форсайт.

Документация к продукту

«Форсайт ФМП. Мобильная платформа. Jvm»

25.03.001

Автор: Дмитрий Воронин
dd.voronin@fsight.ru

Санкт-Петербург

2025

Содержание

Изменения в релизе 25.03.001

* Исправлено поведение метода FMPUser.auth() в офлайне, когда просрочен токен. Теперь он возвращает корректно UnathorizedException.

* FMPQuery теперь корректно загружает ресурсы в разных потоках.

* Добавлена подробная статистика работы методов. Активируется на уровне логирования Инфо в политике логирования.

* Добавлен новый CertException для обработки ошибок сертификата сервера и SSL Pinning.

* Метод FMPDatabase.attach() теперь принимает список вместо vararg. Старый метод отмечен deprecated.

* Также добавлен UserDisabledException, сигнализирующий о заблокированном пользователе.

* Переработана документация. Теперь она полностью в стиле компании. Добавлены кликабельные ссылки на Dokka для всех методов. Обновлены примеры.

* Теперь при возможности используется SecureRandom(). Также добавлен метод FMP.Builder.strictSecurityRequirements(), который приведёт к аварийной остановке приложения, если физическое устройство не поддерживает безопасный SecureRandom().

* Добавлен метод FMPUser.getTokenExpiration(), возвращающий окончание времени жизни Refresh токена. Это, в свою очередь, позволяет определить валидность аутентификации пользователя в офлайне.

* FMPFile теперь поддерживает работу с InputStream и OutputStream. Для этого потребуется передать их в Builder при инициализации файла вместо указания пути на файловой системе.

Работа с технической документацией Dokka

Вместе с фрейморком также поставляется техническая документация Dokka. Её задача отличается от текущей документации. Если здесь разобраны кейсы с примерами их реализации, то в Dokka уже содержится подробная информация по конкретным методам, их параметрам и результатам работы. Рекомендуется сначала ознакомиться с этой документацией, а к документации Dokka обращаться в процессе разработки при возникновении вопросов по работе с методами.

Её можно найти внутри архива-дистрибутива в директории dokka. Представляет собой связанные HTML страницы. Для начала работы откройте файл dokka/index.html в вэб-браузере. Также появилась онлайн-версия с документацей для последней версии фреймворка: fmp-framework.fsight.ru. Документация для предыдущих версий доступна на Maven сервере.

Далее выберите интересующий Вас package, например, ru.fsight.fmp для просмотра основных объектов FMP. Таких, как FMP или FMPUser. Затем выберите интересующий Вас объект. При помощи навигации на странице просмотрите доступные типы, свойства, примеры и ссылки на связанные объекты.

Начало работы с фреймворком

Сама идея фреймворка заключается в уменьшении трудозатрат на популярные задачи разработки. Разработчику не нужно решать вопрос аутентификации, работы с СУБД или облачным хранилищем файлов.

Фреймворк целиком состоит из высокоабстрактных методов, таких как FMPUser.auth() для аутентификации, FMPDatabase.query() для запросов в локальную БД и многих других.

Клиент посылает запросы на сервер через вышеупомянутые методы, сервер же возвращает ответ, который в последствии преобразуется в удобные для работы объектные модели. Объекты FMP наследуются от FMPObject и имеют свойства и методы. Обращение к свойствам не затратно по ресурсам — время работы такой операции всегда безопасно к выполнению в любом потоке. Методы же являются какой-либо операцией, требующей времени на обработку. Методы стоит вызывать в фоновых потоках.

Точкой входа в библиотеку является FMP.Builder, в котором производится инициализация параметров фреймворка. Он же в свою очередь возвращает объект FMP, который создаёт объекты FMPObject для взаимодействия с библиотекой. Про них будет рассказано подробнее в разделе Инициализация FMP.

Также все методы возвращают не просто результат работы, а обёртку FMPResult<T>, которая содержит в себе следующие свойства:

1. status: Boolean — статус выполнения операции. Была ли операция выполнена до конца без ошибок.

2. exception: FMPException? — содержит текст с описанием ошибки, если таковая имеется. Не рекомендуется полагаться на наличие ошибки. Для проверки успешности операции стоит смотреть на status, и если статус операции равен false, то можно смотреть на текст ошибки.

3. response: FMPResponse? — HTTP ответ сервера, если выполнялся запрос к серверу (помогает в отладке).

4. result: T — результат операции. Отличается для конкретного метода. Например, для метода аутентификации FMPUser.auth() результатом будет Boolean, обозначающая статус аутентификации. В случае с методом аутентификации status будет false, если не удалось выполнить запрос на сервер или на сервере произошла ошибка. При этом он может быть

равен true, если запрос на сервер был успешен, но пользователь не был аутентифицирован. В таком случае result будет false, ведь результат аутентификации отрицательный. Поведение status и result могут совпадать и могут различаться для каждого отдельного метода. В общем случае рекомендуем полагаться на результат result. Подробности для каждого метода можно получить в документации Dokka.

Дополнительно класс FMPResult имеет методы-помощники: onSuccess(), onError() и onResult(). onSuccess() срабатывает в случае успеха, onError() в случае ошибки и onResult() сработает в любом из случаев. Далее пример использования:

```
user.auth()
  .onSuccess {
    /* Запрос был успешен. */
    val result = it
  }
  .onError {
    /* Произошла ошибка. */
    val exception = it
  }
  .onResult {
    /* Не важно, была ли ошибка. */
    /* Мы просто хотим результат. */
    /* В случае ошибки он пустой (значение по умолчанию). */
    val maybeResult = it
  }
```

Работа с Maven

Maven является стандартом распространения библиотек на платформе Jvm. Если Вы являетесь разработчиком Jvm, то наверняка уже работали с Maven, когда добавляли разного рода зависимости в проект. Для работы с FMP потребуется указать зависимость, как и с любой другой библиотекой. Единственное отличие в том, что FMP не распространяется через публичные Maven сервера. Поэтому дополнительно потребуется указать адрес публичного репозитория Форсайт.

Далее приведены примеры как для языка Groovy (.gradle), так и Kotlin (.gradle.kts). Выбирайте пример в соответствии с расширением файлов в Вашем проекте.

Добавление репозитория в директории проекта:

settings.gradle (Groovy)

```
dependencyResolutionManagement {
  repositories {
    maven { url "https://fmp-maven.fsight.cloud/" }
  }
}
```

settings.gradle.kts (Kotlin)

```
dependencyResolutionManagement {
  repositories {
    maven("https://fmp-maven.fsight.cloud/")
  }
}
```

После этого добавить зависимость в модуль проекта в директории app. Полный список доступных версий по ссылке. Также доступна версия DEV-SNAPSHOT, которая является бета-версией и обновляется каждые сутки.

app/build.gradle (Groovy)

```
dependencies {
  implementation "ru.fsight.fmp:jvm:25.03.001"
}
```

app/build.gradle.kts (Kotlin)

```
dependencies {
  implementation("ru.fsight.fmp:jvm:25.03.001")
}
```

После этого в проекте должен быть доступен интерфейс FMP и остальные объекты фреймворка.

Инициализация FMP

Перед непосредственной работой с моделями фреймворка необходимо провести его инициализацию. Она производится через класс FMP.Builder. Далее приведён пример базовой инициализации:

```
val device_id: String = "DEVICE_ID"

val fmp: FMP = FMP.Builder()
  .address("https://fmp.example.com")
  .environment("test")
  .project("test")
  .api(FMP.API.V2)
  .deviceID(device_id)
  .storage(context.filesDir.absolutePath)
  .username("test")
  .build()
```

В примере выше мы работаем с данными пользователя, указанного в username. Также мы получаем ID устройства, создаём класс FMP.Builder, далее цепочкой вызовов указываем: адрес сервера FMP, среду проекта и сам проект, версию API сервера из константы в FMP, ID устройства и путь к рабочей директории фреймворка. После этого вызываем метод build(), который возвращает нам реализацию интерфейса FMP. Такой формат инициализации называется Builder и встречается повсеместно во фреймворке. Он позволяет расширять функционал объектов не меняя при этом семантику (аргументы) методов. То есть при появлении нового функционала, для его использования будет достаточно добавить вызов новых методов без модификации существующих методов.

Пока мы привели пример базовой инициализации - только необходимые методы. Полный список доступных методов можно найти в документации Dokka. Также некоторые дополнительные методы будут указаны дальше в реализации кейсов.

Аутентификация пользователя по паролю

Пользователя возможно аутентифицировать двумя способами: при помощи пароля, либо при помощи сохранённого токена. Сейчас рассмотрим аутентификацию первым методом, второй вариант рассмотрен в статье Аутентификация пользователя по токену.

Аутентификация таким методом требует от пользователя ввод его учётных данных в fmp.user.auth(). Важно отметить, что фреймворк не сохраняет учётные данные пользователя, а только токен, который к нему привязан.

Далее необходимо выполнить аутентификацию. Для этого вызывается метод auth(). После этого хорошо будет проверить результат аутентификации и поступить соответствующе.

```
val auth: FMPResult<Boolean> = user.auth("password")

if (auth.result) {
  nextScreen()
} else {
  showError(auth.exception)
}
```

Если аутентификация была успешна — переводим пользователя на следующий экран. В случае ошибки — показываем пользователю текст ошибки.

После успешного выполнения метода auth() от разработчика для дальнейшей работы ничего не требуется. Фреймворк полностью самостоятельно занимается работой с токенами и авторизацией запросов. Работа с токенами представлена в следующем разделе.

Аутентификация пользователя по токену

Помимо аутентификации по учётным данным пользователя возможна аутентификация по токену. Всем управлением токенами занимается сам фреймворк. От разработчика требуется лишь подстроить логику приложения на их использование.

Для использования токена его необходимо для начала получить. Получение токена происходит после одного успешного вызова метода auth() с паролем. То есть для аутентификации по токену требуется хотя бы раз аутентифицироваться по паролю. Про аутентификацию по паролю рассказано в разделе Аутентификация пользователя по паролю.

Логика аутентификации с токеном предельно проста:

1. Не передавая пароль вызвать метод auth(), например, на экране Splash.

2. В случае, если результат метода положительный, то можно смело продолжать работу с фреймворком — аутентификация была успешна.

3. В случае, если результат метода отрицательный, то пользователя следует направить на экран аутентификации по учётным данным.

Далее пример простейшей логики попытки аутентификации по токену:

```
val auth: FMPResult<Boolean> = user.auth()

if (auth.result) {
  navigateMainScreen()
} else {
  navigatePasswordAuthScreen()
}
```

Хранилище пользователя

Фреймворк позволяет указать путь к рабочей директории через метод FMP.Builder.storage(). Все файлы фреймворка будут расположены по указанному пути.

У каждого пользователя имеется своя директория. Помимо конфигурации фреймворка, для разработчика предоставлены две возможные директории для записи: кэш и хранилище. В кэш разработчик записывает временные кэш-файлы. В хранилище записываются файлы постоянного хранения.

Получить полный путь к папке с кэшем можно через свойство FMPUser.cache. Удалить данные кэша возможно при помощи метода FMPUser.wipeCache(). Аналогично для хранилища: FMPUser.storage — полный путь к хранилищу, FMPUser.wipeStorage() — удаление данных в хранилище.

Отдельно можно выделить метод FMPUser.wipe() — он удаляет все данные пользователя, включая токены, логи и конфигурацию фреймворка.

Далее пример работы с кэшем. Аналогично для хранилища.

```
val file = File("${user.cache}/file.txt")

/* Делаем что-то с файлом. */

user.wipeCache() // Удаляем весь кэш пользователя.
```

Работа с локальной базой данных

Работа с базой данных происходит через объект FMPDatabase. Как и с любым другим объектом, сначала необходимо его инициализировать. Для инициализации нам потребуется только путь к файлу базы данных.

```
val database: FMPDatabase = fmp.database
  .path("${user.cache}/database.db")
  .build()

val open: FMPResult<Boolean> = database.open()

if (open.result) {
  database.query("DROP TABLE User;")
}
```

Содержимое базы данных надёжно зашифровано. Получение данных вне установленного приложения на конкретном устройстве не представляется возможным. На время разработки есть возможность отключить шифрование баз данных. Подробнее об этом рассказано в статье Отладка работы фреймворка.

Организация объектов ресурсов и таблиц

Так как ресурсы в подавляющем числе случаев являются постоянными и известны на момент компиляции приложения, хорошей идеей будет сделать их статичными во избежание дублирования кода. Для этого можно создать отдельный класс, в котором будут храниться все ресурсы и таблицы. Для добавления или изменения параметров запросов можно воспользоваться методами копирования и редактирования объектов, описанными в разделе Копирование и модификация объектов.

```
class Resources(val fmp: FMP)
{
  val res1: FMPResource = fmp.resource
    .name("res1")
    .build()

  val res2: FMPResource.Builder = fmp.resource
    .name("res2")
}

class Tables(val resources: Resources)
{
  val table1: FMPTable = fmp.table
    .resource(resources.res1)
    .name("output_table")
    .build()

  val table2: FMPTable.Builder = fmp.table
    .resource(resources.res2)
    .name("output_table")
}

/* Загрузить данные ресурсов вручную. */
resources.res1.rebuild().params("...").build().download()
resources.res2.params("...").build().download()
```

Работа со схемой ресурсов

Возможно динамически узнавать о доступности ресурсов и их свойствах. Для этого есть метод FMPUser.getResources(), который вернёт нам список доступных ресурсов.

В случае, если мы узнали, что схема ресурсов на сервере была изменена во время работы приложения, возможно повторно загрузить схему вызовом метода FMPUser.downloadResources(). По умолчанию метод вызывается автоматически после каждой аутентификации пользователя.

```
fmp.user
  .downloadResources()
  .onSuccess {
    val resources: List<FMPResource> = fmp.user
      .getResources()
      .result
  }
```

Настройка ресурса

Основными параметрами настройки ресурса являются name() и params(). Метод name() указывает название ресурса, определённое на сервере платформы. И params() указывает RPC-параметры ресурса, которые также определяются сервером. Получить название ресурса и его параметры можно у администратора платформы или через метод FMPUser.getResources().

Также есть ряд необязательных, но очень важных методов. Их использование напрямую зависит от реализации ресурса на сервере, и только поэтому они не используются по умолчанию. Первый такой метод называется delta(). Его использование позволяет значительно сокращать трафик и время ожидания загрузки, ведь с ним при запросе клиент не будет получать все данные заново, а лишь разницу между последней загруженной версией. Для использования delta требуется, чтобы ресурс был кэшируемый на сервере.

Метод filter() указывает на поддержку фильтрации. В случае, если фильтрация поддерживается ресурсом, при вызове метода FMPQuery.download() будут загружены только те данные, которые требуются для удовлетворения запроса. Например, если мы используем фильтр «WHERE id = 10», то будут загружены только строки, у которых id равен 10.

Метод cacheByParams() указывает фреймворку, что к названию ресурса следует добавить также хэш-сумму его параметров. Это полезно, например, когда мы получаем данные из одного ресурса, но они отличаются в зависимости от передаваемых RPC-параметров. Работает только локально и не требует настройки на сервере.

```
val resource: FMPResource = fmp.resource
  .name("MyResource")
  .params("{\"upsert_rows\": null, \"delete_ids\": null}")
  .build()
```

Настройка таблицы

FMPTable описывает таблицу, принадлежащую ресурсу. У таблицы нет собственных методов, но она выполняет роль объектной моделью, связывая название таблицы и ресурс.

```
val table: FMPTable = fmp.table
  .resource(resource)
  .name("output_table")
  .build()
```

Получение данных ресурса через FMPQuery

Класс FMPQuery служит комплексной обёрткой для нескольких объектов FMP. Его задача — удовлетворять скомпилированный SQL SELECT запрос разработчика максимально эффективно. Так, например, если FMPResource.isDelta равен true, то FMPQuery будет загружать только разницу в данных, а не все данные целиком. Или FMPResource.isFilter равен true, тогда FMPQuery отфильтрует запрос и загрузит только данные, которые требуются для запроса, например, отфильтрованные через where().

Запрос составляется аналогично обычному SQL запросу, за исключением того, что вызываются соответствующие методы в FMPQuery.Builder вместо цельной строки.

```
val query: FMPQuery = fmp.query
  .select("$table1.id AS id, $table2.data AS data")
  .from(table1)
  .join(table2)
  .on("$table1.id = $table2.user")
  .where("$table1.status = 1")
  .order(mapOf("$table1.id" to FMPQuery.Order.ASC))
  .build()

val result: FMPResult<Boolean> = query.download()
val data: FMPResult<List<Map<String, String>>> = query.get()
val result: List<MyClass> = mutableListOf()

for (item in data) {
  result.add(MyClass(id = item["id"], data = item["data"]))
}
```

Получение данных ресурса вручную

В большинстве случаев рекомендуется составлять запрос через FMPQuery, как описано в разделе Получение данных ресурса через FMPQuery, и хранить объекты ресурсов статично, как описано в статье Организация объектов ресурсов и таблиц.

Дополнительно есть возможность при необходимости загрузить отдельный ресурс целиком. Или, например, вызвать процедуру, игнорируя данные, как описано в разделе Модификация данных ресурса. Для загрузки ресурса необходимо вызвать метод FMPResource.download(), который закэширует данные всего ресурса.

Модификация данных ресурса

Редактирование данных на сервере происходит за счёт RPC (Remote Procedure Call). Простыми словами — удалённый вызов функции. Как и любая функция, RPC принимает параметры. В нашем случае это строка. И через этот параметр и происходит редактирование данных при запросе ресурса.

RPC не является универсальным для каждого ресурса. Функции пишутся разработчиками на сервере FMP. И в зависимости от реализации могут быть различные параметры для получения и редактирования данных. Но для примера используем Локальную БД на сервере, у которой есть заранее прописанная функция RPC.

Параметры RPC запроса задаются в методе FMPResource.Builder.params().

Локальная БД принимает два параметра в формате Json: delete_ids и upsert_rows. delete_ids принимает массив id строк, которые нужно удалить. upsert_rows принимает массив массивов данных строк.

Например, чтобы удалить строки с id равными 3 и 4, наши передаваемые в params() параметры будут выглядеть так:

```
{ "delete_ids": [ [ 3 ], [ 4 ] ], "upsert_rows": null }
```

Для добавления данных в таблицу, содержащую колонки: id и name:

```
{
  "delete_ids": null,
  "upsert_rows": [ [ 1, "name1" ], [ 2, "name2" ] ]
}
```

В случае, если нам достаточно отправить данные, при этом не загружая то, что вернёт нам сервер – существует метод FMPResource.rpc(), который выполнит запрос и проигнорирует данные от сервера.

Работа с транзакциями

Транзакции очень полезны в сетях с плохим покрытием беспроводной сети. Они позволяют избежать дублирования запросов на изменение данных на сервере. Например, если Вы отправили на сервер запрос на увеличение какого-либо счётчика, а в процессе работы оборвалась связь и не было успешного ответа от сервера, но при этом сервер уже получил и обработал запрос — в такой ситуации помогают транзакции.

Их работа заключается в простом принципе привязки данных к запросу. Когда мы передаём данные с присвоенным uuid транзакции, то сервер запоминает, была ли выполнена операция. В случае, если сервер уже обработал запрос с таким id транзакции, он не станет повторно его обрабатывать.

Работать с транзакциями очень просто. Для начала нам потребуется создать модель транзакции:

```
val transaction: FMPTransaction = fmp.transaction
  .resource(resource)
  .build()

val uuid = transaction.uuid
```

Затем мы можем использовать её в моделях FMPQuery и FMPResource:

```
fmp.query
  .transactions(listOf(transaction))
  .build()
  .download()

resource.download(transaction)
```

Для повторной попытки запроса точно также повторяем запросы.

Запросы к Web-ресурсам

Кейс, когда платформа выступает в роли прокси-сервера для внешнего ресурса. Позволяет проксировать HTTP запросы через определённый ресурс на сервере.

Подробности конфигурации ресурса можно узнать у администратора платформы. Ключевые параметры запроса: method() и resource(). Также есть возможность указать headers() — HTTP заголовки, path() — путь у ресурса (для примера http://example.com/foo, путь будет /foo) и data() — данные для передачи.

Пример выполнения и обработки запроса:

```
val web = fmp.web
  .resource(resource)
  .path("/user")
  .method(FMPWeb.Method.POST)
  .data("{\"key\":\"value\"}".toByteArray())
  .headers(mapOf("Accept" to "application/json"))
  .build()

val response: FMPWeb.Response = web.request()
if (response.status.isOK) {
  foo(response.result)
}
```

Работа с файлами

Все взаимодействия с файлами выполняются через FMPFile. Директории тоже попадают в этот объект с единственным отличием — флаг isDirectory для них true. Также FMPFile содержит некоторые методы, специфичные только для файла или директории.

Содержимое файла надёжно зашифровано. Получение данных вне установленного приложения на конкретном устройстве не представляется возможным. Имеется возможность отключения шифрования для конкретных файлов при помощи метода encryption() при инициализации файла. Также на время разработки можно отключить шифрование глобально. Подробнее об этом рассказано в разделе Отладка работы фреймворка.

Простейший кейс: загрузка файла с сервера. Для этого нам необходимо инициализировать объект через Builder и указать три параметра: точка монтирования на сервере; путь к файлу на сервере; путь на локальной файловой системе, куда будет сохранён файл. Далее вызвать метод download() для загрузки файла и затем метод read() для получения содержимого файла.

```
val file: FMPFile = fmp.file
  .mount("local_storage")
  .remote("/test.txt")
  .local("${user.cache}/test.txt")
  .build()

val download: FMPResult<Boolean> = file.download()
if (download.result) {
  showFile(file.read())
}
```

Для выгрузки файла на сервер существует метод upload():

```
val file: FMPFile = fmp.file
  .mount("local_storage")
  .remote("/test.txt")
  .local("/path/to/test.txt")
  .build()

val upload: FMPResult<Boolean> = file.upload()
```

Помимо этого возможно использовать InputStream и OutputStream, что позволит избежать записей на файловую систему. Для этого задействованы методы билдера input() и output().

```
val buffer = ByteArrayOutputStream()
val file: FMPFile = fmp.file
  .mount("local_storage")
  .remote("/test.txt")
  .output(buffer)
  .build()

val download = file.download()
val data = buffer.toByteArray()
val string = data.decodeToString()
```

```
val inMemData = ByteArrayInputStream(byteArray)
val file: FMPFile = fmp.file
  .mount("local_storage")
  .remote("/test.txt")
  .input(inMemData)
  .build()

val upload = file.upload()
```

Push-уведомления

На платформе возможна организация push-уведомлений для приложений. Уведомления приходят от указанных провайдеров, таких как Firebase для Android или Apple Cloud для iOS. Платформа собирает таких провайдеров в одном месте и рассылает уведомления подписанным клиентам. Для подключения к пушам платформы достаточно добавить токен и подписаться на топик.

Рассмотрим пример, когда мы хотим подписаться на топик, чтобы начать получать уведомления. Для этого нам потребуется Firebase токен устройства. Имея токен, мы отправляем его на сервер и подписываемся на интересующий нас топик. Даже если не подписаться на топик, то мы всё ещё можем получить уведомление по имени пользователя.

```
val push: FMPPush = fmp.push
  .token(firebaseToken)
  .build()

val submit: FMPResult<Boolean> = push.submitToken()
if (submit.result) {
  push.subscribe(listOf(topic))
}
```

Логирование на сервер

За функционал логирования отвечает объект FMPLog. Он является синглтон и не может быть изменён. Также его не требуется инициализировать — инициализацией занимается сам фреймворк, получая конфигурацию от системы политик с сервера. Разработчику нужно лишь вызывать методы логирования тогда, когда это требуется. За остальное отвечает администратор проекта на сервере FMP.

```
fmp.log.info("foo()")

val result = bar()
if (result) {
  foobar()
} else {
  fmp.log.error("foo() : bar() failed")
}
```

Работа с исключениями

Исключения, они же Exception, являются стандартной частью языка. Чаще всего распространено, что вызываемые методы «бросают» исключения. Такие методы нужно вызывать внутри блока try-catch. В ФМП результатом работы метода является FMPResult<T>. Поэтому методы фреймворка не «бросают» исключения, а содержат их внутри такого результата.

Все исключения наследуются от FMPException. Так вы можете отделять исключения фреймворка от остальных Exception. Исключения дополнительно разделены на ошибки клиента и сервера: ClientException и ServerException.

Все исключения содержат в себе следующие поля:

1. message: String – краткое описание ошибки.

2. cause: Exception? – если ошибка фреймворка вызвана какой-либо внешней ошибкой, то она будет передана в этом поле.

3. help: String – краткое описание наиболее частых причин возникновения ошибки и способы её устранения.

Для ServerException дополнительно доступно два поля:

1. code: Int – код ошибки сервера.

2. raw: String – необработанный текст ошибки от сервера.

По умолчанию все методы могут возвращать StateException в случае некорректной настройки. Методы, которые выполняют запрос к серверу, могут дополнительно возвращать ServerException или AuthException. Также есть специальная ошибка - UnknownException, которая означает неизвестную фреймворку ошибку и может появиться при вызове любого метода. В случае возникновения подобной ошибки стоит обратиться в техническую поддержку.

Полный список доступных исключений с их описанием доступен в документации Dokka по пути ru.fsight.fmp.exception.

Пример обработки исключений:

```
val result: FMPResult<Boolean> = fmp.user.auth()

when (result.exception) {
  is UnauthorizedException -> showWrongPasswordError()
  is NetworkIOException -> showNoConnectionError()
  is ServerException -> showServerUnavailableError()
  is FMPException -> showUnknownError()
}
```

Пример ручного «бросания» исключения:

```
val result: FMPResult<Boolean> = fmp.user.auth()
result.exception?.let { throw it }
```

Безопасная разработка

Фреймворк забирает на себя ответственность за безопасность работы мобильного приложения. Тем не менее, возможно как сделать работу ещё более безопасной, так и использовать некоторые инструменты неправильно.

Самая большая рекомендация - выключать FMP.Builder.debug* методы вне разработки и тестирования. Лучше всего привязать их к типу сборки, чтобы случайно не забыть их отключить. Таким образом, они будут задействованы только в debug сборке, а в release отключены. Сделать это можно вот так:

```
val fmp: FMP = FMP.Builder
  .debug(BuildConfig.DEBUG)
```

Помимо этого существует метод инициализации strictSecurityRequirements(). Его активация вносит требования к среде выполнения. И если среда выполнения несоответствует строгим требованиям безопасности - работа приложения будет остановлена через выброс SecurityRequirementsException. На данный момент требование одно - чтобы устройство поддерживало SecureRandom(). В будущем могут появиться дополнительные требования.

Обновление фреймворка

Обновление мобильного фреймворка в рамках одного релиза содержит в себе только исправления ошибок. Например, с версии 23.12.*001* на версию 23.12.*004* можно обновиться без изменений в поведении фреймворка. Обновления релиза фреймворка, например, с версии 23.05 на версию 23.12 стоит выполнять вместе с сервером FMP на такой же релиз. При небольшой разнице версий некорректное поведение маловероятно, но возможно. В общем случае тестирование проводится двух последних релизов.

Также поднятие версии сервера без обновления клиента в общем случае считается допустимым. Обновления клиента без обновления сервера требуют тестирования.

Шифрование на десктопах

Настольные ОС, в отличие от мобильных, позволяют достаточно легко создавать криптоконтейнеры: шифровать диски целиком или разделы; и создавать файлы-контейнеры. При этом шифрование на уровне системы более удобное и надёжное в сравнении с шифрованием на уровне приложения. Например, такое шифрование может использовать различные дополнительные меры безопасности: TPM хранилища, смарт-карты и так далее.

Далее будут примеры для нескольких популярных настольных ОС: Linux, Mac OS и Windows.

Linux

Шифрование на Linux системах, к счастью, по большому счёту стандартизировано. Используется LUKS - Linux Unified Key Setup. Он позволяет шифровать систему целиком, отдельные диски, разделы и создавать файлы-контейнеры. Инструмент очень гибкий и позволяет настраивать множество параметров шифрования, при этом предоставлены адекватные значения по умолчанию - ими мы и будем пользоваться.

Шифрование системы целиком варьируется в зависимости от дистрибутива и чаще всего делается при установке системы, поэтому за информацией о шифровании системы целиком стоит обратиться к документации Вашего дистрибутива.

Далее рассмотрим два примера: шифрование внешнего устройства/раздела целиком и создание файла-контейнера.

Шифрование внешнего устройства /dev/sdX (требуются права root):

1. Шифруем устройство целиком или его раздел, теряя при этом его данные: cryptsetup luksFormat /dev/sdX

2. Расшифровываем устройство ключом, указанным в предыдущем шаге: cryptsetup open /dev/sdX containername

3. Если это первый раз, когда мы используем контейнер, то нам необходимо создать на нём файловую систему: mkfs.ext4 /dev/mapper/containername

4. Осталось примонтировать созданную файловую систему для пользования: mount /dev/mapper/containername /mnt/

5. Теперь все файлы, записанные по пути /mnt/, будут зашифрованы в

созданном нами устройстве-контейнере. По умолчанию права записи будут доступны только для root. Если хотите разрешить всем пользователям писать в контейнер, выполните следующую команду, когда контейнер будет примонтирован: chmod 1777 /mnt/

Далее пример создания файла-контейнера. Для создания контейнера требуется root. Для подключения контейнера можно воспользоваться утилитой udiscsctl:

1. Создаём файл нужного размера, где будет находиться наш контейнер: fallocate -l 10GB ./containername

После этого выполняем инструкцию по шифрованию внешнего устройства выше, но заменяя путь к устройству на путь к файлу.

Далее пример использования файла или устройства-контейнера без прав root с утилитой udisksctl.

1. Подключаем файл как устройство: udisksctl loop-setup –file ./containername

2. Открываем контейнер по пути, полученному из предыдущей команды: udisksctl unlock –block-device /dev/loop1

3. Монтируем файловую систему контейнера: udisksctl mount –block-device /dev/dm-2

4. Теперь контейнер подключен по пути, указанном в результате работы предыдущей команды. Он уникален для каждого контейнера.

Mac OS

Шифрование всей системы на Mac OS выполняется через FileVault. Для создания файла-контейнера можно использовать VeraCrypt по примеру для Windows ниже. За более подробной информацией по настройке FileVault обратитесь к официальной документации. Далее пример использования FileVault:

1. Выберите Apple меню > Системные настройки > Приватность & Безопасность > FileVault.

2. Нажмите Включить напротив FileVault. Возможно потребуется ввести пароль вашей учётной записи.

3. Выберите способ расшифровки: Учётная запись iCloud или Ключ восстановления.

4. Нажмите Продолжить.

Widnows

На Windows системным решением является Bitlocker. Также популярна сторонняя программа VeraCrypt.

Bitlocker использует TPM для хранения ключей. TPM - физический модуль, поэтому ключи жёстко привязаны к устройству. Также такой модуль сильно затрудняет неавторизованное извлечение таких ключей. При всём удобстве и надёжности, BitLocker позволяет шифровать только физические устройства: диски и разделы, включая системные. В некоторых случаях требуется создавать портативный файл-контейнер - для таких целей подойдёт VeraCrypt.

Главной особенностью VeraCrypt является возможность создавать файлы-контейнеры. В сравнении с Bitlocker, VeraCrypt может создать файл и использовать его как папку, вся информация в которой будет зашифрована. Это удобно, например, если не нужно шифровать всю систему или диск, а шифровать лишь данные приложения. Бонусом является портативность - файл можно переместить на другое устройство и расшифровать, зная ключ. Загрузить VeraCrypt можно на официальном сайте.

Далее приведена краткая информация по настройке BitLocker. За официальной информацией обратитесь к инструкции от Microsoft.

Шифрование системного диска:

1. Войдите в Windows на своём устройстве под учетной записью администратора, либо под учётной записью с правами администратора.

2. В поле поиска на панели задач введите ”Управление BitLocker”, а затем выберите необходимый результат из списка. Или выберите Пуск > Параметры > Конфиденциальность и безопасность > Шифрование устройства > Шифрование диска BitLocker.

3. Выберите ”Включить BitLocker” и следуйте инструкциям.

Примечание: Этот параметр будет отображаться, только если функция BitLocker доступна для вашего устройства. Она не поддерживается в выпуске Windows 11 Домашняя.

Шифрование внешнего устройства:

1. Войдите в Windows на своём устройстве под учетной записью администратора, либо под учётной записью с правами администратора.

2. Выберите Пуск > Параметры > Конфиденциальность и безопасность >

Шифрование устройства.

3. Если Шифрование устройства отключено, включите его.

Далее информация по настройке VeraCrypt. Более подробная информация по ссылке.

1. Установить и запустить приложение VeraCrypt.

2. Выбрать ”Создать том”.

3. Должно появиться окно мастера создания томов VeraCrypt. На этом этапе нам нужно выбрать место, где будет создан том (далее Контейнер) VeraCrypt. Контейнер может находиться в файле (также именуемом контейнером), в разделе или на диске. В этом примере мы выберем первый вариант и создадим контейнер VeraCrypt внутри файла. Для этого оставляем значение по умолчанию ”Создать зашифрованный файл-контейнер” и нажимаем ”Далее”.

4. Сейчас нам нужно выбрать, какой контейнер VeraCrypt мы хотим создать – обычный или скрытый. В этом примере мы выберем первый вариант и создадим обычный контейнер. Нажимаем ”Далее”.

5. На этом этапе требуется указать место создания файла-контейнера VeraCrypt. Обратите внимание: контейнер VeraCrypt ничем не отличается от любого другого обычного файла. Например, его можно перемещать или удалять как любой другой файл. Также ему потребуется имя файла, которое мы выберем на следующем этапе. Нажмите кнопку ”Выбрать файл”. Появится стандартное диалоговое окно выбора файлов Windows

6. Выберите в файловом окне желаемый путь (место, где вы хотите создать контейнер). В поле ”Имя файла” введите имя, которое вы хотите дать файлу-контейнеру. Нажмите кнопку ”Сохранить”.

7. В окне мастера создания томов нажмите ”Далее”.

8. Здесь можно выбрать для контейнера алгоритмы шифрования и хеширования. Если вы не знаете, что лучше выбрать, просто оставьте предложенные значения и нажмите ”Далее”

9. Здесь мы укажем, что хотим создать контейнер VeraCrypt размером 250 мегабайт. Разумеется, вы можете указать любой другой размер. После того как вы введёте размер в поле ввода (оно выделено красным), нажмите кнопку ”Далее”.

10. Мы подошли к одному из самых важных этапов: нам нужно выбрать

для контейнера хороший пароль. Какой пароль следует считать хорошим, написано в окне мастера. Внимательно прочитайте данную информацию. После того как вы определитесь с хорошим паролем, введите его в первое поле ввода. Затем введите тот же самый пароль в расположенное ниже второе поле ввода и нажмите кнопку "Далее".

11. Произвольно перемещайте мышь в окне мастера создания томов в течение хотя бы 30 секунд. Чем дольше вы будете перемещать мышь, тем лучше – этим вы значительно повысите криптостойкость ключей шифрования (что увеличит их надёжность). Нажмите кнопку "Разметить". После этого начнётся создание контейнера. В зависимости от размера контейнера, его создание может занять время. По окончании появится окно, нажмите "OK".

12. Итак, только что мы успешно создали контейнер VeraCrypt (файловый контейнер). Нажмите кнопку "Выход" в окне мастера создания томов VeraCrypt. Окно мастера должно исчезнуть. На следующих этапах мы смонтируем контейнер, который только что создали. Вернитесь на главное окно VeraCrypt.

13. Выберите в списке букву диска. Она станет буквой диска со смонтированным контейнером VeraCrypt.

14. Нажмите кнопку "Выбрать файл". Появится обычное окно выбора файлов.

15. В окне выбора файлов найдите и укажите файловый контейнер (который мы создали на шагах 6-12). Нажмите кнопку "Открыть".

16. В главном окне VeraCrypt нажмите кнопку "Смонтировать". Появится окно ввода пароля.

17. Укажите пароль (который мы задали на шаге 10) в поле ввода. Нажмите "OK".

18. Сейчас VeraCrypt попытается смонтировать наш контейнер. Если пароль указан неправильно (например, вы ошиблись при вводе), VeraCrypt известит вас об этом, и потребуется повторить предыдущий этап (снова ввести пароль и нажать "OK"). Если пароль правильный, контейнер будет смонтирован.

19. Мы только что успешно смонтировали контейнер как виртуальный диск. Этот виртуальный диск полностью зашифрован и ведёт себя как настоящий диск. Вы можете сохранять файлы на этом виртуальном диске - они будут шифроваться на лету в момент записи. Если вы откроете файл, хранящийся

в контейнере VeraCrypt, например, в медиа проигрывателе, этот файл будет автоматически расшифровываться в памяти непосредственно в момент считывания, то есть на лету. Правильный пароль нужно указать только один раз - при монтировании контейнера. Просматривать содержимое смонтированного контейнера можно точно так же, как содержимое любого другого диска. Например, открыть ”Компьютер” (или ”Мой компьютер”) и дважды щёлкнуть по соответствующей букве диска.

Офлайн работа

Офлайн работа как таковая напрямую не связана с фреймворком, но особенности реализации всё же существуют.

Для удачной офлайн работы в приложении достаточно выполнять следующие условия:

1. Игнорировать вызов методов download() у FMPQuery/FMPResource и FMPFile. Сделать это проще всего имея доступ к флагу, например, isOnline:

```
if (isOnline) {
  file.download()
}
```

2. И в конечном приложении создать логику по офлайн работе: проверка онлайна, показ уведомления пользователю и так далее. Для проверки доступности сервера FMP можно использовать метод FMPUtil.checkConnection(), описанный в статье Утилитарные методы фреймворка.

3. Дополнительно есть возможность проверки времени жизни токена пользователя используя метод FMPUser.getTokenExpiration():

```
val exp = fmp.user.getTokenExpiration().result
val now = System.currentTimeMillis()
if (now > exp * 1000) {
  /* Логика просроченного токена в офлайне. */
}
```

Работа при плохом соединении

В случаях, когда соединение клиента и сервера нестабильно, есть возможность использования механизма повтора запросов. Настраивается при инициализации фреймворка с указанием количества повторов и их интервала. Для установки числа повторов используется метод FMP.Builder.retryCount(Int), а для установки интервала между повторами в секундах — метод FMP.Builder.retryInterval(Int). Таким образом, если мы установим интервал в 2 секунды и число повторов равное 3, то фреймворк будет пытаться отправить максимум 3 запроса с интервалом в 2 секунды.

Если сервер вернул ошибку, то это считается успешным запросом и повторный запрос выполняться не будет. Повторы выполняются только в случаях, когда не удалось связаться с сервером.

Аутентификация сервера (Пиннинг сертификата)

Известный факт, что сервер занимается проверкой личности пользователя. Обычно это делается с использованием учётных данных пользователя - пары из имени пользователя и его пароля, либо же пары Jwt токенов. Но не только сервер может стать целью для атак. Клиенту также стоит проверять, что сервер не пытается обманом запросить у пользователя его учётные данные. В случае с сервером это делается через проверку его сертификата.

Для этого клиенту и серверу нужно заранее обменяться сертификатом. Это обычный файл, который можно положить в проект приложения. Приложению, которое использует фреймворк FMP, нужно всего лишь передать файл при инициализации. Фреймворк займётся проверками сервера.

Для этого при инициализации необходимо вызвать метод FMP.Builder.cert() с передачей данных сертификата. Пример:

```
val fmp: FMP = FMP.Builder()
  .cert(resources.openRawResource(R.raw.cert).readBytes())
  ...
  .build()
```

В случае ошибок проверки сертификата фреймворк вернёт исключение CertException. Это не всегда означает, что кто-то пытается представиться сервером. Такая ситуация может возникнуть из-за перевыпуска сертификата самим сервером. Поэтому администраторам сервера и мобильным разработчикам необходимо договариваться о подобных изменениях заранее.

Копирование и модификация объектов

Модели сами по себе изолированы от внешнего воздействия во избежание ошибок. Модификация происходит через контролируемый процесс копирования объекта. Для этого у каждой модели есть метод rebuild(), который создаёт копию параметров конструктора Builder. Например, если у нас есть модель FMPResource, то вызов метода создаст Builder с актуальными параметрами. После вызова build() будет создан новый объект FMPResource.

```
val modified = resource.rebuild().params("new").build()
```

Отладка работы фреймворка

Для отладки иногда может потребоваться посмотреть, что находится в базе данных или файле. Так как по умолчанию все файлы и базы данных надёжно зашифрованы, был создан метод для отладки.

Чтобы отключить шифрование, необходимо при инициализации FMP.Builder вызвать метод debugNoEncryption(true).

Метод отмечен как @Deprecated и будет показывать предупреждение при каждой сборке приложения. Настоятельно рекомендуем не отключать шифрование в релизных приложениях. Также нет никакой совместимости при смене режимов: нельзя продолжить работу с существующими файлами после включения или отключения шифрования. При смене режимов старые данные будут утеряны.

Также доступен метод debug(Boolean), который при передаче true включает подробное логирование в logcat всего происходящего внутри фреймворка.

Сбор статистики работы методов

Фреймворк поддерживает сбор статистики по работе собственных методов. Статистика включает в себя информацию об устройстве и пользователе, HTTP запросах и времени работы на различных этапах.

Активируется сбор статистики на сервере платформы - при использовании уровня логирования Инфо или Дебаг. Там же указывается и расписание отправки логов. Статистика выгружается вместе с логами в виде файла. Файл размещается в Локальном хранилище, в директории /analytics. Разработчику мобильного приложения ничего дополнительно настраивать не требуется.

Индексация SQLite

Для хранения кэша данных на устройстве фреймворк использует СУБД SQLite. При использовании ресурсов с большим количеством данных есть вероятность столкнуться с медленной работой мобильного приложения. Увеличить скорость работы с данными можно с помощью индексации таблиц.

Перед созданием индексов рекомендуется проанализировать данные записанные в таблице и скорость обновления этих данных. Если поле выбранное для индексации содержит в себе много значений NULL, то индексация может быть не столь эффективна. Также понизить эффективность может и частое обновление данных в таблице, так как при добавлении новых данных запускается процесс пересоздания индексов.

Для создания индекса требуется выбрать поле, которое будет индексировано. Поле для индексации необходимо выбрать то, по которому будет осуществлять выборка данных. При создании индексов следует учитывать, что первоначально приложением должны быть получены данные таблиц, и только после этого они могут быть проиндексированы.

Алгоритм выглядит так:

1. Отправить запрос на получение данных таблицы через метод FMPResource.download() или FMPQuery.download().

2. Создать индекс для требуемой таблицы:

```
resource.database.query("CREATE INDEX index_name "
  + "on table_name (row_name);")
```

Утилитарные методы фреймворка

Фреймворк содержит специальную модель FMPUtil для выполнения утилитарных задач, таких как получение версии фреймворка и проверка доступности сервера.

Узнать версию фреймворка можно через свойство version. Если где-то в Вашем приложении отображается версия приложения, то рекомендуем также отображать рядом версию фреймворка. Это поможет в решении возникших вопросов пользователя.

Метод chechConnection() возвращает true, если сервер доступен и корректно отвечает на запрос. Рекомендуем использовать этот метод для проверки соединения, так как он надёжнее ping запросов и «пустого» запроса на сервер.

Метод getServerInfo() позволяет узнать версию сервера и поддерживаемые версии API.

Часто задаваемые вопросы

1. «У меня вопрос или предложение по самому фреймворку» - В таком случае Вы можете с нами связаться через техническую поддержку по ссылке https://www.fsight.ru/support.

2. «Где получить ссылку на документацию Dokka?» - Держите! https://fmp-framework.fsight.ru/android-dokka.