

Choose your use case / Share data access layer

Create a multiplatform app using Ktor and SQLDelight

IntelliJ IDEA

Android Studio



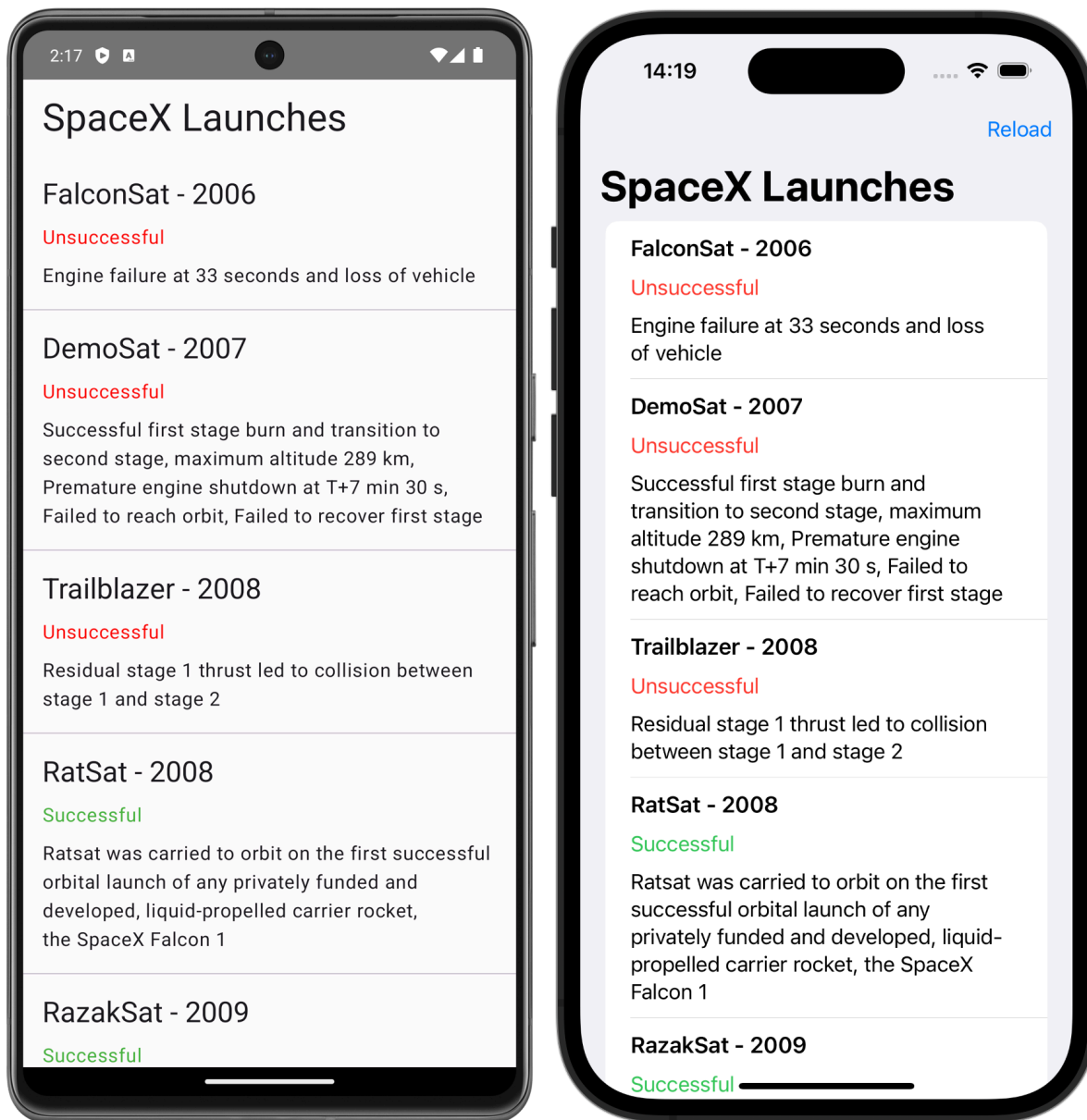
[Edit page](#) 31 October 2025

This tutorial uses IntelliJ IDEA, but you can also follow it in Android Studio – both IDEs share the same core functionality and Kotlin Multiplatform support.

This tutorial demonstrates how to use IntelliJ IDEA to create an advanced mobile application for iOS and Android using Kotlin Multiplatform. This application is going to:

- Retrieve data over the internet from the public [SpaceX API ↗](#) using Ktor
- Save the data in a local database using SQLDelight.
- Display a list of SpaceX rocket launches together with the launch date, results, and a detailed description of the launch.

The application will include a module with shared code for both the iOS and Android platforms. The business logic and data access layers will be implemented only once in the shared module, while the UI of both applications will be native.



You will use the following multiplatform libraries in the project:

- [Ktor](#) as an HTTP client for retrieving data over the internet.
- [kotlinx.serialization](#) to deserialize JSON responses into objects of entity classes.
- [kotlinx.coroutines](#) to write asynchronous code.

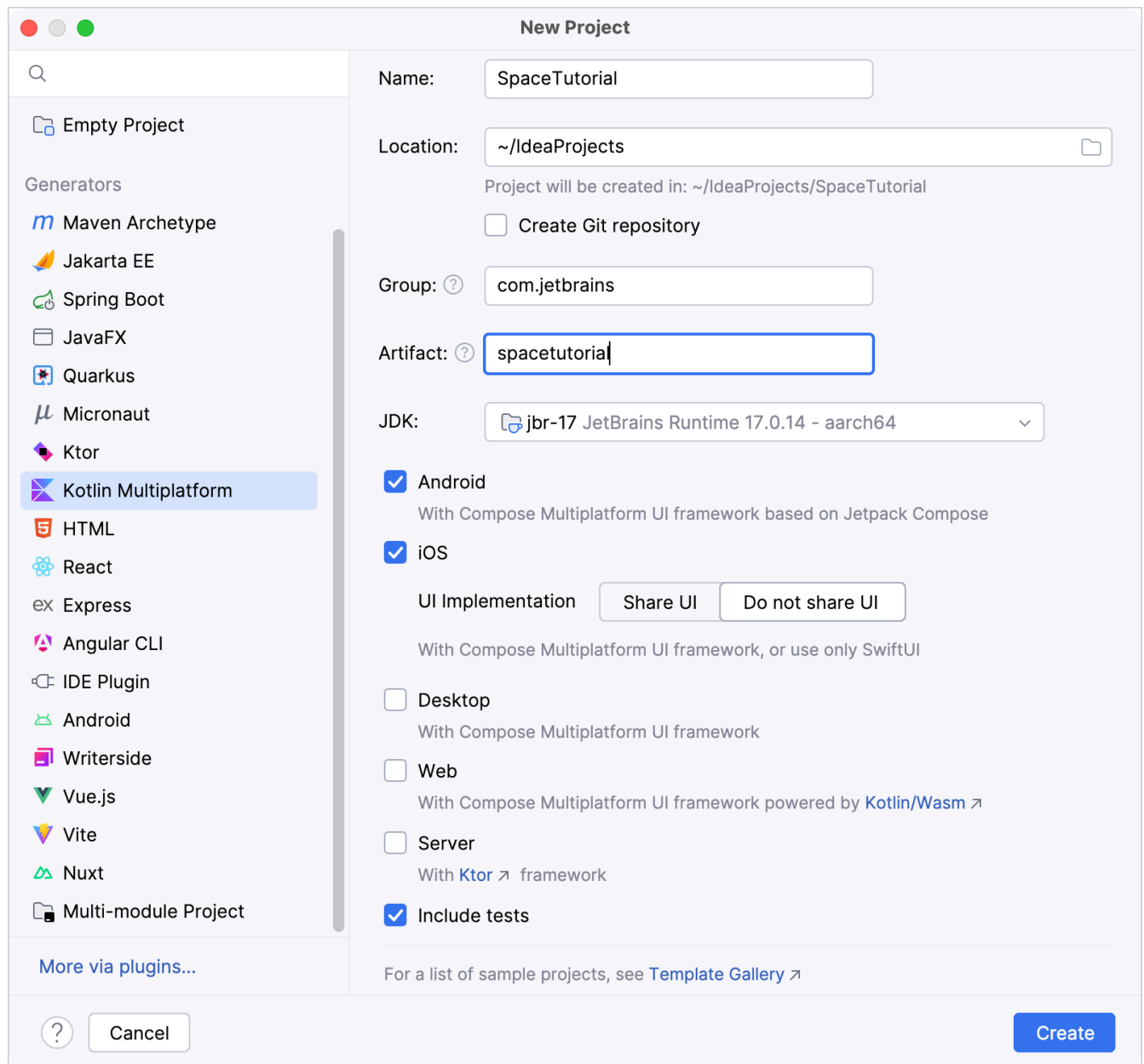
- [SQLDelight](#) ↗ to generate Kotlin code from SQL queries and create a type-safe database API.
- [Koin](#) ↗ to provide platform-specific database drivers via dependency injection.



You can find the [template project](#) ↗ as well as the source code of the [final application](#) ↗ in our GitHub repository.

Create a project

1. In the [quickstart](#), complete the instructions to [set up your environment for Kotlin Multiplatform development](#).
2. In IntelliJ IDEA, select **File | New | Project**.
3. In the panel on the left, select **Kotlin Multiplatform** (in Android Studio, the template can be found in the **Generic** tab of the **New Project** wizard).
4. Specify the following fields in the **New Project** window:
 - **Name:** SpaceTutorial
 - **Group:** com.jetbrains
 - **Artifact:** spacetutorial



5. Select **Android** and **iOS** targets.
6. For iOS, select the **Do not share UI** option. You will implement a native UI for both platforms.
7. Once you've specified all the fields and targets, click **Create**.

Add Gradle dependencies

To add a multiplatform library to the shared module, you need to add dependency instructions (`implementation`) to the `dependencies {}` block of the relevant source sets in the `build.gradle.kts` file.

Both the `kotlinx.serialization` and SQLDelight libraries also require additional configuration.

Change or add lines in the version catalog in the `gradle/libs.versions.toml` file to reflect all needed dependencies:

1. In the `[versions]` block, check the AGP version and add the rest:

```
✕ [versions]
agp = "8.7.3"
...
coroutinesVersion = "1.10.2"
dateTimeVersion = "0.7.1"
koin = "4.1.0"
ktor = "3.3.3"
sqlDelight = "2.1.0"
material3 = "1.3.2"
```


2. In the `[libraries]` block, add the following library references:

```
✕ [libraries]
...
android-driver = { module = "app.cash.sqldelight:android-driver", ver
koin-androidx-compose = { module = "io.insert-koin:koin-androidx-comp
koin-core = { module = "io.insert-koin:koin-core", version.ref = "koi
kotlinx-coroutines-core = { module = "org.jetbrains.kotlinx:kotlinx-c
kotlinx-datetime = { module = "org.jetbrains.kotlinx:kotlinx-datetime
ktor-client-android = { module = "io.ktor:ktor-client-android", versi
ktor-client-content-negotiation = { module = "io.ktor:ktor-client-con
ktor-client-core = { module = "io.ktor:ktor-client-core", version.ref
```

```
ktor-client-darwin = { module = "io.ktor:ktor-client-darwin", version
ktor-serialization-kotlinx-json = { module = "io.ktor:ktor-serializat
native-driver = { module = "app.cash.sqldelight:native-driver", versi
runtime = { module = "app.cash.sqldelight:runtime", version.ref = "sq
androidx-compose-material3 = { module = "androidx.compose.material3:r
```

3. In the `[plugins]` block, specify the necessary Gradle plugins:

```
[plugins]
...
kotlinxSerialization = { id = "org.jetbrains.kotlin.plugin.serialization"
sqlDelight = { id = "app.cash.sqldelight", version.ref = "sqlDelight"
```

4. Once the dependencies are added, you're prompted to resync the project. Click the **Sync Gradle Changes** button to synchronize Gradle files: 

5. At the very beginning of the `shared/build.gradle.kts` file, add the following lines to the `plugins {}` block:

```
plugins {
    // ...
    alias(libs.plugins.kotlinxSerialization)
    alias(libs.plugins.sqlDelight)
}
```

6. The common source set requires a core artifact of each library, as well as the Ktor serialization feature [↗](#) to use `kotlinx.serialization` for processing network requests and responses. The iOS and Android source sets also need SQLDelight and Ktor platform drivers.

In the same `shared/build.gradle.kts` file, add all the required dependencies:

```
kotlin {
    // ...
```

```

sourceSets {
    commonMain.dependencies {
        implementation(libs.kotlinx.coroutines.core)
        implementation(libs.ktor.client.core)
        implementation(libs.ktor.client.content.negotiation)
        implementation(libs.ktor.serialization.kotlinx.json)
        implementation(libs.runtime)
        implementation(libs.kotlinx.datetime)
        implementation(libs.koin.core)
    }
    androidMain.dependencies {
        implementation(libs.ktor.client.android)
        implementation(libs.android.driver)
    }
    iosMain.dependencies {
        implementation(libs.ktor.client.darwin)
        implementation(libs.native.driver)
    }
}
}

```

7. At the beginning of the `sourceSets` block, opt in for the experimental time API of the standard Kotlin library:

```

kotlin {
    // ...

    sourceSets {
        all {
            languageSettings.optIn("kotlin.time.ExperimentalTime")
        }
    }

    // ...
}
}

```

8. Once the dependencies are added, click the **Sync Gradle Changes** button to synchronize Gradle files once again.

After the Gradle sync, you are done with the project configuration and can start writing code.



For an in-depth guide on multiplatform dependencies, see [Dependency on Kotlin Multiplatform libraries](#).

Create an application data model

The tutorial app will contain the public `SpaceXSDK` class as the facade over networking and cache services. The application data model will have three entity classes with:

- General information about the launch
- Links to images of mission patches
- URLs of articles related to the launch



Not all of this data will end up in the UI by the end of this tutorial. We're using the data model to showcase serialization. But you can play around with links and patches to extend the example into something more informative!

Create the necessary data classes:

1. In the `shared/src/commonMain/kotlin/com/jetbrains/spacetutorial` directory, create the `entity` package, then create the `Entity.kt` file inside that package.
2. Declare all the data classes for basic entities:


```

✕ import kotlinx.datetime.TimeZone
import kotlinx.datetime.toInstant
import kotlinx.datetime.toLocalDateTime
import kotlinx.serialization.SerialName
import kotlinx.serialization.Serializable

@Serializable
data class RocketLaunch(
    @SerialName("flight_number")
    val flightNumber: Int,
    @SerialName("name")
    val missionName: String,
    @SerialName("date_utc")
    val launchDateUTC: String,
    @SerialName("details")
    val details: String?,
    @SerialName("success")
    val launchSuccess: Boolean?,
    @SerialName("links")
    val links: Links
) {
    var launchYear = Instant.parse(launchDateUTC).toLocalDateTime(Tir
}

@Serializable
data class Links(
    @SerialName("patch")
    val patch: Patch?,
    @SerialName("article")
    val article: String?
)

@Serializable
data class Patch(
    @SerialName("small")
    val small: String?,
    @SerialName("large")

```

```
    val large: String?  
)
```

Each serializable class must be marked with the `@Serializable` annotation. The `kotlinx.serialization` plugin automatically generates a default serializer for `@Serializable` classes unless you explicitly pass a link to a serializer in the annotation argument.

The `@SerialName` annotation allows you to redefine field names, which helps to access properties in data classes using more readable identifiers.

Configure SQLDelight and implement cache logic

Configure SQLDelight

The SQLDelight library allows you to generate a type-safe Kotlin database API from SQL queries. During compilation, the generator validates the SQL queries and turns them into Kotlin code that can be used in the shared module.

The SQLDelight dependency is already included in the project. To configure the library, open the `shared/build.gradle.kts` file and add the `sqlDelight {}` block in the end. This block contains a list of databases and their parameters:

```

sqlDelight {
    databases {
        create("AppDatabase") {
            packageName.set("com.jetbrains.spacetutorial.cache")
        }
    }
}

```

The `packageName` parameter specifies the package name for the generated Kotlin sources.

Sync the Gradle project files when prompted, or press double `Shift` and search for the **Sync All Gradle, Swift Package Manager projects**.



Consider installing the official [SQLDelight plugin](#) to work with `.sq` files.

Generate the database API

First, create the `.sq` file with all the necessary SQL queries. By default, the SQLDelight plugin looks for `.sq` files in the `sqlDelight` folder of the source set:

1. In the `shared/src/commonMain` directory, create a new `sqlDelight` directory.
2. Inside the `sqlDelight` directory, create a new directory with the name `com/jetbrains/spacetutorial/cache` to create nested directories for the package.
3. Inside the `cache` directory, create the `AppDatabase.sq` file (with the same name as the database you specified in the `build.gradle.kts` file). All the SQL queries for your application will be stored in this file.
4. The database will contain a table with data about launches. Add the following code for creating the table to the `AppDatabase.sq` file:

```
import kotlin.Boolean;

CREATE TABLE Launch (
    flightNumber INTEGER NOT NULL,
    missionName TEXT NOT NULL,
    details TEXT,
    launchSuccess INTEGER AS Boolean DEFAULT NULL,
    launchDateUTC TEXT NOT NULL,
    patchUrlSmall TEXT,
    patchUrlLarge TEXT,
    articleUrl TEXT
);
```

5. Add the `insertLaunch` function for inserting data into the table:

```
insertLaunch:
INSERT INTO Launch(flightNumber, missionName, details, launchSuccess,
VALUES(?, ?, ?, ?, ?, ?, ?, ?);
```

6. Add the `removeAllLaunches` function for clearing data in the table:

```
removeAllLaunches:
DELETE FROM Launch;
```

7. Declare the `selectAllLaunchesInfo` function for retrieving data:

```
selectAllLaunchesInfo:
SELECT Launch.*
FROM Launch;
```

8. Generate the corresponding `AppDatabase` interface (which you will initialize with database drivers later on). To do that, run the following command in the terminal:

```
./gradlew generateCommonMainAppDatabaseInterface
```

The generated Kotlin code is stored in the `shared/build/generated/sqldelight` directory.

Create factories for platform-specific database drivers

To initialize the `AppDatabase` interface, you will pass an `SqlDriver` instance to it. SQLDelight provides multiple platform-specific implementations of the SQLite driver, so you need to create these instances separately for each platform.

While you can achieve this with expected and actual interfaces, in this project, you will use Koin to try dependency injection in Kotlin Multiplatform.

1. Create an interface for database drivers. To do this, in the `shared/src/commonMain/kotlin/com/jetbrains/spacetutorial/` directory, create the `cache` package.
2. Create the `DatabaseDriverFactory` interface inside the `cache` package:

```
package com.jetbrains.spacetutorial.cache

import app.cash.sqldelight.db.SqlDriver

interface DatabaseDriverFactory {
    fun createDriver(): SqlDriver
}
```

3. Create the class implementing this interface for Android: in the `shared/src/androidMain/kotlin` directory, create the `com.jetbrains.spacetutorial.cache` package, then create the `DatabaseDriverFactory.kt` file inside it.

4. On Android, the SQLite driver is implemented by the `AndroidSqliteDatabaseDriver` class. In the `DatabaseDriverFactory.kt` file, pass the database information and the context link to the `AndroidSqliteDatabaseDriver` class constructor:

```
package com.jetbrains.spacetutorial.cache

import android.content.Context
import app.cash.sqldelight.db.SqlDriver
import app.cash.sqldelight.driver.android.AndroidSqliteDatabaseDriver

class AndroidDatabaseDriverFactory(private val context: Context) : DatabaseDriverFactory {
    override fun createDriver(): SqlDriver {
        return AndroidSqliteDatabaseDriver(AppDatabase.Schema, context, "launch.db")
    }
}
```

5. For iOS, in the `shared/src/iosMain/kotlin/com/jetbrains/spacetutorial/` directory, create the `cache` package.
6. Inside the `cache` package, create the `DatabaseDriverFactory.kt` file and add this code:

```
package com.jetbrains.spacetutorial.cache

import app.cash.sqldelight.db.SqlDriver
import app.cash.sqldelight.driver.native.NativeSqliteDatabaseDriver

class IOSDatabaseDriverFactory : DatabaseDriverFactory {
    override fun createDriver(): SqlDriver {
        return NativeSqliteDatabaseDriver(AppDatabase.Schema, "launch.db")
    }
}
```

You will implement instances of these drivers later in the platform-specific code of your project.

Implement cache

So far, you have added factories for platform database drivers and an `AppDatabase` interface to perform database operations. Now, create a `Database` class, which will wrap the `AppDatabase` interface and contain the caching logic.

1. In the common source set `shared/src/commonMain/kotlin`, create a new `Database` class in the `com.jetbrains.spacetutorial.cache` package. It will contain logic common to both platforms.
2. To provide a driver for `AppDatabase`, pass an abstract `DatabaseDriverFactory` instance to the `Database` class constructor:

```
package com.jetbrains.spacetutorial.cache

internal class Database(databaseDriverFactory: DatabaseDriverFactory) {
    private val database = AppDatabase(databaseDriverFactory.createDr:
    private val dbQuery = database.appDatabaseQueries
}
```

This class's visibility `internal` is set to internal, which means it is only accessible from within the multiplatform module.

3. Inside the `Database` class, implement some data handling operations. First, create the `getAllLaunches` function to return a list of all the rocket launches. The `mapLaunchSelecting` function is used to map the result of the database query to `RocketLaunch` objects:

```
+ internal fun getAllLaunches() {...}
```

4. Add the `clearAndCreateLaunches` function to clear the database and insert new data:

```
internal class Database(databaseDriverFactory: DatabaseDriverFactory) {
    // ...

    internal fun clearAndCreateLaunches(launches: List<RocketLaunch>)
```

```

dbQuery.transaction {
    dbQuery.removeAllLaunches()
    launches.forEach { launch ->
        dbQuery.insertLaunch(
            flightNumber = launch.flightNumber.toLong(),
            missionName = launch.missionName,
            details = launch.details,
            launchSuccess = launch.launchSuccess ?: false,
            launchDateUTC = launch.launchDateUTC,
            patchUrlSmall = launch.links.patch?.small,
            patchUrlLarge = launch.links.patch?.large,
            articleUrl = launch.links.article
        )
    }
}

```

Implement the API service

To retrieve data over the internet, you'll use the [SpaceX public API](#) and a single method to retrieve the list of all launches from the `v5/launches` endpoint.

Create a class that will connect the application to the API:

1. In the `shared/src/commonMain/kotlin/com/jetbrains/spacetutorial/` directory, create a `network` package.
2. Inside the `network` directory, create the `SpaceXApi` class:

```

package com.jetbrains.spacetutorial.network

import io.ktor.client.HttpClient

```



```

import io.ktor.client.plugins.contentnegotiation.ContentNegotiation
import io.ktor.serialization.kotlinx.json.json
import kotlinx.serialization.json.Json

class SpaceXApi {
    private val httpClient = HttpClient {
        install(ContentNegotiation) {
            json(Json {
                ignoreUnknownKeys = true
                useAlternativeNames = false
            })
        }
    }
}

```

This class executes network requests and deserializes JSON responses into entities from the `com.jetbrains.spacetutorial.entity` package. The Ktor `HttpClient` instance initializes and stores the `httpClient` property.

This code uses the Ktor `ContentNegotiation` plugin ⁷ to deserialize the result of a `GET` request. The plugin processes the request and the response payload as JSON, serializing and deserializing them as needed.

3. Declare the data retrieval function that returns the list of rocket launches:

```

import com.jetbrains.spacetutorial.entity.RocketLaunch
import io.ktor.client.request.get
import io.ktor.client.call.body

class SpaceXApi {
    // ...

    suspend fun getAllLaunches(): List<RocketLaunch> {
        return httpClient.get("https://api.spacexdata.com/v5/launches")
    }
}

```

The `getAllLaunches` function has the `suspend` modifier because it contains a call of the suspend function `HttpClient.get()`. The `get()` function includes an asynchronous operation to retrieve data over the internet and can only be called from a coroutine or another suspend function. The network request will be executed in the HTTP client's thread pool.

The URL for sending a GET request is passed as an argument to the `get()` function.

Build an SDK

Your iOS and Android applications will communicate with the SpaceX API through the shared module, which will provide a public class, `SpaceXSDK`.

1. In the common source set `shared/src/commonMain/kotlin`, in the `com.jetbrains.spacetutorial` package, create the `SpaceXSDK` class. This class will be the facade for the `Database` and `SpaceXApi` classes.

To create a `Database` class instance, provide a `DatabaseDriverFactory` instance:

```
package com.jetbrains.spacetutorial

import com.jetbrains.spacetutorial.cache.Database
import com.jetbrains.spacetutorial.cache.DatabaseDriverFactory
import com.jetbrains.spacetutorial.network.SpaceXApi

class SpaceXSDK(databaseDriverFactory: DatabaseDriverFactory, val api: SpaceXApi) {
    private val database = Database(databaseDriverFactory)
}
```

You will inject the correct database driver in the platform-specific code through the `SpaceXSDK` class constructor.

2. Add the `getLaunches` function, which uses the created database and the API to get the launches list:

```
import com.jetbrains.spacetutorial.entity.RocketLaunch

class SpaceXSDK(databaseDriverFactory: DatabaseDriverFactory, val api:
    // ...

    @Throws(Exception::class)
    suspend fun getLaunches(forceReload: Boolean): List<RocketLaunch> {
        val cachedLaunches = database.getAllLaunches()
        return if (cachedLaunches.isNotEmpty() && !forceReload) {
            cachedLaunches
        } else {
            api.getAllLaunches().also {
                database.clearAndCreateLaunches(it)
            }
        }
    }
}
```

The class contains one function for getting all launch information. Depending on the value of `forceReload`, it returns cached values or loads the data from the internet and then updates the cache with the results. If there is no cached data, it loads the data from the internet regardless of the `forceReload` flag's value.

Clients of your SDK could use a `forceReload` flag to load the latest information about the launches, enabling the pull-to-refresh gesture for users.

All Kotlin exceptions are unchecked, while Swift has only checked errors (see [Interoperability with Swift/Objective-C](#) for details). Thus, to make your Swift code aware of expected exceptions, Kotlin functions called from Swift should be marked with the `@Throws` annotation specifying a list of potential exception classes.

Create the Android application

IntelliJ IDEA handles the initial Gradle configuration for you, so the `shared` module is already connected to your Android application.

Before implementing the UI and the presentation logic, add all the required UI dependencies to the `composeApp/build.gradle.kts` file:

```
kotlin {  
    // ...  
    sourceSets {  
        androidMain.dependencies {  
            implementation(libs.androidx.compose.material3)  
            implementation(libs.koin.androidx.compose)  
            implementation(libs.androidx.lifecycle.viewmodelCompose)  
        }  
        // ...  
    }  
}
```

Sync the Gradle project files when prompted, or press double `Shift` and search for the **Sync All Gradle, Swift Package Manager projects**.

Add internet access permission

To access the internet, an Android application needs the appropriate permission. In the `composeApp/src/androidMain/AndroidManifest.xml` file, add the `<uses-permission>` tag:

```
<?xml version="1.0" encoding="utf-8"?>  
<manifest xmlns:android="http://schemas.android.com/apk/res/android">  
    <uses-permission android:name="android.permission.INTERNET" />  
</manifest>
```

```
<!--...-->  
</manifest>
```

Add dependency injection

The Koin dependency injection lets you declare modules (sets of components) that you can use in different contexts. In this project, you will create two modules: one for the Android application and another for the iOS app. Then, you will start Koin for each native UI using the corresponding module.

Declare a Koin module that will contain the components for the Android app:

1. In the `composeApp/src/androidMain/kotlin` directory, create the `AppModule.kt` file in the `com.jetbrains.spacetutorial` package.

In that file, declare the module as two singletons, one for the `SpaceXApi` class and one for the `SpaceXSDK` class:

```

package com.jetbrains.spacetutorial

import com.jetbrains.spacetutorial.cache.AndroidDatabaseDriverFactory
import com.jetbrains.spacetutorial.network.SpaceXApi
import org.koin.android.ext.koin.androidContext
import org.koin.dsl.module

val appModule = module {
    single<SpaceXApi> { SpaceXApi() }
    single<SpaceXSDK> {
        SpaceXSDK(
            databaseDriverFactory = AndroidDatabaseDriverFactory(
                androidContext()
            ), api = get()
        )
    }
}

```

The `SpaceXSDK` class constructor is injected with the platform-specific `AndroidDatabaseDriverFactory` class. The `get()` function resolves dependencies within the module: in place of the `api` parameter for `SpaceXSDK()`, Koin will pass the `SpaceXApi` singleton declared earlier.

2. Create a custom `Application` class, which will start the Koin module.

Next to the `AppModule.kt` file, create the `Application.kt` file with the following code, specifying the module you declared in the `modules()` function call:

```

package com.jetbrains.spacetutorial

import android.app.Application
import org.koin.android.ext.koin.androidContext
import org.koin.core.context.GlobalContext.startKoin

class MainApplication : Application() {
    override fun onCreate() {

```

```

        super.onCreate()

        startKoin {
            androidContext(this@MainApplication)
            modules(appModule)
        }
    }
}

```

3. Specify the `MainApplication` class you created in the `<application>` tag of your `AndroidManifest.xml` file:

```

<manifest xmlns:android="http://schemas.android.com/apk/res/android">
    ...
    <application
        ...
        android:name="com.jetbrains.spacetutorial.MainApplication">
        ...
    </application>
</manifest>

```

Now, you are ready to implement the UI that will use the information provided by the platform-specific database driver.

Prepare the view model with the list of launches

You will implement the Android UI using Jetpack Compose and Material 3. First, you'll create the view model that uses the SDK to get the list of launches. Then, you'll set up the Material theme, and finally, you'll write the composable function that brings it all together.

1. In the `composeApp/src/androidMain` source set, in the `com.jetbrains.spacetutorial` package, create the `RocketLaunchViewModel.kt` file:

```

package com.jetbrains.spacetutorial

import androidx.compose.runtime.State
import androidx.compose.runtime.mutableStateOf
import androidx.lifecycle.ViewModel
import com.jetbrains.spacetutorial.entity.RocketLaunch

class RocketLaunchViewModel(private val sdk: SpaceXSDK) : ViewModel() {
    private val _state = mutableStateOf(RocketLaunchScreenState())
    val state: State<RocketLaunchScreenState> = _state
}

data class RocketLaunchScreenState(
    val isLoading: Boolean = false,
    val launches: List<RocketLaunch> = emptyList()
)

```

A `RocketLaunchScreenState` instance will store data received from the SDK and the current state of the request.

2. Add the `loadLaunches` function that will call the `getLaunches` function of the SDK in a coroutine scope of this view model:

```

import androidx.lifecycle.viewModelScope
import kotlinx.coroutines.launch

class RocketLaunchViewModel(private val sdk: SpaceXSDK) : ViewModel() {
    //...

    fun loadLaunches() {
        viewModelScope.launch {
            _state.value = _state.value.copy(isLoading = true, launches = emptyList())
            try {
                val launches = sdk.getLaunches(forceReload = true)
                _state.value = _state.value.copy(isLoading = false, launches = launches)
            } catch (e: Exception) {
                _state.value = _state.value.copy(isLoading = true, launches = emptyList())
            }
        }
    }
}

```



```

        } catch (e: Exception) {
            _state.value = _state.value.copy(isLoading = false, la
        }
    }
}
}
}

```

3. Then add a `loadLaunches()` call to the `init {}` block of the class to request data from the API as soon as the `RocketLaunchViewModel` object is created:

```

class RocketLaunchViewModel(private val sdk: SpaceXSDK) : ViewModel() {
    // ...

    init {
        loadLaunches()
    }
}

```

4. Now, in the `AppModule.kt` file, specify the view model in the Koin module:

```

import org.koin.core.module.dsl.viewModel

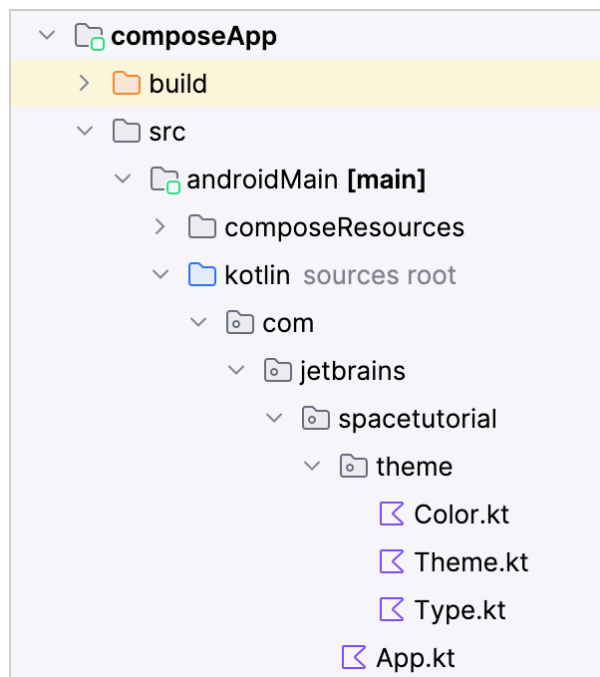
val appModule = module {
    // ...
    viewModel { RocketLaunchViewModel(sdk = get()) }
}

```

Build the Material Theme

You will build your main `App()` composable around the `AppTheme` function supplied by a Material Theme:

1. You can generate a theme for your Compose app using the [Material Theme Builder](#). Pick your colors, pick your fonts, then click **Export theme** in the bottom right corner.
2. On the export screen, click the **Export** dropdown and select the **Jetpack Compose (Theme.kt)** option.
3. Unpack the archive and copy the `theme` folder into the `composeApp/src/androidMain/kotlin/com/jetbrains/spacetutorial` directory.



4. In each file inside the `theme` package, change the `package` line to refer to the package you created:

```
package com.jetbrains.spacetutorial.theme
```

5. In the `Color.kt` file, add two variables for colors you are going to use for successful and unsuccessful launches:

```
val app_theme_successful = Color(0xff4BB543)
val app_theme_unsuccessful = Color(0xffFC100D)
```

Implement the presentation logic

Create the main `App()` composable for your application, and call it from a `ComponentActivity` class:

1. Open the `App.kt` file next to the `theme` directory in the `com.jetbrains.spacetutorial` package and replace the default `App()` composable function:

```
package com.jetbrains.spacetutorial

import androidx.compose.material3.pulltorefresh.rememberPullToRefreshState
import androidx.compose.runtime.Composable
import androidx.compose.runtime.getValue
import androidx.compose.runtime.mutableStateOf
import androidx.compose.runtime.remember
import androidx.compose.runtime.rememberCoroutineScope
import androidx.compose.runtime.setValue
import org.jetbrains.compose.ui.tooling.preview.Preview
import org.koin.androidx.compose.koinViewModel
import androidx.compose.material3.ExperimentalMaterial3Api

@OptIn(
    ExperimentalMaterial3Api::class
)
@Composable
@Preview
fun App() {
    val viewModel = koinViewModel<RocketLaunchViewModel>()
    val state by remember { viewModel.state }
    val coroutineScope = rememberCoroutineScope()
    var isRefreshing by remember { mutableStateOf(false) }
    val pullToRefreshState = rememberPullToRefreshState()
}
```

Here, you are using the [Koin ViewModel API](#) to refer to the `viewModel` you declared in the Android Koin module.

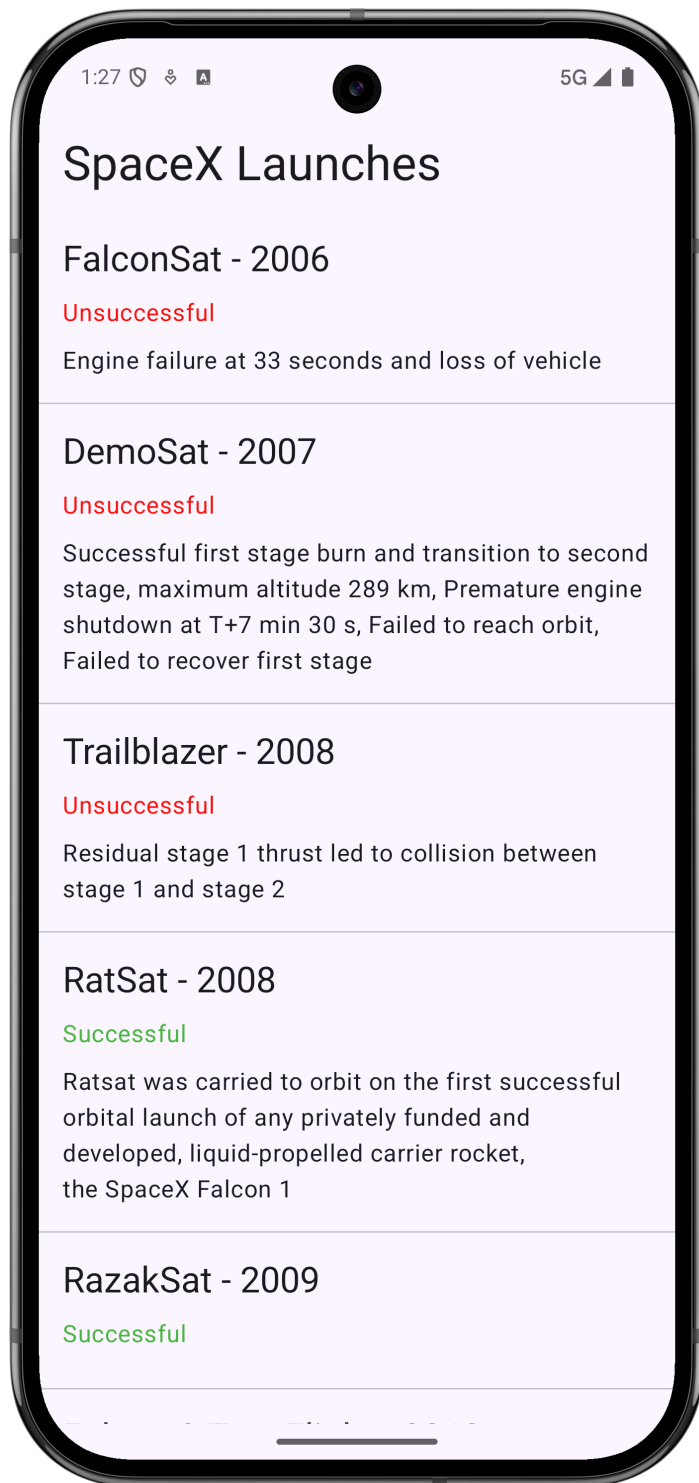
2. Now add the UI code that will implement the loading screen, the column of launch results, and the pull-to-refresh action:

```
+ import com.jetbrains.spacetutorial.theme.AppTheme {...}
```

3. Finally, specify your `MainActivity` class in the `<activity>` tag in the `AndroidManifest.xml` file:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
    ...
    <application
        ...
        <activity
            ...
            android:name="com.jetbrains.spacetutorial.MainActivity">
            ...
        </activity>
    </application>
</manifest>
```

4. Run your Android app: select **composeApp** from the run configurations menu, choose an emulator, and click the run button. The app automatically runs the API request and displays the list of launches (the background color depends on the Material Theme you generated):



You've just created an Android application that has its business logic implemented in the Kotlin Multiplatform module, and its UI made using native Jetpack Compose.

Create the iOS application

For the iOS part of the project, you'll make use of [SwiftUI](#) to build the user interface and the [Model View View-Model](#) pattern.

IntelliJ IDEA generates an iOS project that is already connected to the shared module. The Kotlin module is exported with the name specified in the `shared/build.gradle.kts` file (`baseName = "Shared"`), and imported using a regular `import` statement: `import Shared`.

Add the dynamic linking flag for SQLDelight

By default, IntelliJ IDEA generates projects set up for static linking of iOS frameworks.

To use the native SQLDelight driver on iOS, add the dynamic linker flag that allows Xcode tooling to find the system-provided SQLite binary:

1. In IntelliJ IDEA, select the **File | Open Project in Xcode** option to open your project in Xcode.
2. In Xcode, double-click the project name to open its settings.
3. Switch to the **Build Settings** tab, there switch to the **All** list and search for the **Other Linker Flags** field.
4. Expand the field, press the plus sign next to the **Debug** field, and paste the `-lsqlite3` string into the **Any Architecture | Any SDK**.
5. Repeat the process for the **Other Linker Flags | Release** field.

Other Librarian Flags		
▼ Other Linker Flags		<Multiple values>
Debug		
Any Architecture Any SDK	⬮	-lsqlite3
Release		
Any Architecture Any SDK	⬮	-lsqlite3

Prepare a Koin class for iOS dependency injection

To use Koin classes and functions in Swift code, create a special `KoinComponent` class and declare the Koin module for iOS.

1. In the `shared/src/iosMain/kotlin/` source set, create a file with the name `com/jetbrains/spacetutorial/KoinHelper.kt` (it will appear next to the `cache` folder).
2. Add the `KoinHelper` class, which will wrap the `SpaceXSDK` class with a lazy Koin injection:

```
package com.jetbrains.spacetutorial

import org.koin.core.component.KoinComponent
import com.jetbrains.spacetutorial.entity.RocketLaunch
import org.koin.core.component.inject

class KoinHelper : KoinComponent {
    private val sdk: SpaceXSDK by inject<SpaceXSDK>()

    suspend fun getLaunches(forceReload: Boolean): List<RocketLaunch> {
        return sdk.getLaunches(forceReload = forceReload)
    }
}
```

3. Below the `KoinHelper` class, add the `initKoin()` function, which you will use in Swift to initialize and start the iOS Koin module:

```
import com.jetbrains.spacetutorial.cache.IOSDatabaseDriverFactory
import com.jetbrains.spacetutorial.network.SpaceXApi
import org.koin.core.context.startKoin
import org.koin.dsl.module

fun initKoin() {
    startKoin {
```

```

modules(module {
    single<SpaceXApi> { SpaceXApi() }
    single<SpaceXSDK> {
        SpaceXSDK(
            databaseDriverFactory = IOSDatabaseDriverFactory()
        )
    }
})
}
}
}

```

Now, you can start the Koin module in your iOS app to use the native database driver with the common `SpaceXSDK` class.

Implement the UI

First, you'll create a `RocketLaunchRow` SwiftUI view for displaying an item from the list. It will be based on the `HStack` and `VStack` views. There will be extensions on the `RocketLaunchRow` structure with useful helpers for displaying the data.

1. In IntelliJ IDEA, make sure you are in **Project** view.
2. Create a new Swift file in the `iosApp` folder, next to `ContentView.swift`, and name it `RocketLaunchRow`.
3. Update the `RocketLaunchRow.swift` file with the following code:

```

import SwiftUI
import Shared

struct RocketLaunchRow: View {
    var rocketLaunch: RocketLaunch

    var body: some View {
        HStack() {
            VStack(alignment: .leading, spacing: 10.0) {

```



```

        Text("\(rocketLaunch.missionName) - \(String(rocketLa
        Text(launchText).foregroundColor(launchColor)
        Text("Launch year: \(String(rocketLaunch.launchYear))'
        Text("\(rocketLaunch.details ?? "")")
    }
    Spacer()
}
}
}
}

extension RocketLaunchRow {
    private var launchText: String {
        if let isSuccess = rocketLaunch.launchSuccess {
            return isSuccess.boolValue ? "Successful" : "Unsuccessful"
        } else {
            return "No data"
        }
    }

    private var launchColor: Color {
        if let isSuccess = rocketLaunch.launchSuccess {
            return isSuccess.boolValue ? Color.green : Color.red
        } else {
            return Color.gray
        }
    }
}
}

```

The list of launches will be displayed in the `ContentView` view, which is already included in the project.

4. Create an extension to the `ContentView` class with a `ViewModel` class which will prepare and manage the data. Add the following code to the `ContentView.swift` file:

```

extension ContentView {
    enum LoadableLaunches {
        case loading
    }
}

```

```

        case result([RocketLaunch])
        case error(String)
    }

    @MainActor
    class ViewModel: ObservableObject {
        @Published var launches = LoadableLaunches.loading
    }
}

```

The view model (`ContentView.ViewModel`) connects with the view (`ContentView`) via the Combine framework ↗:

- The `ContentView.ViewModel` class is declared as an `ObservableObject` .
 - The `@Published` attribute is used for the `launches` property, so the view model will emit signals whenever this property changes.
5. Remove the `ContentView_Previews` structure: you won't need to implement a preview that should be compatible with your view model.
 6. Update the body of the `ContentView` class to display the list of launches and add the reload functionality.
 - This is the UI groundwork: you will implement the `loadLaunches` function in the next phase of the tutorial.
 - The `viewModel` property is marked with the `@ObservedObject` attribute to subscribe to the view model.

```

struct ContentView: View {
    @ObservedObject private(set) var viewModel: ViewModel

    var body: some View {
        NavigationView {
            listView()
            .navigationBarTitle("SpaceX Launches")
            .navigationBarItems(trailing:
                Button("Reload") {

```

```

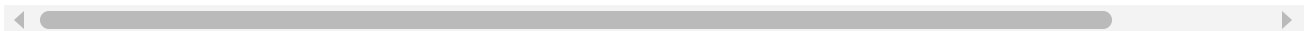
        self.viewModel.loadLaunches(forceReload: true)
    })
}
}

private func listView() -> AnyView {
    switch viewModel.launches {
    case .loading:
        return AnyView(Text("Loading...").multilineTextAlignment(.center))
    case .result(let launches):
        return AnyView(List(launches) { launch in
            RocketLaunchRow(rocketLaunch: launch)
        })
    case .error(let description):
        return AnyView(Text(description).multilineTextAlignment(.center))
    }
}
}

```

7. The `RocketLaunch` class is used as a parameter for initializing the `List` view, so it needs to conform to the `Identifiable` protocol [7](#). The class already has a property named `id`, so all you should do is add an extension to the bottom of `ContentView.swift`:

```
extension RocketLaunch: Identifiable { }
```



Load the data

To retrieve the data about the rocket launches in the view model, you'll need an instance of the `KoinHelper` class from the Multiplatform library. It will allow you to call the SDK function with the correct database driver.

1. In the `ContentView.swift` file, expand the `ViewModel` class to include a `KoinHelper` object and the `loadLaunches` function:

```

extension ContentView {
    // ...
    @MainActor
    class ViewModel: ObservableObject {
        // ...
        let helper: KoinHelper = KoinHelper()

        init() {
            self.loadLaunches(forceReload: false)
        }

        func loadLaunches(forceReload: Bool) {
            // TODO: retrieve data
        }
    }
}

```

2. Call the `KoinHelper.getLaunches()` function (which will proxy the call to the `SpaceXSDK` class) and save the result in the `launches` property:

```

func loadLaunches(forceReload: Bool) {
    Task {
        do {
            self.launches = .loading
            let launches = try await helper.getLaunches(forceReload: forceReload)
            self.launches = .result(launches)
        } catch {
            self.launches = .error(error.localizedDescription)
        }
    }
}

```

When you compile a Kotlin module into an Apple framework, suspending functions [can](#) be called using the Swift's `async` / `await` mechanism.

Since the `getLaunches` function is marked with the `@Throws(Exception::class)` annotation in Kotlin, any exceptions that are instances of the `Exception` class or its subclass will be propagated to Swift as `NSError`. Therefore, all such exceptions can be caught by the `loadLaunches()` function.

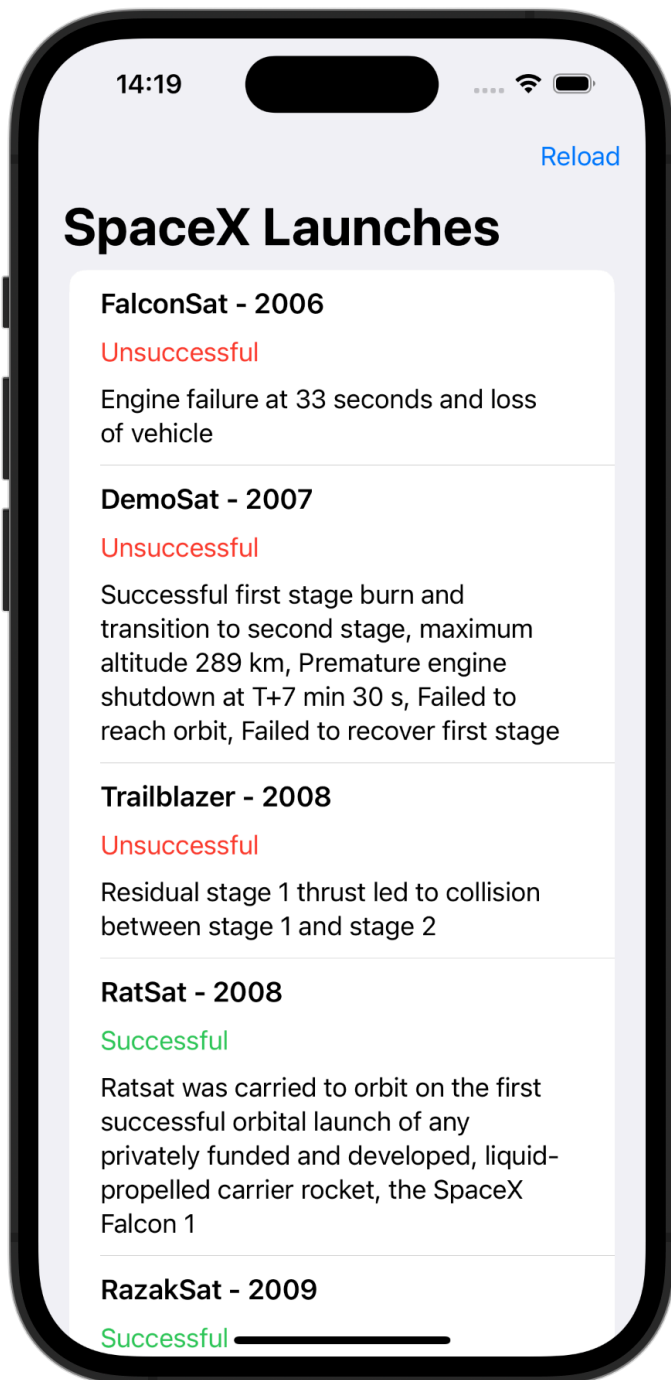
3. Go to the app's entry point, the `iOSApp.swift` file, and initialize the Koin module, the view, and the view model:

```
import SwiftUI
import Shared

@main
struct iOSApp: App {
    init() {
        KoinHelperKt.doInitKoin()
    }

    var body: some Scene {
        WindowGroup {
            ContentView(viewModel: .init())
        }
    }
}
```

4. In IntelliJ IDEA, switch to the **iosApp** configuration, choose an emulator, and run it to see the result:



You can find the final version of the project on the `final` branch ↗.

What's next?

This tutorial features some potentially resource-heavy operations, like parsing JSON and making requests to the database in the main thread. To learn about how to write concurrent code and optimize your app, see the [Coroutines guide ↗](#).

You can also check out these additional learning materials:

- [Use the Ktor HTTP client in multiplatform projects ↗](#)
- [Read about Koin and dependency injection ↗](#)
- [Make your Android application work on iOS](#)
- [Learn more about multiplatform project structure.](#)