

# **Отчет о проделанной работе в ходе изучения основ разработки Telegram ботов**

## **Оглавление**

Настройка и развёртывание	3
Авторизация пользователей	5
Пользовательские сессии	7
Цепочки сообщений	9
Отправка самостоятельного сообщения пользователю	10
Клавиатуры	12

## Настройка и развёртывание

Для работы с API Telegram есть готовый гем <https://github.com/telegram-bot-rb/telegram-bot> который позволяет создать архитектуру приложения на основе **Rails**.

Для этого я создал:

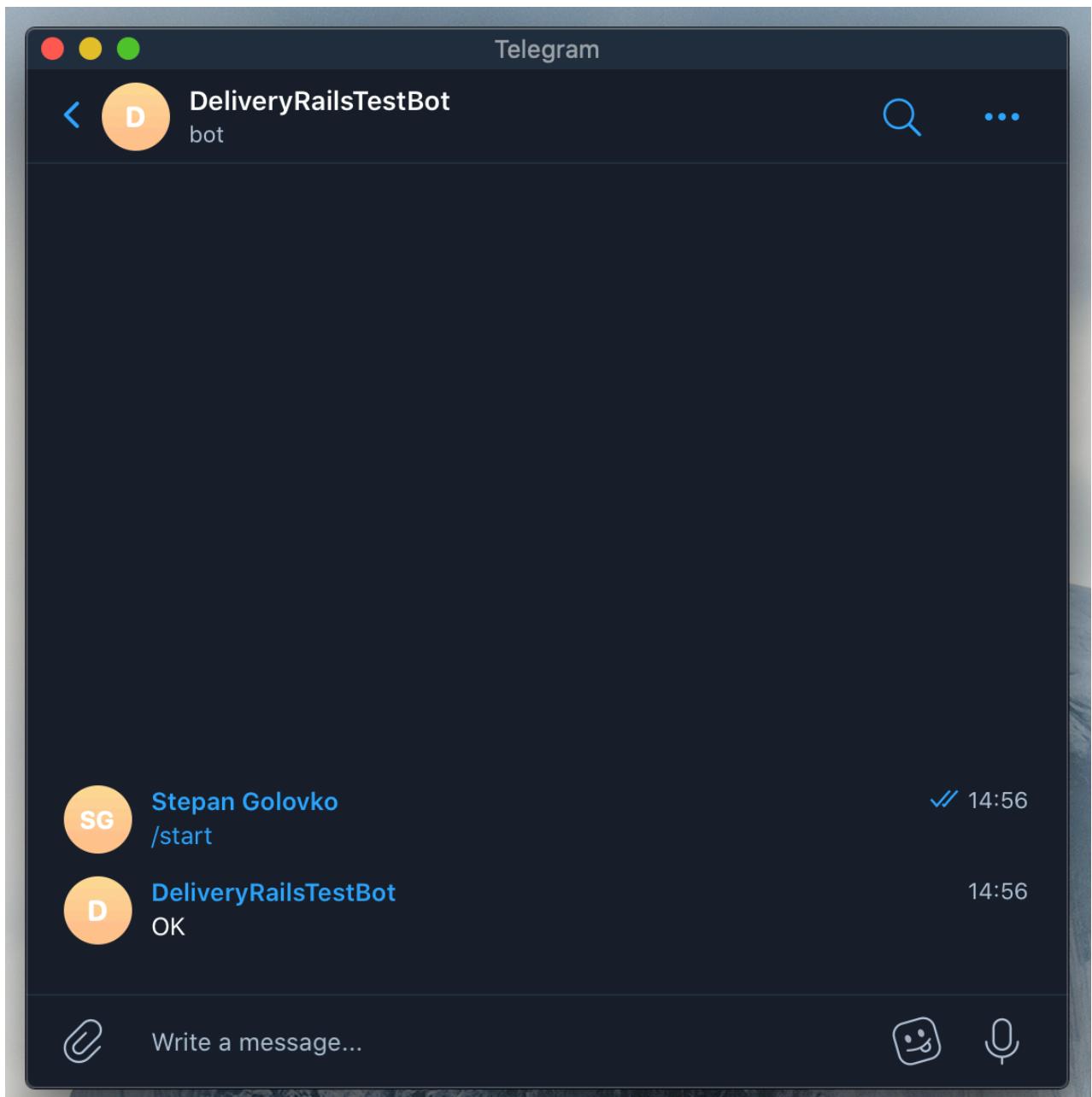
- Репозиторий: [https://github.com/GolovkoStepan/rails\\_telegram](https://github.com/GolovkoStepan/rails_telegram)
- Выполнил деплой на Heroku: <https://rails-telegram.herokuapp.com>
- Зарегистрировал бота с помощью другого бота **@BotFather**
- Имя бота в Telegram: **@DeliveryRailsBot**

При этом могу выделить следующие особенности:

- При регистрации бота, **@BotFather** дает токен для работы с API, который далее мы добавляем в переменные окружения сервера и подгружаем в файле конфигурации бота (**secrets.yml**). Также, следует указать в конфигурации имя пользователя бота (**@DeliveryRailsBot**)
- Для корректной работы гема требуется указать хост для production окружения (**routes.default\_url\_options**).

После настройки, для тестирования был добавлен **вебхук** для обработки пользовательских сообщений. Для этого мы должны определить контроллер для обработки запросов от API Telegram, а также определить маршруты.

Для того, чтобы сервер начал принимать входящие запросы от API Telegram, необходимо выполнить: **rails telegram:bot:set\_webhook**. После этого бот будет отвечать на команду **/start** сообщением «OK»:



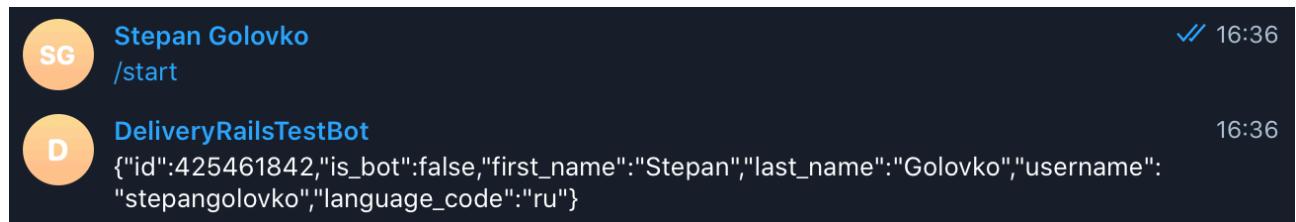
Для **development** окружения есть способ запуска бота без развернутого **rails** приложения. Для этого нужно добавить в **secrets.yml** конфигурацию бота для **development** (по аналогии с **production**) и запустить **rails telegram:bot:poller**. Сервер при этом запускать не требуется.

## Авторизация пользователей

Для того, чтобы получить информацию о пользователе, который отправляет сообщение боту, можно воспользоваться методом `from`. Ниже приведенный фрагмент кода будет отправлять пользователю данные о нем в ответ на команду `/start`:

```
class TelegramWebhooksController < Telegram::Bot::UpdatesController
  def start!(*)
    respond_with type: :message, text: from
  end
end
```

В итоге обратно будет отправлен хеш вида:



Хеш состоит из:

- id пользователя в Telegram
- is\_bot (является ли отправитель ботом)
- first\_name (Имя)
- last\_name (Фамилия)
- username (имя пользователя)
- language\_code (язык)

Этих данных достаточно для авторизации пользователя. Для хранения данных о пользователях создадим модель `TelegramUser` с полями, представленными этим хешем. При запуске команды `/start` пользователь будет сохраняться в БД, если его нет. Ниже будет представлен код, реализующий это:

```

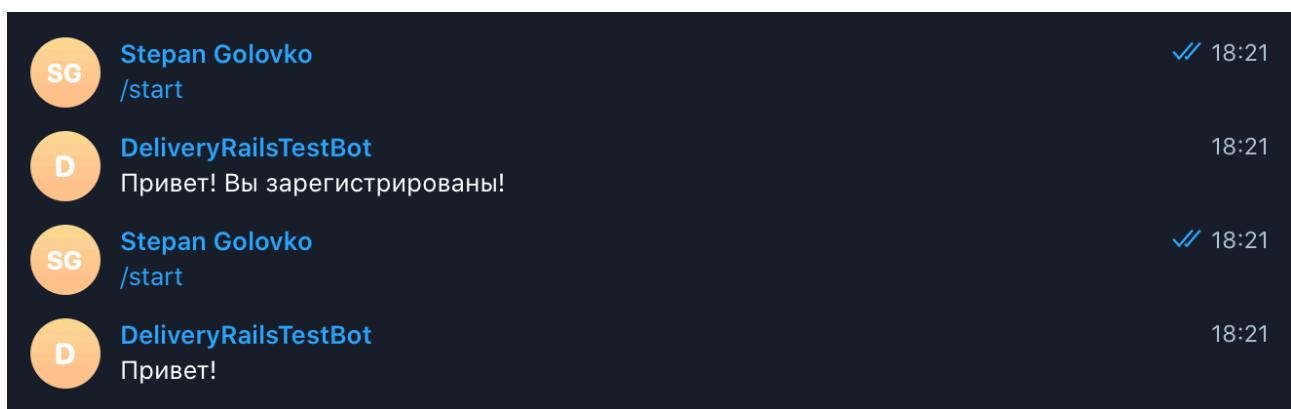
class TelegramWebhooksController < Telegram::Bot::UpdatesController
  def start!(*)
    return if from['is_bot']

    if TelegramUser.find_by(external_id: from['id'])
      respond_with type: :message, text: 'Привет!'
    else
      create_user!
      respond_with type: :message, text: 'Привет! Вы зарегистрированы!'
    end
  end

  private
  def create_user!
    TelegramUser.create!(
      external_id: from['id'],
      first_name: from['first_name'],
      last_name: from['last_name'],
      username: from['username'],
      language_code: from['language_code']
    )
  end
end

```

На следующем изображении показан процесс с от лица пользователя:



Также, можно создавать список пользователей заранее и работать только с теми, которые есть в БД (возможно, подходит лучше в контексте задачи).

## Пользовательские сессии

Для того, чтобы сохранять некоторые данные в рамках одной сессии можно воспользоваться методом session, который возвращает хеш. Таким образом, вызов `session[:some_key] = 'some value'` закеширует строку `'some value'`. Для реализации этой возможности необходимо настроить кеширование с помощью redis для текущего окружения и создать метод, который будет формировать уникальный ключ для текущего пользователя.

Опишем шаги реализации:

- Добавить gem redis в проект
- Настроить кеширование
- Переопределить метод `session_key` в контроллере
- Создать методы-обработчики команд боту

Ниже представлен код, который будет сохранять сообщение пользователя и при запросе его отправлять:

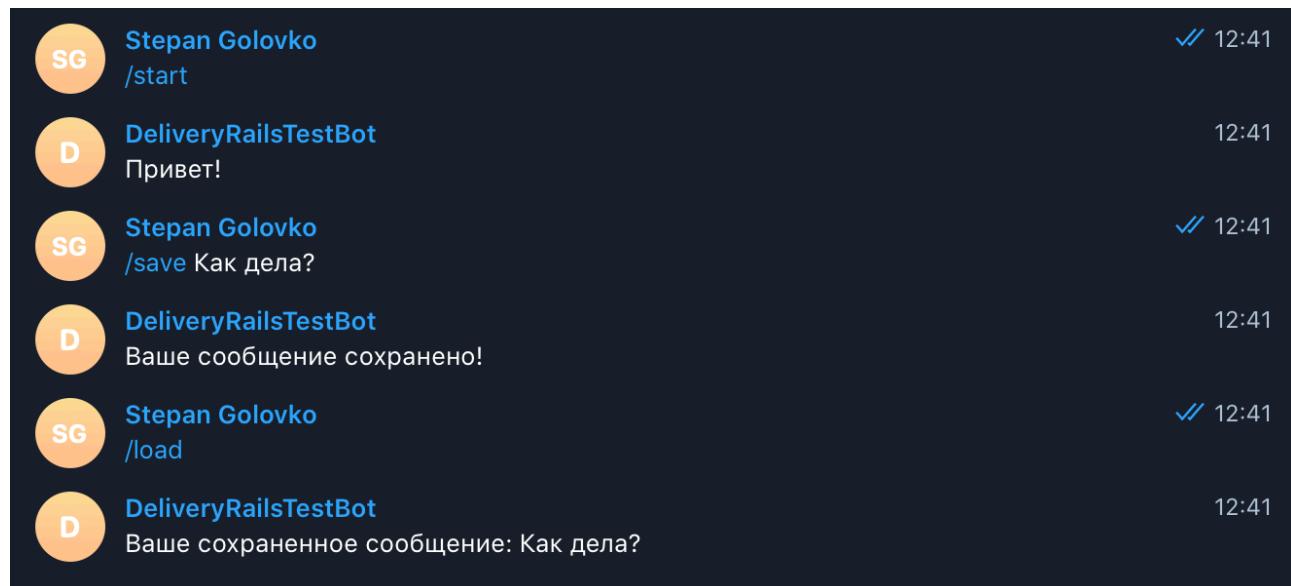
```
def save!(*words)
  session[:msg] = words.join(' ')
  respond_with type: :message, text: 'Ваше сообщение сохранено!'
end

def Load!(*)
  respond_with type: :message, text: "Ваше сохраненное сообщение: #{session[:msg]}"
end

private

def session_key
  "#{bot.username}:#{chat['id']}:#{{from['id']}}" if chat && from
end
```

На следующем изображении показан процесс от лица пользователя:



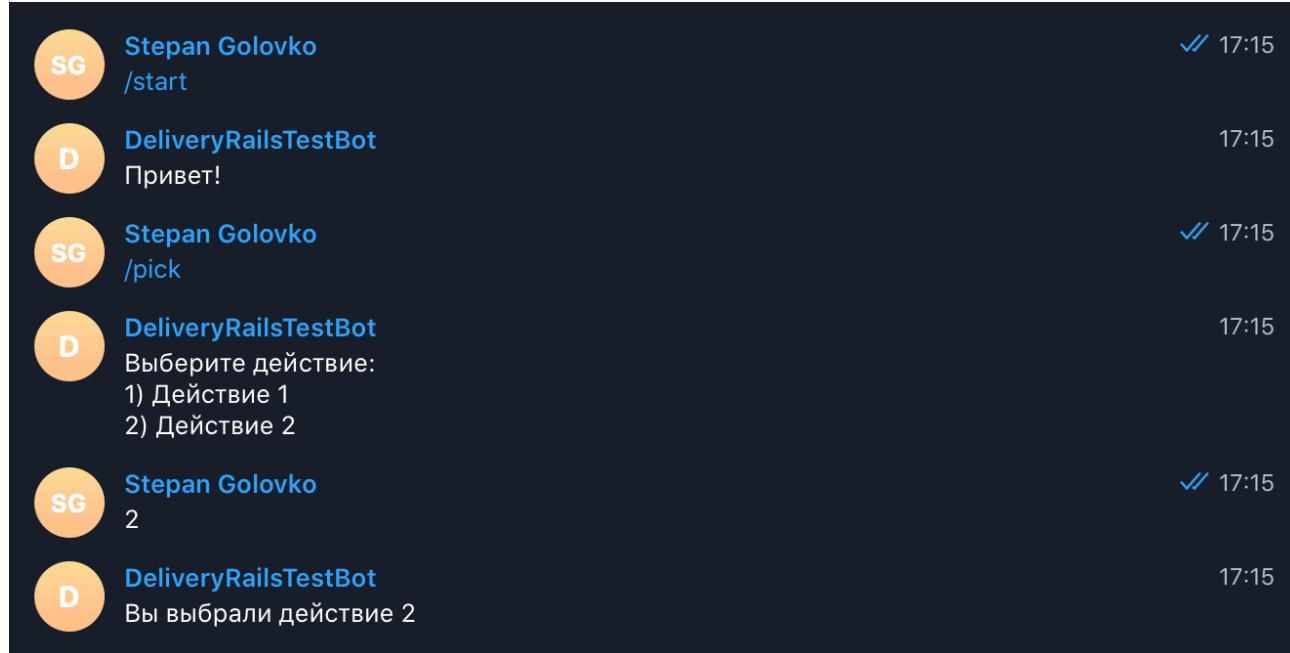
## Цепочки сообщений

Чтобы организовать работу с ботом в виде переписки, существует способ указывать следующий обработчик для ответа пользователя. Для этого есть метод `save_context`. После получения ответного сообщения пользователя управление перейдет в метод, который передаётся в качестве аргумента методу `save_context` в виде символа. Ниже будет приведен код, который использует этот механизм:

```
def pick!(*)
  save_context :pick_action
  respond_with type :message, text: "Выберите действие:\n1) Действие 1\n2) Действие 2"
end

def pick_action(action = nil, *)
  respond_with type :message, text: "Вы выбрали действие #{action}"
end
```

На следующем изображении показан процесс от лица пользователя:



## Отправка самостоятельного сообщения пользователю

В некоторых случаях будет полезно отправлять самостоятельные сообщения (уведомления) асинхронно, например, в **sidekiq** задачах. Для этого есть способ вызвать некоторое действие контроллера отдельно от запущенного сервера. Для решения этой задачи определим следующий код в контроллере:

```
def self.send_message(to:, text:)
  new(bot, { from: { 'id' => to }, chat: { 'id' => to } }).process(*:send_message, text)
end

def send_message(text)
  respond_with type: :message, text: text
end
```

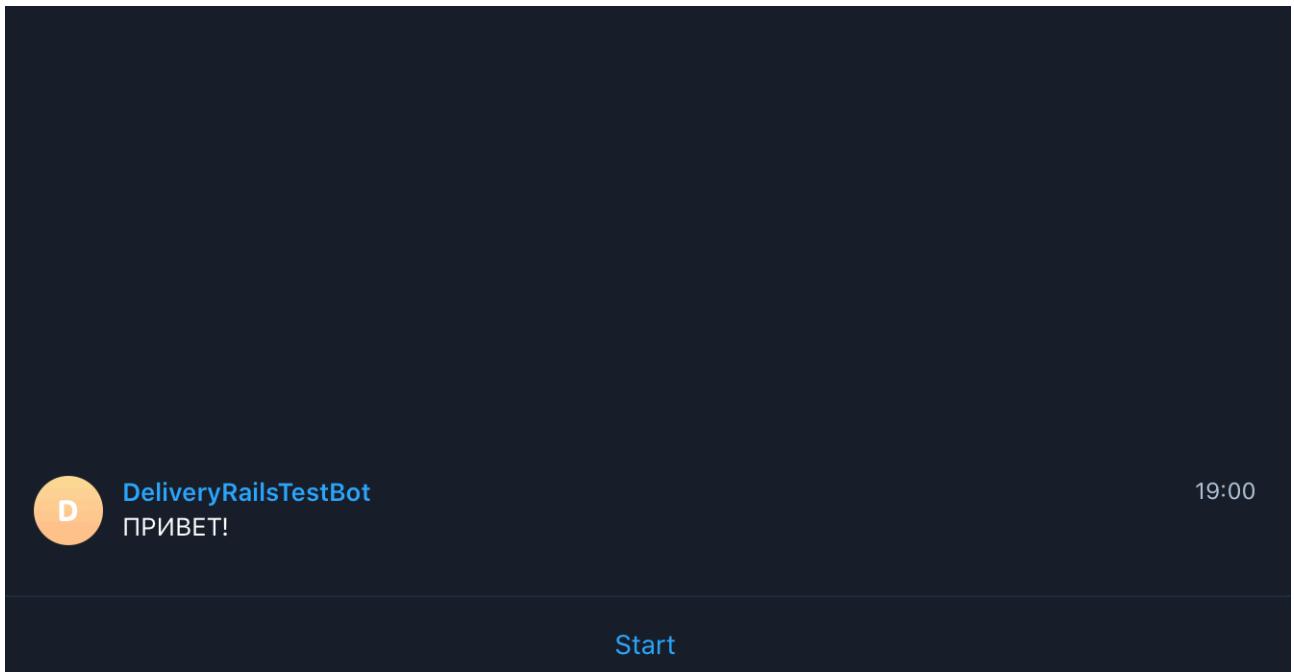
Статический метод **send\_message** принимает на вход 2 аргумента: **to** - идентификатор пользователя и **text** - текст сообщения. Далее будет создан объект класса **TelegramWebhooksController**, у которого будет вызван метод **process**. Этот метод вызывает действие контроллера, имя которого передается первым аргументом, вторым аргументом передаются параметры (в данном случае это один аргумент типа **string**). После этого будет вызван метод объекта **TelegramWebhooksController** **send\_message**, который и отправит пользователю сообщение.

Основной момент, который стоит учитывать: для указания идентификаторов **from** и **chat** должен использоваться так называемый **«rocket syntax»** (`'id' => to`) или в качестве альтернативы можно использовать **with\_independent\_access**.

Воспользоваться этим можно в консоли следующим образом:

```
2.7.2 :167 >
2.7.2 :168 >
2.7.2 :169 >
2.7.2 :170 > c = TelegramWebhooksController.send_message(to: 425461842, text: 'ПРИВЕТ!')
Processing by TelegramWebhooksController#send_message
  Update: null
Responded with message
Completed in 510ms ( ActiveRecord: 0.0ms)
=> {"ok"=>true, "result"=>{"message_id"=>245, "from"=>{"id"=>1488082198, "is_bot"=>true, "fir
2.7.2 :171 >
```

На следующем изображении показан процесс с от лица пользователя:



## Клавиатуры

Существует способ задать разметку клавиатуры для ввода команд.

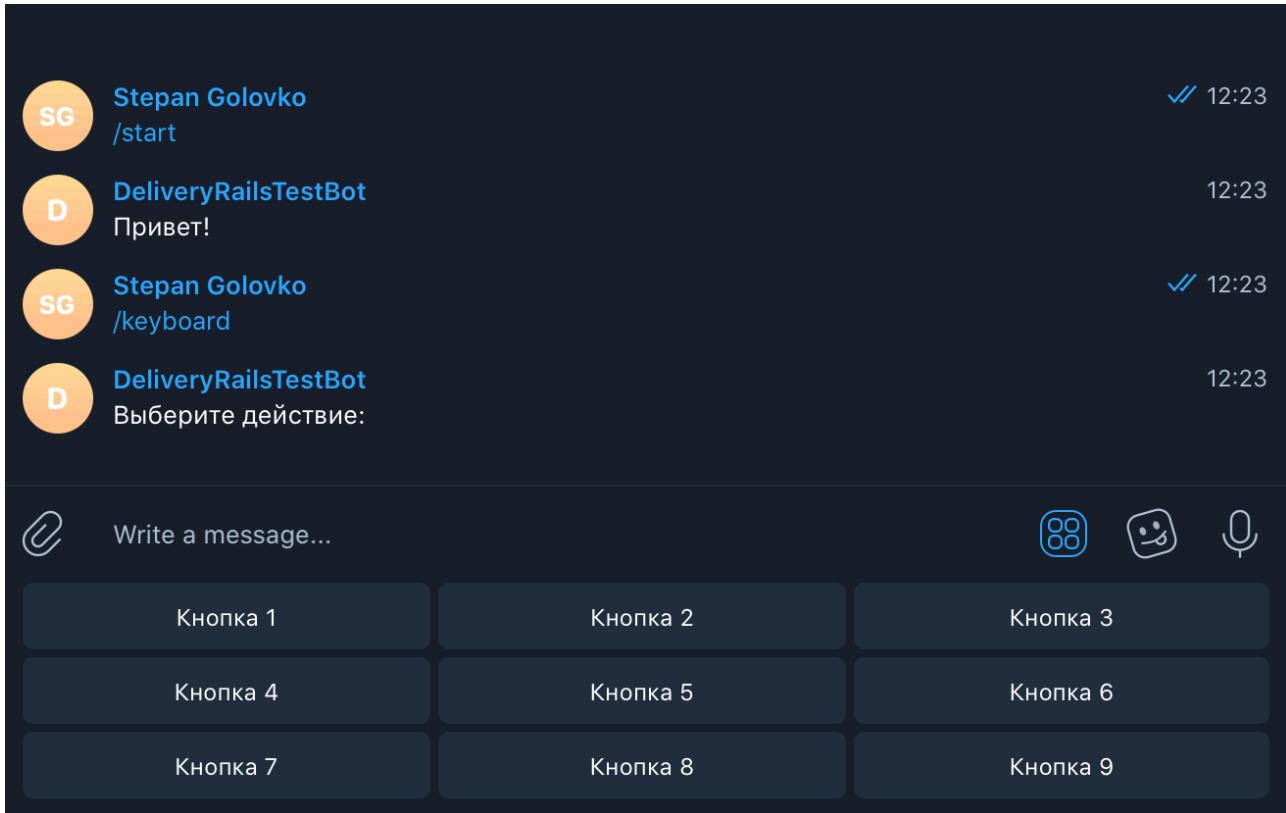
Для этого, создадим следующий код:

```
def keyboard!(*words)
  buttons = [
    ['Кнопка 1', 'Кнопка 2', 'Кнопка 3'],
    ['Кнопка 4', 'Кнопка 5', 'Кнопка 6'],
    ['Кнопка 7', 'Кнопка 8', 'Кнопка 9']
  ]

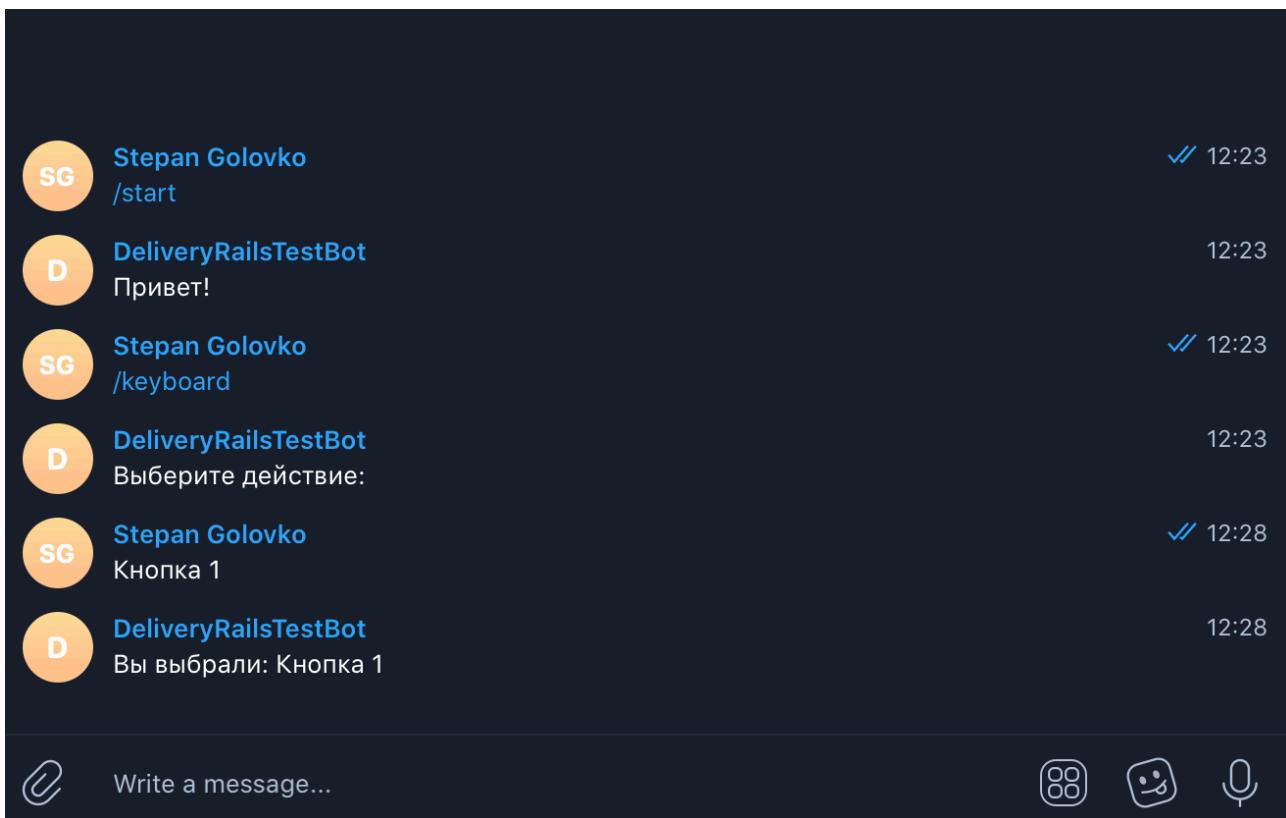
  if words.any?
    respond_with type :message, text: "Вы выбрали: #{words.join( separator ' ')}"
  else
    save_context :keyboard!
    respond_with type :message, * [* text: 'Выберите действие:', reply_markup: {
      keyboard: buttons,
      resize_keyboard: true,
      one_time_keyboard: true,
      selective: true
    }]
  end
end
```

В методе **keyboard!** определены кнопки, которые задаются в виде двухмерного массива. Вызов метода **save\_context :keyboard!** замыкает обработку ответа пользователя на этот же метод (но можно сделать разные обработчики). При нажатии на кнопку пользователь, по сути, отправляет сообщение боту, которое будет обработано в этом же методе. При получении ответа, он будет просто выведен обратно в сообщении. Таким образом, с помощью этого можно реализовать меню выбора действия в более удобном для пользователя формате.

На следующем изображении показан процесс с от лица пользователя:



После нажатия на кнопку экран будет выглядеть следующим образом:

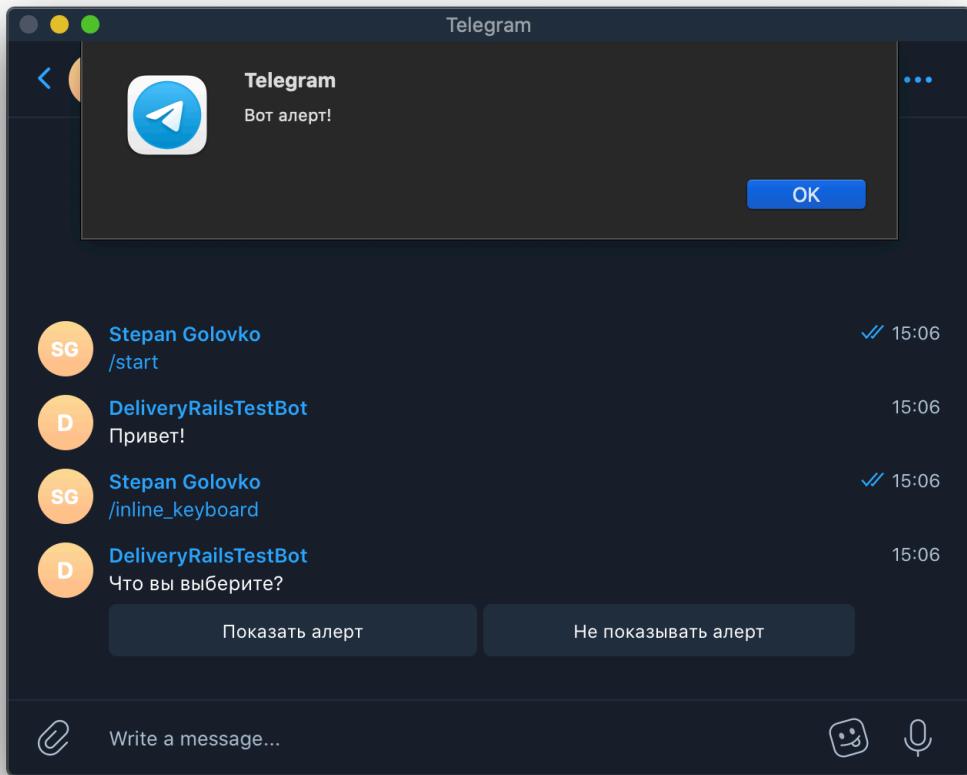


Помимо подхода, представленного выше есть еще один подход к созданию клавиатуры, но уже в качестве отдельного сообщения бота. Такая клавиатура сохраняется в переписке, и ей можно будет воспользоваться позже и несколько раз. Ниже будет представлен код, который реализует это:

```
def inline_keyboard!(*)
  respond_with type :message, * [text: 'Что вы выберите?', reply_markup: {
    inline_keyboard: [
      [
        {text: 'Показать алерт', callback_data: 'alert'},
        {text: 'Не показывать алерт', callback_data: 'no_alert'}
      ]
    ]
  }]
end

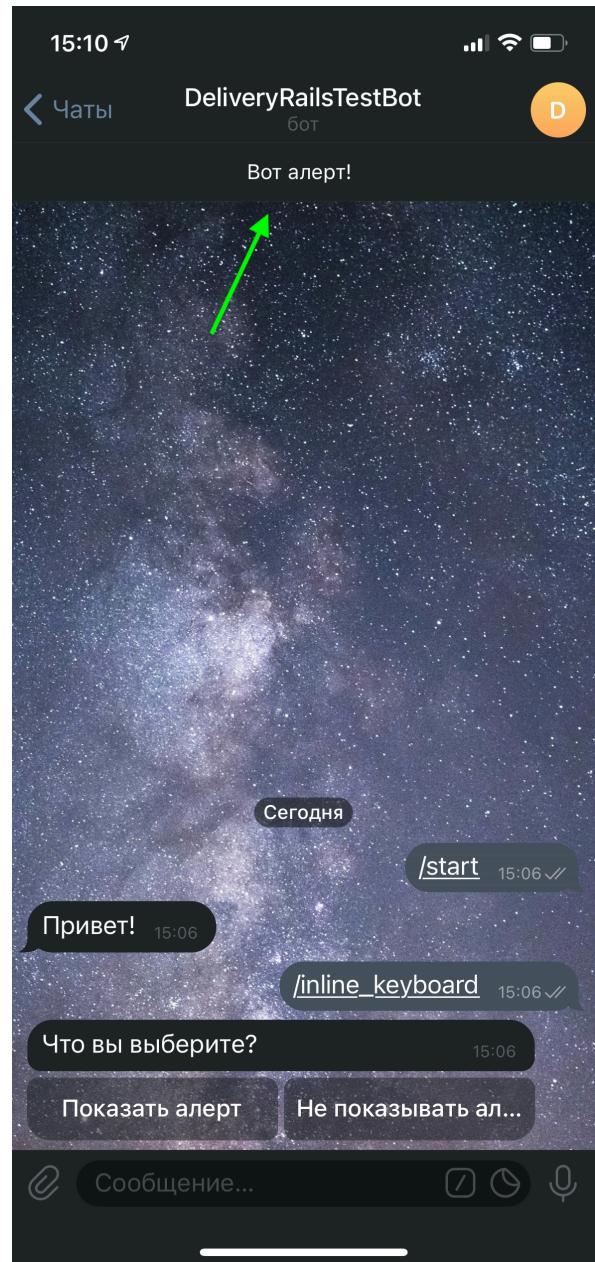
def callback_query(data)
  if data == 'alert'
    answer_callback_query 'Вот алерт!', show_alert: true
  else
    respond_with type :message, text: 'Не будет алерта...'
  end
end
```

В данном примере используется другой подход обработки пользовательского ответа. В хеш `reply_markup` включен хеш `inline_keyboard`, который содержит в себе описание кнопок. При этом, `text` - это заголовок кнопки, а `callback_data` - это некоторый идентификатор действия. После того, как пользователь нажмет на одну из кнопок, управление перейдет в метод `callback_query`. Далее, в зависимости от передаваемого аргумента `data` будет выполнен тот или иной обработчик. Метод `answer_callback_query`, прежде не указанный в этой документации, показывает сообщение в верхней части окна чата, а при передачи `show_alert: true` он покажет диалоговое окно.

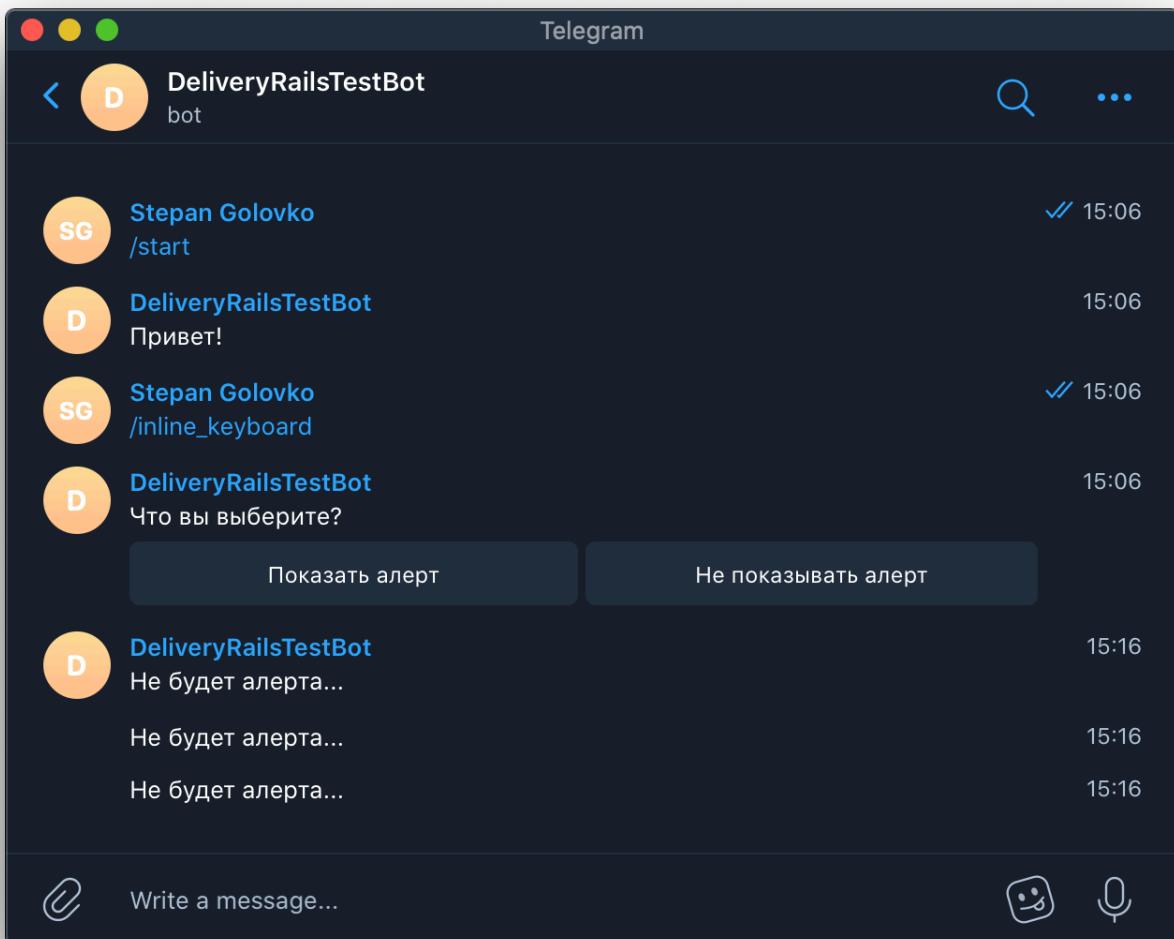


Таким будет сообщение, с установленным **show\_alert: true**

Если мы уберем **show\_alert: true**, то результат будет следующим:



При нажатии на кнопку «Не показывать алерт» пользователь в ответ получает обычное сообщение:



Весь код, который был написан в рамках создания этой документации доступен по ссылке: [https://github.com/GolovkoStepan/rails\\_telegram/tree/sandbox](https://github.com/GolovkoStepan/rails_telegram/tree/sandbox)