Provided TypeConverters for Room

Author: Michał Zieliński (yairobbe@gmail.com)
Issue tracker link: https://issuetracker.google.com/u/0/issues/121067210
Created: August 2020

Summary

Current version of Room allows users to create Type Converters. The limitation of this system is that such converter must define at least one method annotated with `@TypeConverter` annotation and it must meet one of conditions:

- be a static method
- be a method inside a Kotlin object
- have a no-argument public constructor

This is because Room needs to be able to create an instance of the converter (if needed) and invoke converter methods on this object.

The problem with this approach is that users of Room library are not able to pass additional dependencies to custom converters. Author of the original feature request provides an example where they need to use the json serialization library inside a type converter.

A solution to that problem is to allow users to add `@ProvidedTypeConverter` annotation on a Type Converter which will tell Room that it shouldn't instantiate such a converter. It'll be added at runtime during database creation. This document describes API proposal.

API proposal

This section describes how the public API for this feature could look like.

1. Introduce a new `@ProvidedTypeConverter` annotation:

```
@Target(ElementType.TYPE)
@Retention(RetentionPolicy.CLASS)
public @interface ProvidedTypeConverter {
}
```

This annotation will be used to annotate classes containing `@TypeConverter` annotated methods.

As described in the Summary section, current version of Room performs a couple of checks to determine if it'll be able to invoke Type Converter methods. More specifically - it checks at compile time if it'll be able to create an object of a class containing `@TypeConverter` annotated methods (if needed, because it's possible that it's a static class or a Kotlin object). It's a nice mechanism because it informs users about errors at compile time. Introduction of Provided Type Converters changes that because when a `@ProvidedTypeConverter` annotation is present, it should be always used, even if a class that needs to be instantiated contains a no-argument public constructor. Room doesn't know if the converter is present or not at compile time. This means it's losing some of the functionality.
With `@ProvidedTypeConverter` annotation Room can be informed at compile time that it can expect that there will be a Type Converter available at runtime. There are four possible scenarios:

	Type Converter present at runtime	Type Converter not present at runtime
`@ProvidedTypeConverter` present	Happy path - Room can use the converter.	Room expected to have a converter but it wasn't added at runtime. Room can throw an exception.
`@ProvidedTypeConverter` not present	Room will throw an exception saying that `@ProvidedTypeConverter` annotation is missing.	Room will try to instantiate a class containing Type Converters using a no-arg constructor if present or throw an exception.

2. Introduce a new `addTypeConverter` builder method:

```
@NonNull
 public Builder<T> addTypeConverter(
         @NonNull Object typeConverter) {}
```

   parameter:

   - typeConverter parameter is an instance of a Type Converter class annotated with `@ProvidedTypeConverter` annotation

   Developers will be able to pass one or more `TypeConverters` to Room using the familiar builder API. This gives them control over creation of a Type Converter and that means that they'll be able to pass any dependency they want.

Example API usage

This section shows an example usage of the new API.

```
@ProvidedTypeConverter
public class ExampleConverter {

    private Foo dependency;

    public ExampleConverter(Foo dependency) {
        this.dependency = dependency;
    }

    @TypeConverter
    public Bar fromString(String value) {
        return dependency.toBar(value);
    }

    @TypeConverter
    public String barToString(Bar bar) {
        return dependency.toString(bar);
    }
}

// instantiate or inject ExampleConverter instance and
// pass it to RoomDatabase builder
db = Room.databaseBuilder(...)
        .addTypeConverter(exampleConverterInstance)
        .build();
```

Dependency injection

This section shows examples of using this feature with dependency injection libraries.

Dagger/Hilt

```
@Module
public class GsonModule {

    @Provides
    @Singleton
    Gson provideGson() {
        return new GsonBuilder().create();
    }
}

@ProvidedTypeConverter
public class JsonConverter {

    private Gson gson;

    @Inject
    public JsonConverter(Gson gson) {
        this.gson = gson;
    }

    @TypeConverter
    public Bar fromJson(String jsonString) {
        return gson.fromJson(jsonString, Bar.class);
    }

    @TypeConverter
    public String barToString(Bar bar) {
        return gson.toJson(bar);
    }
}

// inject JsonConverter instance and pass it to RoomDatabase builder
db = Room.databaseBuilder(...)
        .addTypeConverter(jsonConverterInstance)
        .build();
```

Koin

```
@ProvidedTypeConverter
class JsonConverter(private val gson: Gson) {

    @TypeConverter
    fun fromJson(jsonString: String): Bar {
        return gson.fromJson(jsonString, Bar::class.java)
    }

    @TypeConverter
    fun barToString(bar: Bar): String {
        return gson.toJson(bar)
    }
}

val fooModule = module {
    single<Gson> { GsonBuilder().create() }
    single<JsonConverter> { JsonConverter(gson = get()) }
}
```