GSMobileDevDoc

Выпуск 0.0.7

Global System

февр. 03, 2026

Содержание

1 Пролог

GlobalErp Mobile Framework является платформой быстрой разработки мобильных приложений от компании GlobalERP.

Фреймворк изолирует прикладного разработчика от системной бизнес-логики и задаёт единый стиль на уровне проекта, что позволяет эффективно разрабатывать приложения любого масштаба. Подробнее смотрите *ключевые возможности*.

Фреймворк позволяет избежать *типовых усложнений*, с которыми сталкивается разработчик мобильных приложений при работе со стандартным SDK.

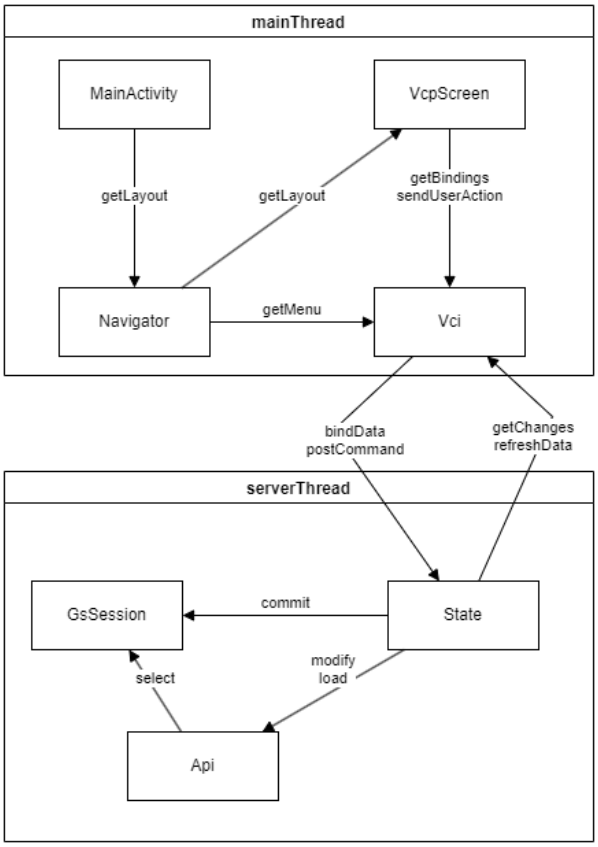
2 Введение

2.1 Ключевые возможности

- Перенос вычислений в *серверный поток*
  Главный поток никогда не блокируется: фрейм-рендер выполнился — задача ушла в очередь, анимация пользовательского интерфейса не зависит от бизнес-логики.
- *Хранение данных*
- Транзакционная обработка данных *GsSession*:
  - SQLite, файлы, сетевые отклики — коммитятся или откатываются одним вызовом.
  - Встроенный генератор ID и горячий кэш сущностей.
- Автоматизация работы с *инъекциями зависимостей*
- *Адаптированный MVVM-паттерн*
- *Навигация* между представлениями
- *Планирование серверных команд*
  Позволяет избавиться от сложных асинхронных запусков.
- *Observable-слой данных*
  `RecordSet`, `SingleRecord`, `RecordValue<T>` передают изменения сразу в Compose-дерево, без необходимости писать адаптеры.
- *Плагин-подход к железу*
  `ActivityPlugin` регистрируется в `GsBaseActivity`; доступ к NFC или камере получаем без Dagger/Hilt.
- Горячий офлайн
  Легко настраивать работу, копить данные и синхронизировать по необходимости
  за счёт *встроенной поддержки БД*.
- Простое тестирование
  `State`/`VCI` можно запустить с `database = null`, логика тестируется без Android-эмулятора, при этом транзакционная модель сохраняется.

2.2 Архитектура

Взаимодействие слоев



Основные понятия

- *Главный поток* (`mainThread`) — поток взаимодействия с пользователем; этот поток не должен блокироваться. Рендерит Compose, обрабатывает жесты и никогда не исполняет длительных операций.
- *Серверный поток* (`serverThread`) — поток для выполнения синхронной бизнес-логики и синхронизации с асинхронными задачами. Последовательно выполняет SQL, операции с файлами, трансформации стека и любые другие долгие операции. Все вызовы VCI к тяжёлым ресурсам отправляются сюда через `postSharedTask` или `postAsyncTask` (минуют карусель событий).
- *Сессия* (*GsSession*) — сессия работы с данными.
  Содержит контекст для хранения и управления компонентами, необходимыми для обработки данных.
- *Состояние* (`State`) — содержит данные и бизнес-логику представления, работающую в серверном потоке.
- *Стек состояний* — хранит историю открытых представлений. Верхнее представление отображается на экране.
  Позволяет пользователю возвращаться на предыдущее представление.

- *Контроллер представления* (`VCI`) — содержит данные и бизнес-логику представления в главном потоке.
- *Представление* (`VcpScreen`) — декларация правил отрисовки представления с использованием библиотеки Compose.
- *Контроллер сессии* (`Api`, `Pkg`) — контроллер сессии обрабатывает бизнес-логику в серверном потоке в разрезе сущностей базы данных или их групп.
- *Навигатор* (`Navigator`) — занимается компоновкой и управлением представлениями приложения.
- *Главная активность* (`MainActivity`) — точка входа Android-приложения, подробнее смотрите руководство по разработке на платформе Android.

Отдельный серверный поток

Отдельный серверный поток для выполнения бизнес-логики лучше, чем рассыпанные потоки `launch(Dispatchers.IO)`, так как позволяет достичь:

- более высокой производительности на последовательных вычислениях за счёт отсутствия необходимости
  межпотоковой синхронизации;
- более простой разработки и отладки за счёт:
  - последовательного выполнения, что позволяет использовать классические принципы структурного программирования;
  - полностью воспроизводимого stack trace —
    нет прыжков между диспатчерами;
  - декларативной двусторонней синхронизации между главным и серверным потоком;
  - контролируемой отмены:
    `InterruptingLock.validate()` бросит исключение на любой стадии, стек откатится, а UI мирно вернётся к последнему стабильному экрану.

Адаптированный MVVM-паттерн

Обеспечивает:

- Предсказуемый интерфейс
  Стабильный FPS даже на сложных формах.
- Единый стиль
  Все экраны одинаково устроены, code review смотрит в основном на логику, а не на повторяющиеся слои.

Детализация принципа view–viewmodel–model:

- `View` — декларация интерфейса на Compose-UI, *объявляется в VCI*.
- `viewmodel` — разбит на 2 слоя для отделения бизнес-логики главного и серверного потоков:
  - *VCI* — бизнес-логика для работы в главном потоке, а также подписка на данные `state`.
  - *State* — бизнес-логика для работы в серверном потоке, а также сбор данных для представления.
- `model`:
  - *Api* — контроллер таблицы для обработки хранимых данных;

– *Entity* — модель хранимых данных.

Жизненный цикл экранов

Таблица ниже отображает карусель событий, которые произойдут при открытии и закрытии экрана.

Событие	Когда вызывается	Что делать
`onInit`(State, VCI)	Сразу после создания, синхронно	Подготовить запросы к БД, инициализировать `RecordSet`
`afterPush`(Sta	State помещён в стек	Подписаться на шину, запустить лёгкие preload-задачи
`onVisit() : SmTrans`(State)	Сразу после push, но до `afterEnter`	Вернуть `SmTrans` для мгновенного перенаправления (например Splash → Login) или `null`
`afterEnter`(St	Каждый раз, когда State стал активным	Обновить данные, запустить `postSharedTask`; триггерится на каждую фоновую задачу
`afterEnter`(V	UI создан и синхронизирован с главным потоком	Восстановить scroll-позицию, открыть диалоги, получить данные из State — главная точка синхронизации перед показом экрана
`beforeExit`(V	UI ещё на экране, но пользователь уходит	Сохранить scroll, закрыть диалоги, очистить временные данные
`beforeExit`(St	UI уже закрыт, транзакция ещё открыта	`flush()` RecordSet, финальные изменения в БД, подготовка к уходу
`onError(e)`(St	Любая необработанная ошибка внутри State	Вернуть `SmTrans` на экран ошибки или кастомизировать обработку
`onClose`(State)	State удалён из стека окончательно	Отписаться от шины, освободить ресурсы

Дополнительные понятия

- *Менеджер состояний*(`StateManager`) - отвечает за обработку очереди событий для серверного потока и транзакционной логики обработки переходов между состояниями. StateManager обращается к UI из серверного потока безопасно через блокировки, поэтому состояние стека остаётся консистентным.

2.3 Тестирование и масштабирование

- Любой `State` и `VCI` можно запустить с `database = null` логика тестируется без эмулятора, транзакционная модель при этом сохраняется.
- Новый экран требует три файла — `State`, `VCI`, `Compose-View` — и не затрагивает существующий код; команда растёт линейно, не переписывая инфраструктуру.

3 Быстрый старт

Ниже — минимальный, но полностью рабочий пример приложения на GMF.
Мы пройдём по всем слоям: от Entity до Compose-экрана, подключим DI-генератор и объясним, куда поместить каждый файл.

3.1 Шаблон проекта

Зависимости в Gradle

```
// build.gradle (модуль :app)
plugins {
    id("com.android.application")
    kotlin("android")
    alias(libs.plugins.google.devtools.ksp)   // KSP для DI-процессора
}

android { /* стандартная конфигурация */ }

dependencies {
    implementation(libs.kotlinx.coroutines.android)
    implementation(libs.androidx.compose.material3)
    ksp(libs.androidx.room.compiler)

    implementation(project(":common"))        // GMF
    ksp(project(":di-processor"))             // кодогенерация @GsApiBean / @GsPkgBean
}
```

Скелет каталогов

```
src/main/java
└─ ru.my.app
   ├─ api/                  - классы доступа к БД (UserApi и др.)
   ├─ db/                   - Entity, Dao, AppDatabase
   ├─ pkg/                  - сетевые / сервисные пакеты (опционально)
   ├─ ui/
   |   └─ users/
   |       ├─ UsersState.kt
   |       ├─ UsersVci.kt
   |       └─ view/
   |           ├─ UsersView.kt
   |           └─ ViewProvider.kt
   ├─ MainActivity.kt       - точка входа
   └─ Delegates.kt          - DataStore, расширения навигатора и т.д.
```

ORM-классы

```
// db/User.kt
@Entity
data class User(
    var name  : String? = null,
    var email : String? = null,
    var age   : Int?    = null,
) : BaseEntity() {
    override fun copyEntity() = copy()
}

// db/UserDao.kt
@Dao
abstract class UserDao : BaseDao<User>()
```

Room-база

```
// db/AppDatabase.kt

@Database(
    entities = [SystemEntity::class, User::class],
    version = 1,
    exportSchema = false
)
abstract class AppDatabase : RoomDatabase() {
    abstract val userDao: UserDao

    companion object {
        @Volatile private var INSTANCE: AppDatabase? = null

        fun getInstance(ctx: Context): AppDatabase =
            INSTANCE ?: synchronized(this) {
                INSTANCE ?: Room.databaseBuilder(
                    ctx,
                    AppDatabase::class.java,
                    "AppDB"
                ).build().also { INSTANCE = it }
            }
    }
}
```

Если нужна базовая реализация БД, можно использовать в `MainActivity`:

```
override fun provideDatabase(): PrototypeDatabase =
    RoomDbSingleton.getInstance(this, "PrototypeDatabase")
```

и не создавать `companion object` в классе БД для хранения инстанса.

DI-процессор

Аннотация `@GsApiBean` говорит KSP-процессору сгенерировать расширение `GsSession.getUserApi()` — обращение к нашему API без строковых имён.

```
// api/UserApi.kt
@GsApiBean
class UserApi(ctx: ApiBeanContext)
    : DbBaseApiGen<User, UserDao, AppDatabase>(ctx) {

    override val dao get() = dbRoom.userDao
    override fun newEntity() = User()

    /** Демо-данные при первом старте */
    fun seed() {
        if (fetchAll().isNotEmpty()) return
        listOf(
            "Alice" to "alice@site.com",
            "Bob"   to "bob@site.com",
            "Chloe" to "chloe@site.com",
        ).forEach { (n, e) ->
            insert().update {
                it.name  = n
                it.email = e
                it.age   = (20..45).random()
            }
        }
        flush()
    }
}
```

Бизнес-логика экрана

```
// ui/users/UsersState.kt
class UsersState : SmStateAbst<UsersState>() {

    val usersRS = newRecordSet()          // RecordSet

    override fun onInit() {
        usersRS.onPopulate { q ->
            q.query("SELECT * FROM User ORDER BY name")
        }
    }

    override fun afterEnter() {
        dbs.getUserApi().seed()           // лениво создаётся через DI
        usersRS.refresh()
    }

    override fun newVci(): SmStateVcp = UsersVci()
}
```

Контроллер главного потока

```
// ui/users/UsersVci.kt
class UsersVci : SmStateVciAbst<UsersState>() {

    var usersORL = nullObservableRecordList

    override fun afterEnter(st: UsersState) {
        usersORL = st.usersRS.observableRecordList
    }

    override fun newScreen() = provideUsersView(this)

    override fun getTitle() = "Пользователи"
}
```

VCI — View Control Interface

Пользовательский интерфейс

```
// ui/users/view/UsersView.kt
fun provideUsersView(vci: UsersVci) = object : VcpScreen {
    @Composable
    override fun Content(pv: PaddingValues) {
        val users by vci.usersORL.observeAsState()
        LazyColumn(
            Modifier
                .fillMaxSize()
                .padding(pv)
        ) {
            items(users) { or ->
                Card(
                    modifier = Modifier
                        .fillMaxWidth()
                        .padding(8.dp),
                    onClick = { /* переход на карточку */ }
                ) {
                    Column(Modifier.padding(16.dp)) {
                        Text(
                            or.getValueAsString("name"),
                            style = MaterialTheme.typography.titleMedium
                        )
                        Text(
                            or.getValueAsString("email"),
                            style = MaterialTheme.typography.bodyMedium
                        )
                    }
                }
            }
        }
    }
}
```

Точка входа

```
// MainActivity.kt
class MainActivity : GsBaseActivity<AppNavigator>() {

    @Composable
    override fun ContentView(navigatorCtrl: NavigatorCtrl) {
        GlobalSystemAppTheme {
            navigator.setBaseMenu(
                listOf(
                    DrawerItem("Пользователи") {
                        getSts().postCallState<UsersState>()
                    }
                )
            )
            GsDrawerNavigatorViewV2(navigatorCtrl)
        }
    }

    override fun provideNavigator()  = AppNavigator()
    override fun provideDatabase()   = AppDatabase.getInstance(this)
    override fun provideFirstState() = UsersState()
}
```

Итог

- Все SQL-операции, сеть и файлы уже работают внутри транзакций `GsSession`.
- UI-поток чист: ни одного `launch(Dispatchers.IO)` в пользовательском коде.
- Навигация описана в три строки (DrawerItem → `postCallState`).
- Расширение команды: новый экран — это ещё `State + VCI + View`, инфраструктуру трогать не нужно.

Получился полноценный экран, который:

1. Умеет читать и писать в БД транзакционно.
2. Никогда не блокирует главный поток.
3. Восстанавливается после сбоев приложения (снимок стека сохраняет State Manager).
4. Готов к расширению: добавление сети, плагинов камеры, офлайн-синхронизации и т.д.

4 Состояние

State создаётся для представления и выполняет следующие функции:

- Хранит данные для контроллера представления

> **Примечание**
>
> UI-слой только подписывается на *Observable*-источники, не знает о SQL и транзакциях.

- Позволяет обращаться к контроллерам данных, использовать БД и сеть
- Запускает тяжёлые задачи
- Принимает решения о навигации

Внимание

Любые запросы к данным должны проходить через State, чтобы гарантировать сериализацию и единый откат.

4.1 Минимальный скелет

```
class UsersState : SmStateAbst<UsersState>() {

    /* ----------- Данные ----------- */
    val usersRS = newRecordSet()

    /* ----------- Контроллер ----------- */
    override fun newVci(): SmStateVcp = UsersVci()

    /* 1. Конструируем запросы и подписки */
    override fun onInit() {
        usersRS.onPopulate { q ->
            q.query("SELECT * FROM Users ORDER BY sName")
        }
    }

    /* 2. Первый вход в стек (UI ещё не создан) */
    override fun afterEnter() {
        dbs.getApi(UserApi::class).seed()   // демо-данные
        // выполняет onPopulate() каждый раз; если вызвать populate(), то считывание
↪будет однократным
        usersRS.refresh()
    }
}
```

4.2 Жизненный цикл

Этап	Поток	Назначение
`onInit()`	Server	Вызывается один раз после создания; регистрируем SQL-запросы и подписки
`afterPush()`	Server	Сразу после `stateStack.push()`; лёгкий хук, не запускать тяжёлые операции
`onVisit()`	Server	Даёт возможность выполнить сквозной переход до создания UI; вернуть `SmTrans` или `null`
`afterEnter()`	Server	Каждый раз, когда State становится верхним; обновляем данные, запускаем `SharedTask`
`afterEnterGui`	Main	UI построен; точка синхронизации в VCI (scroll, диалоги, реакция на данные)
`beforeExitGui`	Main	Пользователь покидает экран; сохраняем scroll, закрываем диалоги, забираем изменённые поля
`beforeExit()`	Server	UI уже снят; транзакция ещё открыта — пишем изменения в БД, вызываем `flush()` у `RecordSet`
`onError(e)`	Server	Ловим необработанные ошибки; можно вернуть альтернативный `SmTrans`
`onClose()`	Server	State окончательно удалён из стека; освобождаем ресурсы, отписываемся

4.3 Контейнеры данных

Список записей

Предоставляет удобные способы работы как из главного, так и из серверного потоков.

Для серверного потока:

- запрос данных из базы данных

Для главного потока:

- получение данных
- возможность безопасного редактирования с последующей передачей изменений в серверный поток

```
val ordersRS = newRecordSet()

ordersRS.onPopulate {
    it.query("SELECT * FROM Orders WHERE gidCustomer = ?", arrayOf(custGid))
}

ordersRS.onUpdateRecord { changes ->
    val newQty = changes.getNewValueAsInt("nQuantity")
    dbs.execSql(
        "UPDATE Orders SET nQuantity = ? WHERE id = ?",
        newQty,
        changes.id
    )
}
```

Единственная строка

Обёртка над списком записей для удобной работы с одной строкой.

```
val orderSR = newSingleRecord()

orderSR.onPopulate {
    it.query("SELECT * FROM Orders WHERE gid = ?", arrayOf(orderGid))
}
```

Строка по значению

Декоратор над набором строк, позволяющий преобразовывать data-класс в строку и обратно.

```
val editMode = newRecordValue(EditFlags(isReadonly = false))
```

4.4 Исполнители

Используются для выделения конкретного потока в разрезе Activity для выполнения специализированных задач.

```
val camExec = newCameraExecutorSubscription()

sm.doLaunch("resize") {
    camExec.executor.submit {
        val path = imageUtil.resize(raw)
        postSharedTask("link photo") { st ->
            dbs.getApi(FileApi::class).attach(orderGid, path)
        }
    }
}
```

Исполнитель запускается, когда `StateManager` активен, и гасится при `onStop()` Activity.

Внимание

Не забывайте вызывать `close()` у `ExecutorSubscription`, если держите его дольше жизни State.

4.5 Сквозной переход

Позволяет автоматически переходить на следующий экран без ожидания действий пользователя.

```
override fun onVisit(): SmTrans? =
    if (dbs.getPkg<AuthPkg>().hasToken())
        callStateTrans<HomeState>()      // уже залогинен
    else
        callStateTrans<LoginState>()     // требуется авторизация
```

4.6 Отмена длительных операций

`InterruptingLock.validate()` вызывается в потенциально длинных циклах.

Если пользователь нажал «Отмена»:

- выбрасывается `CancelTaskException`
- происходит `rollback()`
- UI возвращается к последнему стабильному экрану.

4.7 Хорошие практики

- *Один экран — один State*
  Не склеивайте разные сущности.
- *SQL/REST — только из State*
  VCI должен оставаться чистым.
- *Методы State делайте идемпотентными*
  Пользователь может нажать кнопку дважды.

5 Навигатор

Класс, отвечающий за стандартное поведение пользовательского экрана:

- компоновка панелей и текущего представления
- перерисовка элементов
- блокировка экрана
- показ диалогов
- работа с меню и внутренней шиной событий
- и т.п.

Контроллер представления получает доступ к навигатору через свойство `navigator`.

Контроллер представления может изменить стандартное поведение, переопределив соответствующие методы.
Перечень методов для переопределения определён в интерфейсе `NavigableVcp`.

Классы, реализующие навигатор:

- `GsBottomNavigationBarView`
- `GsDrawerNavigationView`

5.1 Пользовательский Navigator

Фреймворк поставляет базовую реализацию, но вы можете создать свою, реализовав интерфейс `Navigator` и передав её в `GsBaseActivity.provideNavigator()`.
Все контроллеры продолжат работать: интерфейс стабилен.

5.2 Отображение экрана

Экран отрисовывается на основе следующих элементов:

- Контроллер представления
  Используется контроллер представления верхнего состояния в стеке состояний.
  - `View` — разметка на основе библиотеки Compose
  - `NavigableVcp` — инъекция зависимости в навигатор, которая может переопределить стандартное поведение:
    * состав меню
    * видимость меню
    * FAB
    * наименование экрана
- `TopBar`
- `BottomBar`
- `Drawer`
  Только в случае `GsDrawerNavigationView`
- диалоговые окна
- загрузочные окна

В стеке никогда не хранится более одного экрана для каждого `SmState`.
Повторный вызов `State` просто обновляет существующий элемент.

5.3 TopBar

Плашка вверху экрана, используется для:

- навигационной кнопки
  Часто используется для кнопки вызова меню или кнопки «назад».
- наименования экрана
- кнопок действий

Для понимания принципов использования смотрите `AppBar` в `Google Material Design`.

Функции навигатора:

- `hideTopAppBar()`
- `showTopAppBar()`
- `topGsMenuItems`

5.4 BottomBar

Плашка внизу экрана, используется для:

- кнопок глобальной навигации

Для понимания принципов использования смотрите `NavigationBar` в `Google Material Design`.

Функции навигатора:

- `hideBottomBar()`
- `showBottomBar()`
- `setBadgeFor(vciClassName, value)`
  В случае `GsBottomNavigationBarView` задаёт индикацию глобальным переходам.

5.5 Drawer

Выезжающая боковая панель, используется для глобальных переходов и действий.

5.6 Список глобальных переходов

Задаёт доступные переходы в состояния, которые пользователь может выполнить из `Drawer` или `BottomBar` в зависимости от используемого навигатора.

Функции навигатора:

- `setBaseMenu()`

5.7 FAB

Плавающая кнопка, которая размещается поверх интерфейса и предназначена для выполнения главного действия на экране.

По умолчанию иконка + выключена. Переопределяется в `NavigableVcp`.

5.8 Блокировка и диалоги

Функции навигатора:

- `lock("Sync...")` — показывает полноэкранный индикатор и блокирует навигационные жесты
- `unlock()` — снимает блокировку
  Это удобно для сетевых операций, которые важно завершить без выхода пользователя.
- `createDialog(title, msg, ...)` — упрощённый API для простых диалогов «OK/Cancel»
  Данные хранятся в `navigator.simpleDialog`, Compose-слой реагирует автоматически.

5.9 Шина событий

Навигатор предоставляет компактную шину событий (на основе `Flow` Kotlin) между независимыми контроллерами представлений без глобальных синглтонов.

Функции контроллера представления:

- `subscribeEventBus`

```
// подписка
vci.subscribeEventBus<MyPayload>("UpdateEvent") { payload ->
    // реакция
}
```

- `sendEventBus`

```
// публикация
vci.sendEventBus("UpdateEvent", MyPayload(...)) { error ->
    // обработка ошибок
}
```

Примечание

- За каждым именем события хранится `SharedFlow`.
- При закрытии экрана `closeEventBus(name)` автоматически отменяет подписку, предотвращая утечки.

5.10 Индикатор синхронизации

Функции навигатора:

- `setNeedSync(boolean)`
  Включает или выключает иконку «обновить» в `TopBar`.
- `onClickSyncBtn`
  Передаёт обработчик в текущий `VCI`; типичная реализация — вызов `postSharedTask` для синхронизации с сервером.

5.11 Примечание

Пример расширения Navigator

Пример расширения для работы с NFC:

```
class MyNavigator :
    NavigatorCtrl(),
    NavigatorNfc {

    override fun <T> onNfcRead(
        data: String,
        ss: SmStateSubject<out SmState>,
        onRead: (String) -> Unit,
```

(продолжается на следующей странице)

(продолжение с предыдущей страницы)

```
        onShow: (T) -> Unit
    ) {
        // обработка NFC-данных
    }
}

interface NavigatorNfc {
    fun <T> onNfcRead(
        data: String,
        ss: SmStateSubject<out SmState>,
        onRead: (String) -> Unit = {},
        onShow: (T) -> Unit = {}
    )
}

// MainActivity

navigator.onNfcRead<Map<String, String>>(it, getSts()) { data ->
    nfcData = data
    showNfcDialog.value = true
}
```

Пример MainActivity

```
class MainActivity : GsBaseActivity<AppNavigator>() {

    @Composable
    override fun ContentView(navigatorCtrl: NavigatorCtrl) {
        AppTheme {
            navigator.setBaseMenu(
                listOf(
                    DrawerItem("Users") { getSts().postCallState<UsersState>() },
                    DrawerItem("Settings") { getSts().postCallState<SettingsState>() }
                )
            )
            GsDrawerNavigatorView(navigatorCtrl)
        }
    }

    override fun provideNavigator()  = AppNavigator()
    override fun provideDatabase()   = AppDatabase.getInstance(this)
    override fun provideFirstState() = UsersState()
}
```

Главная Activity предоставляет `Navigator`, базовое меню и первое состояние; дальше всё управление переходит к `StateManager` и VCI-слою.

Навигационный слой GMF изолирует UI-логику от бизнес-кода: контроллеры формируют интеракции, а `Navigator` обеспечивает плавные переходы, блокировки, меню и коммуникацию между экранами — всё в нескольких вызовах без шаблонного кода.

6 Контроллер представления

VCI (View-Controller Interface) располагается между серверной логикой `State` и Jetpack Compose-представлением.
Ключевые функции контроллера:

- обновление экрана
- управление доступными действиями пользователя
- управление визуальными компонентами
- хранение данных главного потока
  Контроллер переживает пересоздание представления, например при повороте экрана.
- предоставление навигационных функций

Внимание

Весь код, связанный с UI-слоем, должен находиться именно в представлении,
а бизнес-данные остаются внутри соответствующего State.

6.1 Создание контроллера

Контроллер представления должен наследовать базовый класс `SmStateVciAbst<S : SmState>`.

Шаблон

```
class UsersVci : SmStateVciAbst<UsersState>() {

    /* nullable RecordList, чтобы не использовать lateinit */
    private var usersORL = nullObservableRecordList

    override fun onInit(state: UsersState) {
        usersORL = state.usersRS.observableRecordList
    }

    override fun afterEnter(state: UsersState) {
        usersORL = state.usersRS.observableRecordList      // подписываемся на изменения
    }

    override fun newScreen() = provideUsersView(this)

    override fun getTitle() = "Пользователи"
}
```

6.2 Жизненный цикл

Фаза	Описание
`onInit(stat`	Вызывается один раз, когда VCI создан. Подписаться на `RecordSet`, настроить меню.
`afterEnter(`	Каждый раз, когда `State` становится активным (после `SmTrans` или `postSharedTask`). Обновить UI-данные, получить результат вычислений в стейте.
`beforeExit(`	Перед уходом со страницы. Сохранить позицию списков, закрыть диалоги, отправить накопленные изменения в стейт.

6.3 Общие свойства и методы

- `navigator` — *Навигатор*
- `lazyListState` — используется для хранения позиции основного списка представления

6.4 Работа с данными

Инициализация данных

Kotlin требует, чтобы все значения по возможности были инициализированы. Для удобства в контроллере объявлены начальные значения:

- `nullObservableRecordList`
- `nullObservableRecord`

Подписка на набор строк

Compose-экраны не могут обращаться к БД и ресурсам серверного потока напрямую. Поэтому данные обычно собираются в серверном потоке и сохраняются в состоянии. Для работы с данными, сохранёнными в состоянии, используется механизм привязки. Это позволяет безопасно использовать данные в главном потоке, полученные из серверного потока.

Классы, публикующие данные для состояния:

- `RecordSet`
- `SingleRecord`
- `RecordValue`

Внутри себя они содержат:

- `observableRecordList`
- `observableRecordHolder`

Это позволяет использовать стандартные механизмы подписки Compose.

Привязка происходит в момент инициализации:

```
override fun onInit(state: UsersState) {
    usersORL = state.usersRS.observableRecordList      // подписались один раз
}
```

Пример использования привязанных данных:

```
@Composable
fun UsersView(vci: UsersVci, pv: PaddingValues) {
    val users by vci.usersORL.observeAsState()          // ← живой список

    LazyColumn(Modifier.padding(pv)) {
        items(users) { or ->
            Text(or.getValueAsString("name"))
        }
    }
}
```

Подписка происходит в главном потоке. Все изменения данных, пришедшие из серверного потока, автоматически попадают в Compose.

Чтение и модификация значений

Каждый элемент списка — это `RecordData`.
Доступ к полям идёт через геттеры/сеттеры.

Пример чтения:

```
val email = or.getValueAsString("email")
val age   = or.getValueAsInt   ("age")
```

Пример изменения:

```
IconButton(onClick = { or.setValue("isSelected", 1) }) { /* ... */ }
```

Примечание

Синхронизация изменений между главным потоком и серверным потоком происходит в `beforeExit()` VCI.
Когда пользователь завершит работу с экраном, мы можем сбросить изменения в БД одной пачкой.

6.5 Планирование серверных команд

Основные понятия:

- *задачи* — выполняются в контексте текущего `State`, не меняют стек состояний. Одновременно может
  быть запланировано несколько задач.
- *переходы* — добавляют или удаляют состояния из стека состояний.
  Одновременно может быть запланирован только один переход.
  Переход происходит транзакционно: SQLite-транзакция, `FileManager` и пользовательский экран согласованно перейдут в новое состояние или, в случае ошибки, произойдёт откат до начала выполнения перехода.

См. также

- `StateEventProcessor`

Для отправки действий в серверный поток используется планировщик `getSts()`.

Методы `getSts()`:

- `postSharedTask`:

```
withLock("Загрузка...") {
    getSts<UsersState>().postSharedTask("refresh") { st ->
        st.dbs.getApi(UserApi::class).updateFromServer()
    }
    .onSuccess {
        // успешное завершение
        // можем запланировать еще работу, переходы и т.д.
        // например, getSts().postBack{}
    }
    .onFailure {
        showBaseDialog("Ошибка", it.msg)
    }
}
```

Совет

Используйте `withLock`, чтобы надёжно показать индикатор и гарантированно его убирать.

- `postSharedTask()`
  Перед/после выполнения задачи происходит синхронизация данных между контроллерами.
- `postAsyncTask()`
  Выполняет задачу без событий синхронизации.
- `postCall()`
  Планирование перехода, при этом состояние добавляется в стек состояний; основной метод перехода вперёд.
- `postBack()`
  Планирование перехода назад (снятие состояния со стека); основной метод перехода назад.
- `postCallWithResult()`
  Планирование перехода вперёд с ожиданием результата.
- `postBackWithResult()`
  Планирование перехода назад (снятие состояния со стека) с результатом для ожидающего состояния.

Внимание

Коллбеки `onSuccess / onFailure` приходят в главном потоке, блокировка UI не требуется.

Чтобы получить результат `postCallWithResult`, нужно вызвать зеркальный метод `postBackWithResult`.
Если результат не будет передан при возвращении, произойдёт ошибка; сам результат обрабатывается в параметре `onResult`.

При попытке вызвать `postBackWithResult` без ожидающего состояния также будет выброшена ошибка.
Таким образом, разработчик всегда знает, когда что-то не пришло или было отправлено по ошибке.

Примеры для каждого варианта:

`postSharedTask` — **задача с синхронизацией и обработкой результата**

```
fun refreshUsers() {
    showLoadingView()
    getSts().postSharedTask("refreshUsers") { st ->
        st.dbs.getApi(UserApi::class).updateFromServer()
    }
    .onSuccess {

    }
    .onFailure {
        stopLoadingView()
        showBaseDialog(
            title = "Ошибка",
            msg = it.message ?: "Не удалось обновить данные",
            isError = true
        )
    }
}
```

`postAsyncTask` — **без синхронизации**

```
fun sendAnalyticsEvent(event: String) {
    getSts().postAsyncTask("analyticsEvent") { st ->
        st.dbs.execSql(
            "INSERT INTO AnalyticsLog(event) VALUES(?)",
            event
        )
    }
    // без `afterEnter`
}
```

`postCall` / `postCallState` — **переход на новый экран**

```
// Переход на экран камеры без результата
fun toCamera() {
    showLoadingView()
    getSts().postCallState<GsCameraState> { tb ->
        tb.onBefore { bb ->
            bb.afterSubscribe { stTo ->
                stTo.cameraUseCaseState = GsCameraUseCaseState.IMAGE_CAPTURE
            }
        }
    }
}
```

`postBack` — **возврат назад**

```
override fun smPostBack() {
    if (mediaPreviewState.value == MediaPreviewState.HIDDEN) {
        getSts().postBack { /* можно настроить onBefore/onAfter, если нужно */ }
    } else {
        mediaPreviewState.value = MediaPreviewState.HIDDEN
        showBottomBar()
    }
}
```

`postCallStateWithResult` — **переход с ожиданием результата**

Пример: открываем камеру, ждём результат `CameraResult`, обрабатываем его в текущем VCI.

```
fun toQrScanner() {
    showLoadingView()

    getSts().postCallStateWithResult<GsCameraState, CameraResult>(
        body = { tb ->
            tb.onBefore { bb ->
                bb.afterSubscribe { stTo ->
                    stTo.cameraUseCaseState = GsCameraUseCaseState.QRCODE_SCANNER
                    stTo.qrDelegate = this@CreateDemandVci
                }
            }
        },
        onResult = { result ->
            stopLoadingView()
            if (result is CameraResult.Qr) {
                showBaseDialog(
                    title = "Информация",
                    msg = result.text
                )
            }
        }
    )
}
```

`postBackWithResult` — **возврат назад с результатом**

Экран-дочерний возвращает результат родителю:

```
fun backWithQr(qrCode: String) {
    showLoadingView()
    getSts().postBackWithResult(CameraResult.Qr(qrCode))
}

fun backErrorWithQr(e: Throwable) {
    showLoadingView()
```

(продолжается на следующей странице)

(продолжение с предыдущей страницы)

```
    getSts().postBackWithResult(CameraResult.Error(e))
}
```

Композиция бизнес-логики

- Используйте `getSts` только из VCI.
  Это позволит:
  - избежать ошибок доступа к данным из разных потоков;
  - повысить читаемость кода.

```
@Composable
fun Toolbar(vci: UsersVci) {
    IconButton(onClick = { vci.refreshUsers() }) { /* ... */ }
}

// в VCI
fun refreshUsers() {
    showLoadingView()
    getSts().postSharedTask("refresh") { st ->
        st.dbs.getApi(UserApi::class).syncFromServer()
    }.onSuccess {
        usersORL.refresh()         // обновляем список
        stopLoadingView()
    }.onFailure {
        showBaseDialog(
            "Ошибка",
            it.message ?: "Не удалось обновить данные",
            isError = true
        )
    }
}
```

6.6 Меню

Список пунктов меню задаётся в событии `afterEnter`:

```
override fun afterEnter(state: UsersState) {
    topGsMenuItems.setOf(
        GsMenuItem("Обновить") { refresh() },
        GsMenuItem("Выход")   { navigator.doLogout() }
    )

    bottomGsMenuItems.single("Добавить") { addUser() }
}
```

6.7 События навигации

- `onBack` — обработчик кнопки «Назад».

6.8 Расширение навигатора

Методы расширения навигатора объявлены в интерфейсе `NavigableVcp` и могут быть реализованы в контроллере представления.

Можно переопределить:

- `newScreen` — представление типа *VcpScreen*
- `getTitle` — заголовок
- `topGsMenuItems` — элементы `TopBar`
- `bottomGsMenuItems` — элементы `BottomBar`
- `showFAB` — видимость FAB
- `Fab` — собственная реализация `@Composable Fab()`
- `fabClick` — действия на FAB

Пример FAB

```
override fun showFAB() = usersORL.size > 0

override fun Fab() =
    SmallFloatingActionButton(onClick = ::addUser) {
        Icon(Icons.Default.PersonAdd, contentDescription = null)
    }
```

6.9 Стандартные диалоги

```
showBaseDialog(
    title = "Удалить пользователя?",
    okText = "Да",
    dismissText = "Нет",
    onOk = ::confirmDelete
)
```

6.10 Event Bus

`Flow`-шина событий работает на всё приложение:

```
subscribeEventBus<SyncDone>("SyncDone") {
    navigator.setNeedSync(false)
}
```

6.11 VcpScreen

Класс-адаптер, который отдаёт Composable-дерево как функцию `Content(pv: PaddingValues)`. Именно `screen` попадает в `Navigator` для реального рендеринга во вкладку стека. `PaddingValues` определяет размеры отступов для меню.

6.12 Стандартные делегаты

Делегаты типизируют стандартные события для обработки сообщений.

Стандартные делегаты:

- `QrCodeScannerDelegate`

Внимание

Экспериментальный функционал, находится в разработке.

6.13 Полный пример

```
class DemandListVci : SmStateVciAbst<DemandListState>() {

    private var demandORL = nullObservableRecordList
    val isRefreshing = mutableStateOf(false)

    /* 1. Init one time */
    override fun onInit(state: DemandListState) {
        demandORL = state.demandRS.observableRecordList
        onBack { smVci -> smVci.smPostBack() }
    }

    /* 2. Every enter */
    override fun afterEnter(state: DemandListState) {
        demandORL = state.demandRS.observableRecordList
    }

    /* 3. Exit */
    override fun beforeExit(state: DemandListState) = Unit

    /* 4. UI */
    override fun newScreen() = provideDemandListView(this)
    override fun getTitle() = "Заявки"

    /* 5. Actions */
    fun refresh() {
        showLoadingView()
        getSts<DemandListState>().postSharedTask("refresh") { st ->
            st.dbs.getApi(EamDemandApi::class).syncFromServer()
        }.onSuccess {
            stopLoadingView()
        }.onFailure {
```

(продолжается на следующей странице)

(продолжение с предыдущей страницы)

```
            stopLoadingView()
            showBaseDialog("Ошибка", it.message ?: "")
        }
    }
}
```

6.14 Хорошие практики

- Не держите ссылок на `RecordData`, всегда получайте новую через `observeAsState()`.
  Каждый раз там будет актуальная строка.
- Вызывайте только публичные методы VCI.
  Не трогайте `dbs` из Composable.
- Из VCI в `State` переходите через `postSharedTask()/postAsyncTask()`.
  Никогда не ходите в `State` напрямую.
- Держите бизнес-логику в `State`.
- Держите сетевые/SQL-операции — в `Api`/`Pkg`.
- Помните: VCI — это тонкий контроллер UI.
  Он должен оставаться ответственным за подписки, показ, навигацию и т.д.
  Изоляция VCI позволяет делать код простым, тестируемым и устойчивым к изменениям.
- Используйте собственные `ExecutorSubscription` для *Bluetooth*, *камеры*, *видео* и т.д. вместо `postAsyncTask`.
  Это снизит нагрузку на серверный поток.

6.15 Плохие практики

- Не злоупотребляйте методом `refreshView()`.
  Этот метод уже вызывается навигатором, ручной вызов нужен только в особых случаях.

7 Сессия

7.1 Транзакционный контекст

`GsSession` — компонент, отвечающий за транзакционную обработку данных. Содержит:

- контекст транзакции:
  - кэш, который закрепляет одну объект-строку на весь жизненный цикл транзакции;
  - список изменений
    При завершении транзакции изменения собираются в пачки `insert / update / delete`, которые сбрасываются в БД по 500 записей;
  - методы (`begin` / `commit` / `rollback`);
  - транзакционный кэш (`rowCache`);
  - списки изменений;
- Room-базу данных (`RoomDatabase`);

- методы для работы с сессией базы данных:
  - нативные запросы (`query*`);
- файловую подсистему (`FileManager`);
- инъекции зависимостей для контроллеров сессии (`Api` и `Pkg`).

Любой код, работающий с данными за пределами представления, стоит располагать в контроллерах сессии.
Это позволяет гарантировать целостность транзакций и отсутствие гонок.

7.2 Инъекция зависимостей

Контроллеры сессий создаются с автоматической инициализацией необходимых ссылок (инъекцией зависимостей).
Для управления используются следующие аннотации:

- `@GsApiBean`, `@GsPkgBean`, `@GsBean` — генерируют расширения вида `GsSession.getUserApi()` и фабрику `GlobalFactory`,
  избавляя разработчика от ручного Dagger/Hilt и позволяя создавать компоненты по имени без дополнительного кода.

Примечание

Изменение одного класса приводит лишь к перегенерации связанного файла, сборка проекта остаётся быстрой.

См. также

- модуль `di-processor`

7.3 Контроллеры сессий

Контроллер сессии создаётся один раз на сессию.

Api

Используется для организации бизнес-логики в разрезе сущности базы данных.
Содержит методы работы со строками таблицы:

- создание
  При этом автоматически генерируется идентификатор записи;
- удаление;
- обновление;
- загрузка;
- кэширование.

DbPkg

Используется для организации общей транзакционной бизнес-логики.

Выдача пакетов

```
val userApi = session.getApi(UserApi::class) // способ без генерации
val userApi = session.getUserApi()           // способ с генерацией
```

- При первом запросе объект создаётся, вызывается `enterSession()`, дальше он возвращается из кэша.
- `recreateApi()` удаляет старый экземпляр и тут же создаёт новый — удобно после `logout`.

7.4 Потоковая модель

Контроллеры сессий являются потоконебезопасными объектами, рассчитанными на работу в серверном потоке.

Внимание

Не пытайтесь обращаться к контроллерам сессии из главного потока.

7.5 Работа с транзакцией

- `flush()` — собирает изменения из всех `Api` и применяет их пачками по 500 строк.
- `commit()` — сохраняет изменения.
- `rollback()` — откатывает изменения и очищает кэш.

Работа с сырыми запросами

- querySingleMap

```
val map = session.querySingleMap(
    "SELECT * FROM User WHERE gid = ?",
    arrayOf(gid)
)
```

- querySingleData

```
// map → data-class
data class MiniUser(val name: String, val age: Int?)

val mu: MiniUser? = session.querySingleData(
    MiniUser::class,
    "SELECT sName AS name, nAge AS age FROM User WHERE id = ?",
    arrayOf("57")
)
```

8 Хранение данных

8.1 Реляционные данные

Для хранения реляционных данных используется СУБД `SQLite` и ORM `Room`.

Для подключения базы данных необходимо:

1. Подключить *зависимости в Gradle*.
2. Создать класс базы данных.
3. Зарегистрировать базу данных в `MainActivity`.
   В случае, если `MainActivity` унаследована от `GsBaseActivity`, добавьте в класс:

```
override fun provideDatabase(): PrototypeDatabase =
    RoomDbSingleton.getInstance(this, "PrototypeDatabase")
```

Класс базы данных

```
// db/AppDatabase.kt

@Database(
    entities = [SystemEntity::class, User::class],
    version = 1,
    exportSchema = false
)
abstract class AppDatabase : RoomDatabase() {
    abstract val userDao: UserDao

    companion object {
        @Volatile private var INSTANCE: AppDatabase? = null

        fun getInstance(ctx: Context): AppDatabase =
            INSTANCE ?: synchronized(this) {
                INSTANCE ?: Room.databaseBuilder(
                    ctx,
                    AppDatabase::class.java,
                    "AppDB"
                ).build().also { INSTANCE = it }
            }
    }
}
```

8.2 Файлы

Файлы сохраняются в каталоге приложения согласно настройке `DbConfig.dataPath`.

Для работы с файлами используется `FileManager` в `GsSession`.
Он позволяет организовать транзакционную работу с файлами.

Принципы работы:

- любые загруженные/созданные файлы сначала пишутся во временную область;
- при `commit()` — перемещаются в постоянную область;
- при `rollback()` — удаляются.

Таким образом БД и файлы остаются синхронными.

Работа с файлами

```
val localPath = takePhoto()                              // /cache/...
val draft     = dbs.fileManager.copyToSendDir(localPath)

gate.upload(draft)                                       // сервер принял → HTTP 200
dbs.commit()                                             // файл переносится в /data/
```

Если сеть упала — вызов `rollback()` удалит файл из черновиков и откатит БД.

9 Контроллер таблицы

Контроллер таблицы (`Api`) служит для транзакционной работы со строками базы данных.

`Api`-контроллер предоставляет:

- *кэш строк*:
  - один объект `Entity` — одна строка;
  - повторный `load(id)` вернёт тот же экземпляр;
- *список изменений* — изменения накапливаются до момента `flush()`;
- *`flush()`* — все изменения (`inserted / updated / deleted`) собираются в `GsSession.flush()` и пишутся пачками;
- *счётчик идентификаторов* — `insert()` получает уникальный `id` из счётчика.

Для создания контроллера:

1. Создайте базу данных.
2. Создайте сущность.
3. Создайте `Dao`.
4. Создайте `Api`-контроллер.
5. Скомпилируйте проект.
   При компиляции проекта произойдёт генерация необходимых классов.

9.1 Сущность

Сущность описывает таблицу базы данных в терминах ORM.
Для сущности генерируется `Dao`, а также создаётся таблица в базе данных.

Для создания сущности:

1. Создайте `data class`.
2. Унаследуйте класс от `BaseEntity`.

> **Совет**
>
> Не создавайте поле `id`, так как оно определено в `BaseEntity`.

3. Добавьте аннотацию `@Entity`.
4. Укажите в аннотации имя таблицы базы данных.

> **Внимание**
>
> Имя таблицы должно соответствовать имени класса.

5. Добавьте класс в список `entities` базы данных.

   Пример:

```
@Database(
    entities = [SystemEntity::class, User::class]
)
abstract class AppDatabase : RoomDatabase()
```

Пример сущности:

```
// db/User.kt
@Entity(tableName = "User")
data class User(
    var sName: String? = null,
    var nAge : Int?    = null
) : BaseEntity() {
    override fun copyEntity() = copy()
}
```

9.2 Dao

`Dao` содержит набор сгенерированных на основе сущности методов для реализации ORM.

1. Создайте абстрактный класс.
2. Унаследуйте его от `BaseDao`.
   В качестве типа укажите сущность для этого `Dao`.
3. Аннотируйте класс аннотацией `@Dao`.
4. Добавьте переменную с `Dao` в класс базы данных.

   Пример:

```
abstract class AppDatabase : RoomDatabase() {
    abstract val userDao: UserDao
}
```

Пример `Dao`:

```
// db/UserDao.kt
@Dao
abstract class UserDao : BaseDao<User>()
```

9.3 Создание Api

1. Создайте класс.
   Класс должен принимать параметр `ctx: ApiBeanContext`. Контекст нужен для работы механизма инъекции зависимостей.
2. Унаследуйте класс от `DbBaseApiGen`.
   `DbBaseApiGen` автоматически подставляет `dbRoom` и `dbSession` через DI-процессор.

```
@GsApiBean
class UserApi(ctx: ApiBeanContext)
    : DbBaseApiGen<User, UserDao, AppDatabase>(ctx) {

    override val dao get() = dbRoom.userDao
    override fun newEntity() = User()

    fun seedDemo() {
        // инициализация демо-данных
    }
}
```

9.4 Создание Api на основе рефлексии

1. Создайте класс.
2. Унаследуйте класс от `DbBaseApi`.

Внимание

Раздел находится в разработке.

9.5 Оптимизация производительности

- Делайте `insertAll(list)` через `Api` — это всегда batch без `forEach`.
- Используйте `rowCache` для реконфигурируемых фильтров:
  запросили 500 `id`, фильтруете в Kotlin-коде, не дёргая SQL повторно.

10 Кодогенерация

Кодогенерация позволяет автоматизировать рутинные операции.

10.1 DI-процессор

Автоматизирует работу с *инъекциями зависимостей в GsSession*.

Отвечает за генерацию:

- `Session-extensions`
  Файл `${Cls}SessionGen.kt` добавляет три метода к `GsSession`:
  `getCls()`, `destroyCls()`, `recreateCls()`.
- `GlobalFactory.kt`
  Для всех `@GsBean` создаётся объект-фабрика с thread-safe `ConcurrentHashMap`.
  Позволяет получить, пересоздать или обнулить singleton без DI-контейнера.
- `MainStorage.kt`
  Минимальное потокобезопасное хранилище для глобальных бинов.

Примечание

При необходимости можно добавить произвольный DI-фреймворк, например Dagger / Hilt.

Ключевые аннотации

Аннотация	Применяется к классу-бину	Жизненный цикл	Как вызывается
`@GsPkgB`	Package-класс, работающий внутри `GsSession`	Создаётся и кешируется на время сессии	`session.get<ИмяPkg>()` (сгенерированный extension)
`@GsApiB`	API-класс поверх Room	То же, что и `@GsPkgBean`	`session.get<ИмяApi>()`
`@GsBean`	Глобальный компонент (парсер, crypto-утилита и др.)	Singleton всего процесса	`GlobalFactory.get<Имя>()`

Пример

```
@GsApiBean
class UserApi(ctx: ApiBeanContext)
    : DbBaseApiGen<User, UserDao, AppDb>(ctx) { /* ... */ }

@GsPkgBean
class SyncPkg(ctx: PkgBeanContext) : DbPkg2(ctx)

@GsBean
class CryptoUtil   // не зависит от сессии


// Работаем в State или другом серверном коде

val api    = session.getUserApi()     // сгенерировано
val pkg    = session.getSyncPkg()
val crypto = GlobalFactory.getCryptoUtil()
```

Подключение к проекту DI

```
plugins {
    id("com.google.devtools.ksp") version "1.9.22-1.0.17"
}

dependencies {
    ksp(project(":di-processor"))
}
```

Правила создания

- Конструктор аннотированного класса должен принимать ровно один аргумент-контекст (`ApiBeanContext` или `PkgBeanContext`).
- Для singleton с побочным состоянием необходимо использовать `@GsBean` и управлять жизненным циклом вручную через `GlobalFactory`.

Диагностика ошибок

Сгенерированный код лежит в `$buildDir/generated/ksp/.../di/`.
При любой проблеме (цикличность, нарушение областей) KSP прерывает компиляцию и выводит понятное сообщение вида:
`Cycle detected: a.b.C -> x.y.D -> a.b.C`.

Исправьте зависимости — Gradle пересоберётся без перезапуска IDE.

10.2 Room-процессор

Отвечает за генерацию:

- базы данных;
- `Dao`.

Подключение к проекту Room

```
dependencies {
    ksp(libs.androidx.room.compiler)
}
```

10.3 Результат кодогенерации

Получается один JAR $<$ 1 МБ, не требующий рефлексии и не трогающий рантайм,
что позволяет легко обфусцировать код.

11 Плагин активити

Activity Plugin — это автономный модуль, который встраивается прямо в `GsBaseActivity`, добавляя ей новую функцию (NFC-сканер, BLE-монитор, WebRTC-звонки и т.п.) без изменений в бизнес-логике экранов и без пересборки ядра GMF.

К плагинам активити можно получать доступ из главного потока.

Базовые плагины (камера, NFC, QR-сканер) включены «из коробки» и будут пополняться; разработчик может добавить собственный, реализовав тот же интерфейс.

11.1 Мотивация

- Изолировать платформенные API: весь код работы с NFC, камерой или сенсорами живёт в плагине, а приложение общается только через колбэк-интерфейсы.
- Повторно использовать логику: один плагин можно подключить сразу во многие приложения.
- Синхронизировать жизненный цикл: плагин получает те же события (`onCreate/Resume/Pause...`), что и `Activity`, и не нарушает транзакции State Manager'а.

11.2 Контракт

```
interface ActivityPlugin : IPkg {

    /* ———————— 1. Инициализация ———————— */

    /** Один раз до показа UI. Можно выполнять работу в IO-диспетчере. */
    suspend fun initializeAsync(activity: GsBaseActivity<*>) {}

    /** Вызывается сразу после инициализации UI. */
    fun onInitializeUi(activity: GsBaseActivity<*>, navigator: Navigator) {}


    /* ———————— 2. Жизненный цикл ———————— */

    fun onCreate (activity: GsBaseActivity<*>) {}
    fun onStart  (activity: GsBaseActivity<*>) {}
    fun onResume (activity: GsBaseActivity<*>) {}
    fun onPause  (activity: GsBaseActivity<*>) {}
    fun onStop   (activity: GsBaseActivity<*>) {}
    fun onDestroy(activity: GsBaseActivity<*>) {}

    /** Activity получила новый Intent (например, NFC-метку). */
    fun onNewIntent(activity: GsBaseActivity<*>, intent: Intent) {}
}
```

11.3 Подключение плагина в Activity

```
class MainActivity : GsBaseActivity<AppNavigator>() {

    override fun provideModules(): List<KClass<out ActivityPlugin>> =
        listOf(MyAnalyticsPlugin::class)
}
```

- **Тип-список** — передаёте `KClass`; экземпляр будет лениво создан через `session.getSimplePkg()`. Так удобнее, если плагин в разных модулях и нужен DI-контейнер.

11.4 Порядок инициализации

1. `GsBaseActivity` собирает `StateManager` и `Navigator`.
2. Вызывает `plugin.initializeAsync()` **параллельно** с инициализацией `StateManager`'а; здесь выполняются тяжёлые операции (запуск CameraX, чтение лицензий и т.п.).
3. UI создан — `onInitializeUi()` даёт плагину объект `Navigator`.
4. Далее события `onCreate/Start/Resume...` приходят в том же порядке, что и в `Activity`.

11.5 Обмен данными с экраном

- Плагин хранит публичные `MutableStateFlow` / `Callback`-свойства — `ViewModel` или VCI подписывается и получает данные.
- Если нужно уведомить только текущий экран, проще всего передать лямбду-делегат при вызове (в `afterEnter`), как показано в NFC-примере, либо получить плагин через сессию.

11.6 Работа с Navigator

В `onInitializeUi()` плагин может:

- показать диалог через `navigator.createDialog(...)`;
- запустить экран через `navigator.setBaseMenu()` или вставить `SmTrans` в текущий стек.

Главное — не обращаться к UI до этой фазы: до создания `Navigator` компоновка ещё не готова.

11.7 Пример NFC-плагина

```
class NfcActivityPlugin : ActivityPlugin {

    private val isWriteNow = AtomicBoolean(false)
    private var adapter: NfcAdapter? = null
    var onRead: (String) -> Unit = {}

    override suspend fun initializeAsync(activity: GsBaseActivity<*>) {
        adapter = activity.getSystemService(NfcManager::class.java).defaultAdapter
        setupForegroundDispatch(activity)
    }

    override fun onResume(activity: GsBaseActivity<*>) {
        adapter?.enableForegroundDispatch(activity, pendingIntent, filters, null)
    }

    override fun onPause(activity: GsBaseActivity<*>) {
        adapter?.disableForegroundDispatch(activity)
    }

    override fun onNewIntent(activity: GsBaseActivity<*>, intent: Intent) {
        if (!isWriteNow.get()) {
            val payload = readNdef(intent)
            onRead(payload)
```

(продолжается на следующей странице)

(продолжение с предыдущей страницы)

```
        }
    }

    /* остальной код - чтение, запись, шифрование */
}
```

- Плагин полностью управляет `NfcAdapter`, не нагружая `Activity`.
- В `onRead` передаёт данные обратно в экран (VCI) или `State`.

Activity-плагины позволяют подключать новые аппаратные возможности так же легко, как dependency в Gradle. Всё управление происходит в одном месте, жизненный цикл синхронизирован, а бизнес-код экранов остаётся чистым.

12 Дополнительные контроллеры

12.1 Pkg

Используется для организации бизнес-логики, не требующей прямого доступа к сессии.

Интерфейс	Роль
`GsSession`	Контракт; доступен везде через DI или `StateManager`.
`GsSessionImpl`	Единственная реализация; создаётся внутри `StateManager`.
`IApi` / `IDbPkg`	Сессионные *singletons*; живут до закрытия текущей сессии.
`IPkg`	Пакеты без доступа к БД; кэшируются отдельно от сессионных компонентов.

Внимание

Экспериментальный функционал, находится в разработке.

13 Приложение А

13.1 Ключевые усложнения нативного SDK

- Несогласованные фоновые операции
  REST-запрос стартовал в `IO`, запись в SQLite — в `Default`, обратное уведомление пришло в `mainThread`. Малейшая задержка вызывала гонку и слепой интерфейс.
- Сложная транзакционность
  Часть данных уже записана в БД, часть файлов загрузилась, а сеть оборвалась. Вручную откатить всё — отдельная задача со своими багами.
- Boilerplate
  Room → Repository → UseCase → ViewModel → LiveData → Compose. Каждая сущность дублирует набор полей — правка трёх строк превращалась в десять правок и pull-request на 200 строк.

- Непредсказуемый офлайн
  За очередь задач отвечали отдельные сервисы, и как только бизнес-логика усложнялась, синхронизация также усложнялась и появлялись трудноуловимые баги.
- Трудная интеграция железа
  Камера, NFC, голосовой ввод требовали кастомных сервисов, контекстных разрешений и многословного DI-кода.

Детальное описание преодолённых усложнений

Усложнение	Решение, предоставляемое GlobalErp Mobile Framework
Последовательное выполнение долгих операций блокирует UI	Вся логика БД, сетевые вызовы и файловые операции выполняются в едином серверном потоке. Поток UI занят только отрисовкой.
Race condition между результатами запросов	Подписка осуществляется на `ObservableRecord`; события доставляются строго в порядке фиксации транзакции.
Разрозненные реализации retry-логики для REST, файлов и локальной БД	`Pkg`, `Api`, `FileManager` работают внутри той же транзакции *GsSession*. `commit` / `rollback` выполняются единообразно.
Сложность атомарного отката: часть данных уже записана, часть — нет	`GsSession` объединяет БД, файлы, сеть и любые пакеты, созданные пользователем: единая транзакция обеспечивает согласованный откат.
Отмена длительной операции оставляет систему в непредсказуемом состоянии	`InterruptingLock` прерывает SQL, REST и вычисления; стек состояний автоматически откатывается к стабильному экрану.
Навигация разбросана между `Activity`, флагами back stack и коллбэками	Стек-ориентированная навигация (`SmState` + `SmTrans`) управляется на серверном потоке.
Решения о переходах принимаются до синхронизации с UI.	
Передача данных между слоями требует множества Mapper / Repository / ViewModel	Цепочка *SQL* → *RecordSet* → *Compose* избавляет от промежуточных слоёв. Данные приходят в UI напрямую.
Хрупкий офлайн-режим, необходимость писать очередь синхронизации	Все изменения фиксируются локально; синхронизация с сервером оформляется единообразно через очереди задач и `Flow`, без разрозненных сервисов.
Сложная интеграция аппаратных функций (камера, NFC, QR-сканер)	Плагины `ActivityPlugin` подключаются в `GsBaseActivity`. Дополнительных DI-конфигураций не требуется.
Несогласованная обработка аппаратной кнопки «Назад» и жестов	`Navigator` централизованно управляет `TopBar`, `BottomBar`, FAB и back-навигацией для всех экранов.
Невозможность гарантировать порядок выполнения фоновых задач при навигации	Задачи (`postSharedTask`, `postAsyncTask`) регистрируются в очереди событий State Manager. Выполняются строго после завершения предыдущих транзакций.
Отсутствие единых правил для тестирования	`State` / `VCI` можно запускать с параметром `database = null`; достаточно