Discord – это платформа для общения в режиме реального времени, разработанная для групповых обсуждений, обмена сообщениями, голосовых звонков и видеозвонков. Она предоставляет возможность создания серверов, на которых пользователи могут создавать текстовые и голосовые каналы для общения по интересам.

[Устное описать скриншот]

Боты выглядят, как обычные аккаунты, но управляются не человеком, а программой. Обычно они разрабатываются для облегчения административных задач, предоставления дополнительной функциональности и просто ради развлечения участников.

Существует множество популярных ботов. Самые известные – это Dyno, MEE6, Dank Memer, Rythm, Tatsumaki и Poll Bot.

Бесплатные боты в Discord обычно предоставляют наиболее популярные и простые функции.

[Устно рассказать про функции; упомянуть: боты музыки – вкл, выкл, плейлист; боты базовые – приветствие, уровень, бан-мут, вайп; боты-пикчеры; боты-защитники.]

Если же задача поставлена достаточно уникальная, то бесплатного бота может и не найтись. В худшем случае, ни один платный бот не выполняет то, что нужно именно вам.

В таком случае встаёт вопрос разработки своего бота. Благо, Discord предоставляет хорошо документированный API, который позволяет любому желающему создавать ботов с кастомным функционалом. Для этого требуется лишь опыт работы с HTTP-запросами.

[Устно рассказать про сайт, про группы разрабов и одиночных разрабов, про токены.]

Уже на стадии разработки бота могут возникнуть различные проблемы. Например, работа с HTTP-запросами к Discord API достаточно утомительна за счёт относительно низкоуровневого подхода. Существуют сторонние библиотеки с наборами готовых функций, но они иногда устаревают и оказываются заброшенными.

Развёртывание уже готового бота – тоже непростая задача. Чтобы бот работал круглосуточно, нужно либо покупать сервер (довольно дорого), либо арендовать хост (относительно дорого), либо запускать бота на своём компьютере и держать его включённым круглосуточно (неудобно).

Боты для Discord, как и боты для других мессенджеров и соцсетей, чаще всего пишутся на Python. Однако, в теории, разработка бота возможна на любом языке, который поддерживает работу с HTTP-запросами.

Необходимость создания бота для собственного сервера и опыт работы с Java/Kotlin привёл меня к мысли о создании **бота, который будет компилироваться и работать на телефоне**. Телефон – идеальный бесплатный хост, который работает круглосуточно и имеет доступ к мобильному интернету.

Поскольку Java входит в число самых популярных языков программирования, неудивительно, что для неё существует сторонняя библиотека по работе с Discord API – **Javacord**.

Javacord требует для работы JDK 8 (1.8), следовательно, его уровень байткода совместим с абсолютным большинством версий Android.

Kotlin, предоставляющий наиболее современный пользовательский опыт работы с Android API, также совместим с JDK 8 (1.8). Стоит отметить, что все библиотеки для Java совместимы и с Kotlin.

[Устно рассказать про проблему версий Java на Android.]

Первоначально я решил разработать бота на Kotlin, который будет компилироваться под Windows. Доступ к терминалу облегчал тестирование и отладку приложения, что удобно на ранней стадии разработки.

Бот начал реализовываться в виде **многомодульного проекта Gradle** (Kotlin DSL). Первыми появились модули **Windows** и **Common**. Согласно задумке, первый модуль должен был содержать сугубо GUI и входные данные (токен бота и идентификатор владельца), а второй — весь функционал, который позже сможет работать и на Android.

[Устно рассказать про прецедент смены аккаунта.]

У ботов нет общепринятой архитектуры – некоторые разработчики пишут ботов даже в процедурном стиле.

Имея опыт работы с Java, я принял решение использовать Layered Architecture (Controller, Service, DAO) с применением паттернов Singleton Factory и Bean.

[Устно рассказать про паттерны: в общем и про эти конкретно.]

Для работы компонентов было задействовано три библиотеки, не считая Javacord: **Zip4j**, **Gson** и **Apache HTTP Client**.

[Устно рассказать про библиотеки; упомянуть «переезд» апаче.]

Поскольку бот не взаимодействует с базой данных, то было решено, что слой DAO будет предназначен для работы с файловой системой. Всего было добавлено три компонента – функции для работы с bin-файлами данных, функции для работы с json-файлами конфигурации и функции для работы с zip-архивами.

[Устно рассказать, что именно в джсоне, какие именно функции зип, что лежит в бин, какое там шифрование.]

– Слой Service, согласно изначальному плану, должен был содержать шесть компонентов — функции для проверки доступа к команде, функции для авторизации бота, функции владельца, функции админа, функции бота и функции рядового участника.

Слой Controller представляет собой Listener’ы шины событий библиотеки Javacord.

[Устно рассказать, что делают листенеры.]

Во время первой фазы разработки проводилось написание основного кода для функций бота, а также его отладка, оптимизация и исправление ошибок. Так, в определённый момент был совершён переход от обычных команд Discord к интеракционным, которые являются отдельным ивентом и позволяют не запускать проверки на каждом сообщении в чате.

Для разработки графического интерфейса был использован **Compose**

Desktop — фреймворк для разработки декларативных GUI на Kotlin, который позволяет применить опыт и стиль разработки интерфейсов Android на других платформах.

В результате первой фазы разработки был завершён весь основной функционал бота, а именно:

– Добавление и удаление дней рождения, менеджеров и секретных каналов бота.

– Установка языка бота и шанса цитирования/простановки реакции.

– Массовое удаление сообщений в чате.

– Рандомное цитирование участников и простановка реакций к сообщениям.

– Поздравление участников с днём рождения.

– Импорт и экспорт данных бота в формате zip-архива.

– Дистанционное выключение бота.

– Рандомный ответ на вопрос участника, выбор слова из списка, выбор числа из диапазона, вывод информации о сервере.

– Работа с нейросетью «Порфирьевич» для продления текста.

Во второй фазе разработки началась разработка третьего модуля проекта – **Android**. В ходе разработки пришлось вносить коррективы в модуль Common, поскольку работа с файловой системой в Android ограничена и зависит от **App Context**.

Методом проб и ошибок был выбран тип приложения — **форграунд-сервис** (то есть сервис, который работает в фоновом режиме, но при этом явно, с уведомлением). Именно этот способ позволяет создать приложение, которое, при отсутствии режима экономии, будет работать в Android бесконечно.

Графический интерфейс был разработан при помощи **Jetpack Compose**, то есть той же технологией, что и интерфейс для Windows.

В результате второй фазы разработки был создан третий модуль, позволяющий запустить бота на телефоне. Программа показала высокую стабильность и длительную (около тридцати дней) работу в круглосуточном режиме.

[Как минимум 18 дек 2023 – 24 янв 2024.]

В ходе третьей фазы проводилась «полировка» уже существующего функционала, повышение стабильности и переработка плохих архитектурных решений.

[Экранирование кавычек, грамотная асинхронная работа, чтение и запись данных на каждом событии.]

В частности, именно в этот момент был введён седьмой компонент сервисов – функции для взаимодействия с DAO. Он позволил систематизировать обращения к DAO и собрать их все в одном месте, чтобы безошибочно спроектировать структуру папок с данными.

[Устно рассказать, почему импорт и экспорт стал важен.]

Помимо новых сервисов, была разработана новая система локализации на основе форматированных строк. Это позволяет делать качественные пере-

воды на любой язык, учитывая то, что в разных языках порядок слов и построение предложений могут отличаться.

[Устно рассказать, какая система была до этого.]

В презентации уже были рассмотрены слои архитектуры, но не были рассмотрены структуры данных. Всего бот использует четыре структуры данных: это **ServerData**, ApiResponse и BotData.

– ServerData – самый важный класс данных, содержащий всю информацию о сервере.

– ApiResponse – класс, хранящий ответ с API «Порфирьевича».

– BotData – синглтон, хранящий данные из GUI Windows/Android.

Все алгоритмы сервисов изначально похожи, отличаясь лишь на глубоких уровнях вложенности.

Пример кода:

```
override fun clearMessages(event: InteractionCreateEvent) {
  val sc = event.slashCommandInteraction.get()

  if (sc.fullCommandName.contains("clear_messages")) {
   sc.respondLater().thenAccept {
    val server = sc.server.get()
    val data = dataService.loadServerData(server)

    val embed = if (!accessService.fromAdminAtLeast(sc, data)) {
     EmbedBuilder().access(sc, data, I18n.of("no_access", data))
    } else {
     dataService.wipeServerMessages(server)
     EmbedBuilder().success(sc, data, I18n.of("cleared_messages", data))
    }
    sc.createFollowupMessageBuilder().addEmbed(embed).send().get()
   }.get()
  }
 }
```