package com.dinadurykina.skylexicon.network  
  
*/\*\*  
 \* A generic class that holds a value with its loading status.  
 \* Универсальный класс, который содержит значение со статусом загрузки.  
 \** ***@param*** *<T>  
 \*/*sealed class Result<out **R**> {  
  
 data class Success<out **T**>(val data: **T**) : Result<**T**>()  
 data class Error(val exception: Exception) : Result<Nothing>()  
 object Loading : Result<Nothing>()  
  
 override fun toString(): String {  
 return when (this) {  
 is Success<\*> -> "Success[data=$data]"  
 is Error -> "Error[exception=$exception]"  
 Loading -> "Loading"  
 }  
 }  
}  
  
*/\*\*  
 \* `true` if [Result] is of type [Success] & holds non-null [Success.data].  
 \* 'true', если [Result] имеет тип [Success] и содержит ненулевые [Success.data].  
 \*/*val Result<\*>.*succeeded* get() = this is Result.Success && data != null  
  
/\*  
 private suspend fun getTasks(forceUpdate: Boolean = false): Result<List<Task>> {  
 fun observeTasks(): LiveData<Result<List<Task>>> {  
 return tasksLocalDataSource.observeTasks()  
private val observableTasks = MutableLiveData<Result<List<Task>>>()  
 override fun observeTasks(): LiveData<Result<List<Task>>> {  
 return observableTasks  
override fun observeTask(taskId: String): LiveData<Result<Task>> {  
 return observableTasks.map { tasks ->  
 when (tasks) {  
 is Result.Loading -> Result.Loading  
 is Error -> Error(tasks.exception)  
 is Success -> {  
 val task = tasks.data.firstOrNull { it.id == taskId }  
 ?: return@map Error(Exception("Not found"))  
 Success(task)  
 }  
 }  
 }  
override suspend fun getTask(taskId: String): Result<Task> {  
 // Simulate network by delaying the execution.  
 // Имитация сети путем задержки выполнения.  
 delay(SERVICE\_LATENCY\_IN\_MILLIS)  
 TASKS\_SERVICE\_DATA[taskId]?.let {  
 return Success(it)  
 }  
 return Error(Exception("Task not found"))  
 }  
 \*/

import androidx.lifecycle.Observer  
  
*/\*\*  
 \* Used as a wrapper for data that is exposed via a LiveData that represents an event.  
 \* Используется в качестве оболочки для данных, которые отображаются через живые данные, представляющие событие.  
 \*  
 \* Что такое событие?  
 \* В приложении вы используете настраиваемый Event класс  
 \* для LiveData представления одноразовых событий (таких как навигация или всплывающая закусочная).  
 \* Event LiveData Наблюдается в TasksFragment.  
 \* В этом случае символ newTaskEvent означает, что была нажата клавиша FAB с плюсом, и вы должны перейти к AddEditTaskFragment.  
  
 \*/*open class Event<out **T**>(private val content: **T**) {  
  
 @Suppress("MemberVisibilityCanBePrivate")  
 var hasBeenHandled = false  
 private set // Allow external read but not write Разрешить внешнее чтение, но не запись  
  
 */\*\*  
 \* Returns the content and prevents its use again.  
 \* Возвращает содержимое и предотвращает его повторное использование.  
 \* IfNotHandled Если Не Обработано  
 \*/* fun getContentIfNotHandled(): **T**? {  
 return if (hasBeenHandled) {  
 null  
 } else {  
 hasBeenHandled = true  
 content  
 }  
 }  
  
 */\*\*  
 \* Returns the content, even if it's already been handled.  
 \* Возвращает содержимое, даже если оно уже было обработано.  
 \*/* // заглядывать в контент  
 fun peekContent(): **T** = content  
}  
  
*/\*\*  
 \* An [Observer] for [Event]s, simplifying the pattern of checking if the [Event]'s content has already been handled.  
 \* [Наблюдатель] для [события]s, упрощающий шаблон проверки того, было ли содержимое [события] уже обработано.  
 \*  
 \* [onEventUnhandledContent] is \*only\* called if the [Event]'s contents has not been handled.  
 \* [on Event Unhandled Content] вызывается \*только\* в том случае, если содержимое [события] не было обработано.  
 \*/*class EventObserver<**T**>(private val onEventUnhandledContent: (**T**) -> Unit) : Observer<Event<**T**>> {  
 override fun onChanged(event: Event<**T**>?) {  
 event?.getContentIfNotHandled()?.*let* **{** onEventUnhandledContent(**it**)  
 **}** }  
}  
  
/\* использовать оболочку событий см выше open class Even  
class ListViewModel : ViewModel {  
 private val \_navigateToDetails = MutableLiveData<Event<String>>()  
  
 val navigateToDetails : LiveData<Event<String>>  
 get() = \_navigateToDetails  
  
  
 fun userClicksOnButton(itemId: String) {  
 \_navigateToDetails.value = Event(itemId)  
 // Trigger the event by setting a new Event as a new value  
 // Запустить событие, установив новое событие как новое значение  
 }  
}  
// в фрагменте  
myViewModel.navigateToDetails.observe(this, Observer {  
 it.getContentIfNotHandled()?.let { // Only proceed if the event has never been handled  
 // Продолжаем, только если событие никогда не обрабатывалось  
 startActivity(DetailsActivity...)  
 }  
})  
  
 viewModel.openTaskEvent.observe(viewLifecycleOwner, EventObserver {  
 openTaskDetails(it)  
 })  
 viewModel.newTaskEvent.observe(viewLifecycleOwner, EventObserver {  
 navigateToAddNewTask()  
 })  
  
Преимущество этого подхода состоит в том, что пользователю необходимо указать намерение,  
 используя getContentIfNotHandled() или peekContent().  
 Этот метод моделирует события как часть состояния:  
 теперь они просто сообщение, которое было использовано или нет.  
  
 \*/

/\* ОК: использовать SingleLiveEvent он ограничен одним наблюдателем  
class ListViewModel : ViewModel {  
 private val \_navigateToDetails = SingleLiveEvent<Any>()  
  
 val navigateToDetails : LiveData<Any>  
 get() = \_navigateToDetails  
  
  
 fun userClicksOnButton() {  
 \_navigateToDetails.call()  
 }  
}  
// в фрагменте:  
myViewModel.navigateToDetails.observe ( this , Observer {  
 startActivity ( DetailsActivity ...)  
})  
 \*/

Ивенты

Крутое решение от Гугла — оборачивать дата классы в класс-обертку Event, в котором у нас может быть несколько состояний, как правило это LOADING, SUCCESS и ERROR.

data class Event<out T>(val status: Status, val data: T?, val error: Error?) {

companion object {

fun <T> loading(): Event<T> {

return Event(Status.LOADING, null, null)

}

fun <T> success(data: T?): Event<T> {

return Event(Status.SUCCESS, data, null)

}

fun <T> error(error: Error?): Event<T> {

return Event(Status.ERROR, null, error)

}

}

}

enum class Status {

SUCCESS,

ERROR,

LOADING

}

Вот как это работает. Во время сетевого запроса мы создаем ивент со статусом LOADING. Ждем ответа от сервера и потом оборачиваем данные ивентом и отправляем его с заданным статусом дальше. Во вью проверяем тип ивента и в зависимости от состояния устанавливаем разные состояния для вью. Примерно на такой-же философии строится архитектурный паттерн MVI