```

```
        repo.load()
          .map { models ->
            MainViewState.Content(models.map {
              RowState.fromModel(it, formatter, context)
            })
          }
          .collect { _states.value = it }
      } catch (ex: Exception) {
        _states.value = MainViewState.Error(ex)
      }
    }
  }
```

We `launch()` a coroutine that uses `load()` to get the list of `ObservationModel` objects. We then `map()` those into our view-state, then use `collect()` to "subscribe" to the `Flow`, pouring the states into our `MutableLiveData`. If something goes wrong and Room throws an exception, that just gets caught by our `try/catch` construct, so we can update the `MutableLiveData` with an error state.

You can also remove the `sub` property and the `onCleared()` function, as we no longer are using the `CompositeDisposable`. Plus, you can remove all RxJava-related `import` statements, such as the one for `CompositeDisposable` itself.

At this point, `MainMotor` should look like:

```
package com.commonsware.coroutines.weather

import android.content.Context
import androidx.lifecycle.LiveData
import androidx.lifecycle.MutableLiveData
import androidx.lifecycle.ViewModel
import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.flow.collect
import kotlinx.coroutines.flow.map
import kotlinx.coroutines.launch
import org.threeten.bp.OffsetDateTime
import org.threeten.bp.ZoneId
import org.threeten.bp.format.DateTimeFormatter

data class RowState(
  val timestamp: String,
  val icon: String,
  val temp: String?,
  val wind: String?,
  val pressure: String?
```

```
) {
  companion object {
    fun fromModel(
      model: ObservationModel,
      formatter: DateTimeFormatter,
      context: Context
    ): RowState {
      val timestampDateTime = OffsetDateTime.parse(
        model.timestamp,
        DateTimeFormatter.ISO_OFFSET_DATE_TIME
      )
      val easternTimeId = ZoneId.of("America/New_York")
      val formattedTimestamp =
        formatter.format(timestampDateTime.atZoneSameInstant(easternTimeId))

      return RowState(
        timestamp = formattedTimestamp,
        icon = model.icon,
        temp = model.temperatureCelsius?.let {
          context.getString(
            R.string.temp,
            it
          )
        },
        wind = model.windSpeedMetersSecond?.let { speed ->
          model.windDirectionDegrees?.let {
            context.getString(
              R.string.wind,
              speed,
              it
            )
          }
        },
        pressure = model.barometricPressurePascals?.let {
          context.getString(
            R.string.pressure,
            it
          )
        }
      )
    }
  }
}

sealed class MainViewState {
  object Loading : MainViewState()
  data class Content(val observations: List<RowState>) : MainViewState()
  data class Error(val throwable: Throwable) : MainViewState()
```

```
}

class MainMotor(
  private val repo: IObservationRepository,
  private val formatter: DateTimeFormatter,
  private val context: Context
) : ViewModel() {
  private val _states =
    MutableLiveData<MainViewState>().apply { value = MainViewState.Loading }
  val states: LiveData<MainViewState> = _states

  init {
    viewModelScope.launch(Dispatchers.Main) {
      try {
        repo.load()
          .map { models ->
            MainViewState.Content(models.map {
              RowState.fromModel(it, formatter, context)
            })
          }
          .collect { _states.value = it }
      } catch (ex: Exception) {
        _states.value = MainViewState.Error(ex)
      }
    }
  }

  fun refresh() {
    viewModelScope.launch(Dispatchers.Main) {
      try {
        repo.refresh()
      } catch (t: Throwable) {
        _states.value = MainViewState.Error(t)
      }
    }
  }

  fun clear() {
    viewModelScope.launch(Dispatchers.Main) {
      try {
        repo.clear()
      } catch (t: Throwable) {
        _states.value = MainViewState.Error(t)
      }
    }
  }
}
```

And, at this point, if you run the app, it should work as it did before, just without RxJava.

Step #12: Reviewing the Instrumented Tests

So far, we have avoided thinking about tests. This tutorial practices TDD (Test Delayed Development), but, it is time to get our tests working again.

First, let's take a look at our one instrumented test: `MainMotorTest`. As the name suggests, this test exercises `MainMotor`, to confirm that it can take `ObservationModel` objects and emit `RowState` objects as a result.

`MainMotorTest` makes use of two JUnit4 rules:

- `InstantTaskExecutorRule` from the Android Jetpack, for having all normally-asynchronous work related to `LiveData` occur on the current thread
- A custom `AndroidSchedulerRule` that applies an RxJava `TestScheduler` to replace the stock `Scheduler` used for `AndroidSchedulers.mainThread()`:

```
package com.commonsware.coroutines.weather

import io.reactivex.Scheduler
import io.reactivex.android.plugins.RxAndroidPlugins
import org.junit.rules.TestWatcher
import org.junit.runner.Description

class AndroidSchedulerRule<T : Scheduler>(val scheduler: T) : TestWatcher() {
  override fun starting(description: Description?) {
    super.starting(description)

    RxAndroidPlugins.setMainThreadSchedulerHandler { scheduler }
  }

  override fun finished(description: Description?) {
    super.finished(description)

    RxAndroidPlugins.reset()
  }
}
```

`MainMotorTest` also uses Mockito and Mockito-Kotlin to create a mock implementation of our repository, by mocking the `IObservationRepository`

interface that defines the `ObservationRepository` API:

```
@RunWith(AndroidJUnit4::class)
class MainMotorTest {
  @get:Rule
  val instantTaskExecutorRule = InstantTaskExecutorRule()
  @get:Rule
  val androidSchedulerRule = AndroidSchedulerRule(TestScheduler())

  private val repo: IObservationRepository = mock()
```

The `initialLoad()` test function confirms that our `MainMotor` handles a normal startup correctly. Specifically, we validate that our initial state is `MainViewState.Loading`, then proceeds to `MainViewState.Content`:

```
  @Test
  fun initialLoad() {
    whenever(repo.load()).thenReturn(Observable.just(listOf(TEST_MODEL)))

    val underTest = makeTestMotor()
    val initialState = underTest.states.value

    assertThat(initialState is MainViewState.Loading, equalTo(true))
    androidSchedulerRule.scheduler.triggerActions()

    val state = underTest.states.value as MainViewState.Content

    assertThat(state.observations.size, equalTo(1))
    assertThat(state.observations[0], equalTo(TEST_ROW_STATE))
  }
```

Of particular note:

- We use `Observable.just()` to mock the response from `load()` on the repository
- We use the `TestScheduler` that we set up using `AndroidSchedulerRule` to advance from the loading state to the content state

The `initialLoadError()` test function demonstrates tests if `load()` throws some sort of exception, to confirm that we wind up with a `MainViewState.Error` result:

```
  @Test
  fun initialLoadError() {
    whenever(repo.load()).thenReturn(Observable.error(TEST_ERROR))
```

```
    val underTest = makeTestMotor()
    val initialState = underTest.states.value

    assertThat(initialState is MainViewState.Loading, equalTo(true))
    androidSchedulerRule.scheduler.triggerActions()

    val state = underTest.states.value as MainViewState.Error

    assertThat(TEST_ERROR, equalTo(state.throwable))
  }
```

In this case, we use `Observable.error()` to set up an `Observable` that immediately throws a test error.

The `refresh()` test function confirms that we can call `refresh()` without crashing and that `MainMotor` can handle a fresh set of model objects delivered to it by our `Observable`. We simulate our ongoing Room-backed `Observable` by a `PublishSubject`, so we can provide initial data and can later supply "fresh" data, simulating the results of Room detecting our refreshed database content:

```
  @Test
  fun refresh() {
    val testSubject: PublishSubject<List<ObservationModel>> =
      PublishSubject.create()

    whenever(repo.load()).thenReturn(testSubject)
    whenever(repo.refresh()).thenReturn(Completable.complete())

    val underTest = makeTestMotor()
    val initialState = underTest.states.value

    assertThat(initialState is MainViewState.Loading, equalTo(true))

    testSubject.onNext(listOf(TEST_MODEL))
    androidSchedulerRule.scheduler.triggerActions()

    underTest.refresh()

    testSubject.onNext(listOf(TEST_MODEL, TEST_MODEL_2))
    androidSchedulerRule.scheduler.triggerActions()

    val state = underTest.states.value as MainViewState.Content

    assertThat(state.observations.size, equalTo(2))
    assertThat(state.observations[0], equalTo(TEST_ROW_STATE))
    assertThat(state.observations[1], equalTo(TEST_ROW_STATE_2))
```

```
}
```

Note that we use `Completable.complete()` as the mocked return value from `refresh()` on our repository, so `MainMotor` gets a valid `Completable` for use.

Finally, the `clear()` test function confirms that we can call `clear()` without crashing and that `MainMotor` can handle an empty set of model objects delivered to it by our `Observable`:

```
@Test
fun clear() {
  val testSubject: PublishSubject<List<ObservationModel>> =
    PublishSubject.create()

  whenever(repo.load()).thenReturn(testSubject)
  whenever(repo.clear()).thenReturn(Completable.complete())

  val underTest = makeTestMotor()
  val initialState = underTest.states.value

  assertThat(initialState is MainViewState.Loading, equalTo(true))

  testSubject.onNext(listOf(TEST_MODEL))
  androidSchedulerRule.scheduler.triggerActions()

  underTest.clear()

  testSubject.onNext(listOf())
  androidSchedulerRule.scheduler.triggerActions()

  val state = underTest.states.value as MainViewState.Content

  assertThat(state.observations.size, equalTo(0))
}
```

This is not a complete set of tests for `MainMotor`, but it is enough to give us the spirit of testing the RxJava/Jetpack combination.

Step #13: Repair `MainMotorTest`

Of course, `MainMotorTest` is littered with compile errors at the moment, as we changed `MainMotor` and `ObservationRepository` to use coroutines instead of RxJava. So, we need to make some repairs.

First, add this dependency to `app/build.gradle`:

```
    androidTestImplementation 'org.jetbrains.kotlinx:kotlinx-coroutines-test:1.3.0-RC'
```

This is a release candidate of the `kotlinx-coroutines-test` dependency. It gives us a `TestCoroutineDispatcher` that we can use in a similar fashion as to how we used `TestScheduler` with RxJava.

Next, add this `MainDispatcherRule` alongside `MainMotorTest` in the `androidTest` source set:

```
package com.commonsware.coroutines.weather

import kotlinx.coroutines.Dispatchers
import kotlinx.coroutines.ExperimentalCoroutinesApi
import kotlinx.coroutines.test.TestCoroutineDispatcher
import kotlinx.coroutines.test.resetMain
import kotlinx.coroutines.test.setMain
import org.junit.rules.TestWatcher
import org.junit.runner.Description

// inspired by https://medium.com/androiddevelopers/easy-coroutines-in-android-viewmodelscope-25bffb605471

@ExperimentalCoroutinesApi
class MainDispatcherRule(paused: Boolean) : TestWatcher() {
  val dispatcher =
    TestCoroutineDispatcher().apply { if (paused) pauseDispatcher() }

  override fun starting(description: Description?) {
    super.starting(description)

    Dispatchers.setMain(dispatcher)
  }

  override fun finished(description: Description?) {
    super.finished(description)

    Dispatchers.resetMain()
    dispatcher.cleanupTestCoroutines()
  }
}
```

This is another JUnit rule, implemented as a `TestWatcher`, allowing us to encapuslate some setup and teardown work. Specifically, we create a `TestCoroutineDispatcher` and possibly configure it to be paused, so we have to manually trigger coroutines to be executed. Then, when a test using the rule starts, we use `Dispatchers.setMain()` to override the default `Dispatchers.Main` dispatcher. When a test using the rule ends, we use `Dispatchers.resetMain()` to return to normal operation, plus clean up our `TestCoroutineDispatcher` for any lingering coroutines that are still outstanding.

Then, in `MainMotorTest`, replace these `AndroidSchedulerRule` lines:

```
  @get:Rule
  val androidSchedulerRule = AndroidSchedulerRule(TestScheduler())
```

...with these lines to set up the `MainDispatcherRule`:

```
  @get:Rule
  val mainDispatcherRule = MainDispatcherRule(paused = true)
```

At this point, we are no longer using `AndroidSchedulerRule`, so you can delete that class.

However, our use of `MainDispatcherRule` has a warning, indicating that this is an experimental API. So, add `@ExperimentalCoroutinesApi` to the declaration of `MainMotorTest`:

```
@ExperimentalCoroutinesApi
@RunWith(AndroidJUnit4::class)
class MainMotorTest {
```

Next, in the `initialLoad()` function, change:

```
    whenever(repo.load()).thenReturn(Observable.just(listOf(TEST_MODEL)))
```

...to:

```
    whenever(repo.load()).thenReturn(flowOf(listOf(TEST_MODEL)))
```

Here, we use `flowOf()`, which is the `Flow` equivalent of `Observable.just()`. `flowOf()` creates a `Flow` that emits the object(s) that we provide — here we are just providing one, but `flowOf()` accepts an arbitrary number of objects.

Then, replace:

```
    androidSchedulerRule.scheduler.triggerActions()
```

...with:

```
    mainDispatcherRule.dispatcher.runCurrent()
```

This replaces our manual trigger of the `TestScheduler` with an equivalent manual trigger of the `TestCoroutineDispatcher`.

At this point, `initialLoad()` should have no more compile errors and should look like:

```
@Test
fun initialLoad() {
  whenever(repo.load()).thenReturn(flowOf(listOf(TEST_MODEL)))

  val underTest = makeTestMotor()
  val initialState = underTest.states.value

  assertThat(initialState is MainViewState.Loading, equalTo(true))
  mainDispatcherRule.dispatcher.runCurrent()

  val state = underTest.states.value as MainViewState.Content

  assertThat(state.observations.size, equalTo(1))
  assertThat(state.observations[0], equalTo(TEST_ROW_STATE))
}
```

Now, we need to fix `initialLoadError()`. Change:

```
  whenever(repo.load()).thenReturn(Observable.error(TEST_ERROR))
```

...to:

```
  whenever(repo.load()).thenReturn(flow { throw TEST_ERROR })
```

`Observable.error()` creates an `Observable` that throws the supplied exception when it is subscribed to. The equivalent is to use the `flow()` creator function and simply throw the exception yourself — that will occur when something starts consuming the flow.

Then, replace:

```
  androidSchedulerRule.scheduler.triggerActions()
```

...with:

```
  mainDispatcherRule.dispatcher.runCurrent()
```

As before, this replaces our manual trigger of the `TestScheduler` with an equivalent manual trigger of the `TestCoroutineDispatcher`.

At this point, `initialLoadError()` should have no more compile errors and should

look like:

```
@Test
fun initialLoadError() {
  whenever(repo.load()).thenReturn(flow { throw TEST_ERROR })

  val underTest = makeTestMotor()
  val initialState = underTest.states.value

  assertThat(initialState is MainViewState.Loading, equalTo(true))
  mainDispatcherRule.dispatcher.runCurrent()

  val state = underTest.states.value as MainViewState.Error

  assertThat(TEST_ERROR, equalTo(state.throwable))
}
```

Next up is the `refresh()` test function. Replace:

```
  val testSubject: PublishSubject<List<ObservationModel>> =
    PublishSubject.create()

  whenever(repo.load()).thenReturn(testSubject)
  whenever(repo.refresh()).thenReturn(Completable.complete())
```

...with:

```
  val channel = Channel<List<ObservationModel>>()

  whenever(repo.load()).thenReturn(channel.consumeAsFlow())
```

As with RxJava, we need to set up a `Flow` where we can deliver multiple values over time. One way to do that is to set up a `Channel`, then use `consumeAsFlow()` to convert it into a `Flow`. We can then use the `Channel` to `offer()` objects to be emitted by the `Flow`. So, we switch our mock `repo.load()` call to use this approach, and we get rid of the `repo.refresh()` mock, as `refresh()` no longer returns anything, so stock Mockito behavior is all that we need.

You will see that `consumeAsFlow()` has a warning. This is part of a `Flow` preview API and has its own warning separate from the one that we fixed by adding `@ExperimentalCoroutinesApi` to the class. To fix this one, add `@FlowPreview` to the class:

```
@FlowPreview
@ExperimentalCoroutinesApi
```

```
@RunWith(AndroidJUnit4::class)
class MainMotorTest {
```

Next, replace:

```
    testSubject.onNext(listOf(TEST_MODEL))
    androidSchedulerRule.scheduler.triggerActions()
```

...with:

```
    channel.offer(listOf(TEST_MODEL))
    mainDispatcherRule.dispatcher.runCurrent()
```

Now we `offer()` our model data to the `Channel` to be emitted by the `Flow`, and we advance our `TestCoroutineDispatcher` to process that event.

Then, replace:

```
    testSubject.onNext(listOf(TEST_MODEL, TEST_MODEL_2))
    androidSchedulerRule.scheduler.triggerActions()
```

...with:

```
    channel.offer(listOf(TEST_MODEL, TEST_MODEL_2))
    mainDispatcherRule.dispatcher.runCurrent()
```

This is the same basic change as the previous one, just for the second set of model objects.

At this point, `refresh()` should have no compile errors and should look like:

```
  @Test
  fun refresh() {
    val channel = Channel<List<ObservationModel>>()

    whenever(repo.load()).thenReturn(channel.consumeAsFlow())

    val underTest = makeTestMotor()
    val initialState = underTest.states.value

    assertThat(initialState is MainViewState.Loading, equalTo(true))

    channel.offer(listOf(TEST_MODEL))
    mainDispatcherRule.dispatcher.runCurrent()
```

```
    underTest.refresh()

    channel.offer(listOf(TEST_MODEL, TEST_MODEL_2))
    mainDispatcherRule.dispatcher.runCurrent()

    val state = underTest.states.value as MainViewState.Content

    assertThat(state.observations.size, equalTo(2))
    assertThat(state.observations[0], equalTo(TEST_ROW_STATE))
    assertThat(state.observations[1], equalTo(TEST_ROW_STATE_2))
  }
```

Finally, we need to fix the `clear()` test function. This will use the same techniques that we used for the `refresh()` test function.

Replace:

```
    val testSubject: PublishSubject<List<ObservationModel>> =
      PublishSubject.create()

    whenever(repo.load()).thenReturn(testSubject)
    whenever(repo.clear()).thenReturn(Completable.complete())
```

...with:

```
    val channel = Channel<List<ObservationModel>>()

    whenever(repo.load()).thenReturn(channel.consumeAsFlow())
```

Next, replace:

```
    testSubject.onNext(listOf(TEST_MODEL))
    androidSchedulerRule.scheduler.triggerActions()
```

...with:

```
    channel.offer(listOf(TEST_MODEL))
    mainDispatcherRule.dispatcher.runCurrent()
```

Then, replace:

```
    testSubject.onNext(listOf())
    androidSchedulerRule.scheduler.triggerActions()
```

...with:

```
    channel.offer(listOf())
    mainDispatcherRule.dispatcher.runCurrent()
```

At this point, `clear()` should have no compile errors and should look like:

```
@Test
fun clear() {
  val channel = Channel<List<ObservationModel>>()

  whenever(repo.load()).thenReturn(channel.consumeAsFlow())

  val underTest = makeTestMotor()
  val initialState = underTest.states.value

  assertThat(initialState is MainViewState.Loading, equalTo(true))

  channel.offer(listOf(TEST_MODEL))
  mainDispatcherRule.dispatcher.runCurrent()

  underTest.clear()

  channel.offer(listOf())
  mainDispatcherRule.dispatcher.runCurrent()

  val state = underTest.states.value as MainViewState.Content

  assertThat(state.observations.size, equalTo(0))
}
```

And, most importantly, if you run the tests in `MainMotorTest`, they should all pass.

At this point, you can remove all `import` statements related to RxJava, such as `Observable`.

Step #14: Remove RxJava

At this point, we no longer need RxJava in the project. So, you can remove the following lines from the `dependencies` closure of `app/build.gradle`:

```
    implementation "androidx.room:room-rxjava2:$room_version"
```

```
    implementation "io.reactivex.rxjava2:rxandroid:2.1.1"
    implementation 'io.reactivex.rxjava2:rxkotlin:2.3.0'
```

```
    implementation "com.squareup.retrofit2:adapter-rxjava2:$retrofit_version"
```

```
    implementation 'nl.littlerobots.rxlint:rxlint:1.7.4'
```

After this change — and the obligatory "sync Gradle with project files" step – both the app and the tests should still work.

And you're done!