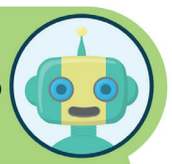
В книге рассказывается о работе диалоговых интерфейсов как способе взаимодействия машины с людьми на естественном языке.

Рассматриваемые темы:

- конструирование потока диалога пользователя в чат-боте;
- разработка чат-ботов для выполнения таких задач, как, например, заказ еды;
- обзор новых инструментов и служб в экосистеме чат-ботов;
- интеграция сторонних интерфейсов данных;
- развертывание чат-ботов на платформах мессенджеров типа Facebook;
- разработка чат-ботов с помощью платформ Microsoft Bot Framework, DialogFlow и Twilio;
- использование чат-ботов для отправки и отслеживания твитов, а также ретвитов сообщений на платформе Twitter;
- реализация в чат-ботах технологий Google Assistant и Amazon Alexa.

К концу книги вы самостоятельно разработаете группу собственных чат-ботов и голосовых пользовательских интерфейсов для всех ведущих платформ.

Издание будет полезно начинающим разработчикам в области искусственного интеллекта.



Интернет-магазин:
www.dmkpress.com
Книга – почтой:
e-mail: orders@alians-kniga.ru
Оптовая продажа:
«Альянс-книга»
Тел./факс: (499) 782-3889
e-mail: books@alians-kniga.ru

[PACKT] PUBLISHING

ДМК ПРЕСС ИЗДАТЕЛЬСТВО
www.дмк.рф

ISBN 978-5-97060-542-4

9 785970 605424 >

Разработка чат-ботов и разговорных интерфейсов

Разработка чат-ботов и разговорных интерфейсов

Срини Джанарсанам

ДМК ПРЕСС ИЗДАТЕЛЬСТВО

Срини Джанарсанам

Разработка чат-ботов и разговорных интерфейсов

Чат-боты и голосовые пользовательские интерфейсы на платформах Chatfuel, Dialogflow, Microsoft Bot Framework, Twilio и Alexa Skills

Srini Janarthanam

Hands-On Chatbots and Conversational UI Development

Build chatbots and voice user interfaces with Chatfuel, Dialogflow, Microsoft Bot Framework, Twilio, and Alexa Skills

Packt

Срини Джанарсанам

Разработка чат-ботов и разговорных интерфейсов

Чат-боты и голосовые пользовательские интерфейсы на платформах Chatfuel, Dialogflow, Microsoft Bot Framework, Twilio и Alexa Skills

ДМК Пресс
издательство
Москва, 2019

УДК 004.896
ББК 32.813
Д40

Срини Джанарсанам

Д40 Практическое руководство по разработке чат-интерфейсов: пер. с англ.: М. Райтман. – М.: ДМК Пресс, 2019. – 340 с.

ISBN 978-5-97060-542-4

В книге рассказывается о работе диалоговых интерфейсов как способе взаимодействия машины с людьми на естественном языке. Вы научитесь разрабатывать чат-боты для выполнения различных прикладных задач с помощью платформ Microsoft Bot Framework, DialogFlow и Twilio, а также изучите механизмы развертывания чат-ботов на платформах мессенджеров типа Facebook. В заключении освоите отправку и отслеживание сообщений на платформе Twitter, а также поэкспериментируете с технологиями Google Assistant и Amazon Alexa.

Издание будет полезно начинающим разработчикам в области искусственного интеллекта

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

Материал, изложенный в данной книге, многократно проверен. Но, поскольку вероятность технических ошибок все равно существует, издательство не может гарантировать абсолютную точность и правильность приводимых сведений. В связи с этим издательство не несет ответственности за возможные ошибки, связанные с использованием книги.

ISBN 978-1-78829--466-9 (анг.)
ISBN 978-5-97060-542-4 (рус.)

Copyright © 2018 by Packt Publishing
© Оформление, издание, перевод, ДМК Пресс, 2019

Содержание

Об авторе **12**

О рецензенте **13**

Предисловие **13**

О чем эта книга 14

Что вам потребуется при чтении этой книги 15

Для кого эта книга 16

Условные обозначения 16

Загрузка кода примеров 17

Глава 1. Введение **18**

Диалоговые пользовательские интерфейсы 19

Краткая история чат-ботов 20

Последние разработки 21

Архитектура диалогового пользовательского интерфейса 23

Классификация 25

Приложения 25

Инструменты разработчика 28

Инструменты прототипирования 28

Каналы 28

Средства разработки чат-ботов 29

Аналитика 29

Распознавание естественного языка 30
Службы каталогов 30
Монетизация 31
Выгоды 31
Чат-боты никуда не денутся 32
Приступим! 34

Глава 2. Путеводитель по городу 35

Начало работы 36
Создание вашего первого бота 37
Базовые элементы 40
Блоки 40
Карты 41
Кнопки 41
Плагины 41
Атрибуты 41
Блоки по умолчанию 41
Настройка бота 42
Дополнительные карты 45
Графика 45
Аудио 46
Видео 47
Быстрые ответы 47
Галерея 48
Списки 48
Навигация 49
Кнопки 50
Переход к блоку 51
Учет контекста 52
Пользовательский ввод 53
Настраиваемые пользовательские атрибуты 55
Контекстные переменные по умолчанию 56
Распознавание естественного языка 57
Блок по умолчанию 57

Обработка на стороне сервера 60
Широковещание 66
Шаблоны 67
Аналитика 68
Заключение 68
Ссылки 69

Глава 3. Поговорим о погоде 70

Задачи чат-бота 71
Конструирование диалога 71
Задачи на стороне сервера 72
Получение данных о погоде 72
Построение интерфейса на стороне сервера 76
Реализация чат-бота 81
Создание веб-службы чат-бота 88
Публикация в Facebook Messenger 93
Создание страницы Facebook 94
Создание имени пользователя 95
Создание приложения Facebook 96
Создание интерфейса для веб-приложения Facebook .. 100
Связывание приложения Facebook и приложения интерфейса 104
Тестирование чат-бота 105
Добавление других тестировщиков 106
Публикация для общего доступа 106
Дополнительные форматы сообщений Facebook 108
Типы содержимого 108
Шаблоны 109
Заключение 115
Ссылки 115

Глава 4. Создание бота-персонажа 116

Знакомство с Dialogflow 116
Настройка Dialogflow 117
Создание базового агента 118
Развертывание чат-бота 121
Контекст 127
Сущности 131
Развернутые ответы 141
Импорт и экспорт агентов 143
Осуществления 149
Заключение 154
Ссылки 156

Глава 5. Общественный транспорт 157

Обзор Transport API 157
Создание учетной записи разработчика 158
Изучение набора данных 159
Структура диалога 163
Ближайшая станция 163
Следующий поезд 164
Время прибытия 164
Создание простого SMS-бота 165
Начало работы 165
Настройка панели управления 166
Отправка простых сообщений 167
Уведомления о поездах 170
Планирование задач 173
Создание двухстороннего чат-бота 175
Обработка произношения пользователей 181
Заключение 193
Ссылки 193

Глава 6. Поиск ресторана 194

Платформа MS Bot 195
Эмулятор канала 195
Создание бота 195
Развертывание бота 198
Другие типы сообщений 200
Поток диалога 209
Состояние диалога 221
Начало работы с Zomato 222
Получение данных из Zomato 223
Служба для поиска ресторанов 225
Подключение к Skype 233
Заключение 238
Ссылки 238

Глава 7. Новостной бот 239

Начало работы с приложением Twitter 240
Создание Twitter-бота 242
Обзор SDK Twitter 246
Обновление статуса 246
Ретвиты сообщений 247
Поиск твитов 247
Обзор новостной службы 248
Настройка NewsAPI 248
Создание новостного Twitter-бота 252
Персонализация новостного бота 255
Создание базы данных интересов пользователей 259
Доступ к базе интересов пользователей 264
Информирование бота об интересах пользователей 267
Заключение 272
Ссылки 273

Глава 8. Мой телегид 274

Amazon Alexa 275
Разработка бота цитат 276
Настройка навыков 277
Модель взаимодействия 278
Менеджер диалогов – цитаты 285
Тестирование на Echo 291
Телегид 291
Построение модели взаимодействия 292
Настройка сервера для навыка телегида 296
Тестирование навыка телегида 303
Встроенные цели и слоты 305
Заключение 307
Ссылки 307

Глава 9. Мой верный слуга 308

Служба Todoist 309
Получение ключа 309
Добавление задачи 311
Получение всех задач на день 312
Создание действия 313
Создание агента Dialogflow 313
Создание веб-приложения 318
Тестирование агента на Dialogflow 322
Интеграция в Ассистента Google 325
Проверка на устройстве Google Home 330
Заключение 331
Выводы 332
Ссылки 332

Приложение. Дополнительные ресурсы 333

Статьи 333

Список конференций 333

Список журналов 334

Группы в социальных сетях 334

Предметный указатель 335

Об авторе

Срини Джанарсанам – эксперт в области диалоговых систем, более 15 лет работающий в этой сфере. Он руководил и работал над несколькими проектами, создавая диалоговые системы для различных областей, включая туризм, здравоохранение и образование. Он получил докторскую степень по философии (PhD) в Эдинбургском университете за работу в области искусственного интеллекта и обработки естественного языка.

В настоящее время он является соучредителем и директором Chatomate, британского бренда, специализирующегося на чатах, искусственном интеллекте и решениях для автоматизации бизнеса. Ранее он работал научным сотрудником в университете Хериот-Ватт и университете Эдинбурга. Он опубликовал более 50 статей и тезисов в онлайн-журналах по чат-ботам, других журналах и на международных научных конференциях.

Я бесконечно благодарен всем, кто поддерживал меня в этом невероятном путешествии. Я благодарю мою жену Джайантхи, моего сына Адвайта, моих родителей и всех в семье за их ободрение, поддержку и помощь в этом проекте. Я хочу поблагодарить моих друзей Приянку, Анупа, Тиру, Суреша, Джайю, Нару, Дивакара и многих других за то, что они побудили меня заняться этим проектом и следить за его прогрессом на всем протяжении.

Я хочу поблагодарить рецензентов книги за то, что они протянули мне руку, чтобы сделать книгу такой, какой она есть сейчас. Я также хочу поблагодарить всю команду издательства Packt Publishing, которая внесла огромный вклад, чтобы вы получили эту книгу.

Наконец, я хочу поблагодарить Бога за то, что он дал мне возможность узнать мои сильные стороны, и за его благодать и поддержку при написании этой книги.

О рецензенте

Вамси Венигалла – лидер в технологии с более чем 12-летним опытом работы в строительных организациях, стратегическом планировании, развертывании множества трансформационных платформ/продуктов, ИТ-программ, в управлении проектами, стратегии и трансформации (бизнес, люди, процессы и технологии) и в использовании результатов для различных бизнес-функций в мобильных, финансовых системах, снабжении, HR, розничной торговле и цепочке поставок в областях изготовления и производства продуктов питания.

Вамси закончил магистратуру по информатике в Университете штата Северная Каролина, Северная Каролина.

Предисловие

Создание чат-ботов – это весело. Хотя чат-боты – это лишь еще один вид программного обеспечения, они очень разные с точки зрения ожиданий пользователей. Чат-боты направлены на поддержание диалога. Эта способность обрабатывать язык заставляет проектировать их своего рода человеческую личность и интеллект, независимо от того, хотим мы этого или нет. Разработать программное обеспечение с личностью и интеллектом достаточно сложно и, следовательно, интересно.

Это книга для программистов, которые заинтересованы в изучении мира диалоговых пользовательских интерфейсов. Эта книга организована в виде восьми проектов чат-ботов, которые представят экосистему инструментов, методов, концепций и даже гаджетов, связанных с диалоговыми интерфейсами. Мы начнем изучать простые кнопочные чат-боты, перейдем к использованию текстовых сообщений и, наконец, голоса. Мы начнем с платформы без кода для создания нашего первого чат-бота и перейдем к изучению библиотек и служб, чтобы самостоятельно программировать модули. Каждая глава представляет собой уникальный проект диалогового интерфейса с источником данных. Вы будете учиться понимать данные, идентифицировать диалоговые задачи, проектировать диалог и, наконец, следовать пошаговым инструкциям для реализации интерфейса.

О чем эта книга

Глава 1 начинается с изучения того, что такое диалоговые пользовательские интерфейсы, их истории и последних разработок. Затем мы рассмотрим их базовую архитектуру, приложения и преимущества, а также некоторые факторы, в силу которых диалоговые интерфейсы так популярны.

В главе 2 вы узнаете, как создавать чат-боты без написания кода. Мы изучим платформу для разработки и хостинга под названием Chatfuel, на которой разработаем путеводитель по вашему городу. Мы разработаем и построим простого чат-бота с кнопками и научимся разворачивать чат-боты в Facebook Messenger, одном из самых популярных каналов для чатов.

Глава 3 рассказывает о построении чат-бота для того же канала, но на этот раз мы создадим его с нуля, используя язык Java и платформу Node.js. Мы создадим чат-бота, который расскажет нам о погоде, используя открытый источник данных под названием OpenWeatherAPI, разместим его в виде облачного приложения и свяжем с Facebook-приложением, которое подключается к странице Facebook. Мы изучим технологии Facebook Pages и SDK Messenger с точки зрения создания и развертывания чатов.

Глава 4 переходит от кнопочных чат-ботов к тем, которые распознают естественный язык. Мы рассмотрим популярный инструмент под названием Dialogflow (ранее известный как API.AI), который можно использовать для создания чатов, распознающих естественный язык. Мы построим бота – персонажа, который эмулирует популярную личность Альберта Эйнштейна. Затем мы рассмотрим, как интегрировать чат-бота в веб-сайт и Facebook Messenger.

В главе 5 переходим к более традиционному каналу связи – «службе коротких сообщений» (SMS). Мы проанализируем, как создать чат-бота для канала SMS. Используя службу данных, называемую Transport API, мы создадим чат-бота, информирующего о расписании поездов посредством SMS. Мы также интегрируем модуль Dialogflow в чат-бот, чтобы тот мог распознать, что говорит пользователь.

В главе 6 вы создадите чат-бота с помощью Microsoft Bot Builder. Это быстро набирающий популярность инструмент для создания ботов, который применяется для проектирования эффективного менеджера диалогов, а не создания его с нуля, как это было ранее. Мы создадим чат-бота для поиска ресторанов, использующего службу данных под названием Zomato. Затем встроим его в канал Skype.

Глава 7 объясняет совершенно другой канал взаимодействия – Twitter. Мы разработаем Twitter-бота, который сканирует твиты пользователей, анализирует их интересы и отправляет им персонализированные новости ежечасно, подключаясь к источнику данных, называемому News API.

В главе 8 мы рассмотрим развитие голосовых ботов, таких как Alexa. Затем создадим бота Alexa, чтобы поговорить о телепрограмме. Мы создадим инструмент, который получает доступ к данным телевизионной программы из службы данных TVMaze и отвечает на запросы пользователей в Amazon Echo.

Глава 9 посвящена продолжению работы над голосовыми ботами и направлена на создание действий для Ассистента Google. Этот инструмент позволит пользователям управлять своим списком дел с помощью устройства Google Home. Мы создадим облачное приложение, в которое будет интегрирована онлайн-служба Todoist для записи новых задач и выполнения текущих.

Приложение перечисляет различные статьи, список журналов, список конференций и т. д., где вы можете получить больше знаний о чат-ботах.

Что вам потребуется при чтении этой книги

Ниже перечислены некоторые инструменты, которые мы широко используем во всей книге. Я рекомендую вам установить их на свой компьютер и ознакомиться с ними, так как это поможет вам при выполнении проектов.

- Heroku – это облачная платформа, которая позволяет размещать веб-приложения в облаке. Чтобы начать работу, зарегистрируйтесь с бесплатной учетной записью на сайте **www.heroku.com** и установите консольный инструмент Heroku на странице **devcenter.heroku.com/articles/heroku-cli**. Вы также можете узнать, как развернуть приложения на платформе Heroku. Я рекомендую вам ознакомиться со справочными материалами по адресу **devcenter.heroku.com/start**.
- **Git** – это инструмент управления версиями, с которым вы, возможно, знакомы. Мы будем интенсивно использовать Git, поскольку это часть процесса Heroku. Вам нужно установить консольный инструмент Git в вашей системе. Вы найдете его на странице **git-scm.com/downloads**.

- **Node.js** – это среда выполнения JavaScript, в которой понадобится создавать веб-приложения в большинстве наших проектов. Вы найдете ее дистрибутив на странице **nodejs.org/en/download/**.
- **Java SDK и Eclipse**: Java 1.8 используется для создания чат-бота в главе 3. Вам нужно установить Java SDK и среду разработки Eclipse, чтобы выполнять инструкции в этой главе. Кроме того, вы можете запрограммировать чат-бота, используя Node.js или другой язык программирования, поддерживаемый Heroku. Вы найдете дистрибутив Java 1.8 на сайте **www.oracle.com/technetwork/java/javase/downloads/index.html**, а Eclipse – на сайте **www.eclipse.org/downloads/**.

Для кого эта книга

Эта книга предназначена для разработчиков, которые заинтересованы в создании интерактивных пользовательских интерфейсов/чат-ботов. Вы должны уметь работать с платформой Node.js и программировать на языке Java. Опыт использования библиотек Node.js, таких как `request` и `express`, и базовое понимание инструментов типа Git и Heroku CLI также будут полезны.

Условные обозначения

В этой книге вы увидите текст с разным форматированием, которым различаются разные виды информации. Ниже показано несколько примеров этих стилей и дано объяснение их смысла.

Фрагменты кода в тексте, имена таблиц базы данных, имена папок, имена файлов, расширения файлов, имена путей, фиктивные URL-адреса, ввод пользователя и твиттер-дескрипторы показаны следующим образом: «Создайте файл *index.js* и добавьте в него инструкцию `request`».

Блок кода оформляется так:

```
// Комментарий EddieBot
const express = require('express')
const bodyParser = require('body-parser')
const app = express()
app.set('port', (process.env.PORT || 5000))
```

Любой ввод или вывод в командной строке записан следующим образом:

```
npm init
```

Новые термины и важные слова выделены полужирным шрифтом. Слова, которые вы видите на экране, например в меню или диалоговых окнах, отображаются в тексте следующим образом: «Щелкните мышью по ссылке **Get Started** (Начать) в нижней части окна чата».

```
[зн] Примечания и важные заметки выглядят так.
[зн] Советы и хитрости выглядят так.
```

Загрузка кода примеров

Вы можете загрузить файлы с примерами кода для этой книги по адресу **github.com/PacktPublishing/Hands-On-Chatbots-and-Conversational-UI-Development**.

Глава 1. Введение

Мы вошли в эру интеллектуальных машин, в которой лидируют диалоговые интерфейсы. За последние пару лет мы были охвачены множеством новых видов машин и программного обеспечения, общеизвестных как боты. Боты – это автоматизированные аппаратные или программные средства, которые основаны на технологиях искусственного интеллекта (ИИ). Недавние разработки в алгоритмах машинного обучения, такие как глубокое обучение и углубленное закрепление, улучшили производительность задач ИИ, таких как автоматическое распознавание речи (ASR), распознавание естественного языка (NLU), синтез текста в речь (TTS) и распознавание изображений. Это сократило время пути человечества к технологической сингулярности, момент, когда искусственный интеллект превосходит естественный человеческий разум не по дням, а по часам.

Одной из долгосрочных целей в области ИИ является создание компьютерных систем, которые смогут проводить человекоподобные беседы с пользователями. Благодаря недавним достижениям в области технологий ИИ мы сейчас на один шаг ближе к достижению этой цели. Теперь уже не за горами, что мы можем взаимодействовать с устройствами и гаджетами в наших домах и офисах, используя только голос. Нам еще предстоит пройти долгий путь по созданию стандартов и цифровых сущностей, способных беспрепятственно разговаривать на естественном языке. Однако недавний всплеск интересов и огромные инвестиции в продвижение этих идей свидетельствуют о том, что мы находимся на пути к разработке такого глобального стандарта. Если вы в восторге от последних разработок в области ИИ и технологий автоматизации, эта книга для вас. Мы отправимся в путь к будущему, что гуру-дизайнер Марк Кертис называет разговорной сингулярностью, когда диалоговые устройства исчезают, а разговор между человеком и машиной беспрепятствен и естествен.

Это книга для программистов, которые начинают создавать диалоговые интерфейсы. Сегодня простые кнопочные чат-боты могут

быть созданы без необходимости написать хоть одну строку кода. С них мы и начнем. Мы постепенно перейдем к более сложным и гибким архитектурам и изучим каналы для использования, такие как Facebook Messenger, SMS и Twitter. Мы также рассмотрим инструменты для распознавания естественного языка и управления разговорами по мере продолжения. Наконец, мы закончим наш экскурс, создав ботов с голосовой связью на таких платформах, как Amazon Alexa и Google Assistant.

Диалоговые пользовательские интерфейсы

Диалоговые пользовательские интерфейсы так же стары, как и современные компьютеры. ENIAC, первый программируемый универсальный компьютер, был построен в 1946 году. В 1950 году Алан Тьюринг, британский компьютерный ученый, предложил измерить уровень интеллекта в машинах с использованием диалогового теста, называемого тестом Тьюринга. Тест состоял в том, что машина конкурировала с человеком как партнером по диалогу и судей-людей (еще одного человека). Судьи должны были взаимодействовать с каждым из двух участников (человеком и машиной), используя интерфейс текстового типа, который не похож на большинство современных приложений чата обмена сообщениями. В ходе беседы судьи должны были определить, кто из двух участников был машиной. Если, по меньшей мере, 30% судей не смогли провести различие между двумя участниками, считалось, что машина прошла тест. Это была одна из самых ранних концепций по диалоговым интерфейсам и их отношению к уровню интеллекта машин, которые обладают такими возможностями. Однако попытки создания подобных интерфейсов не были успешными в течение нескольких последующих десятилетий.

В течение примерно 35 лет, с 1980-х годов, графические пользовательские интерфейсы (GUI) были доминирующим способом взаимодействия с машинами. С недавним развитием в области ИИ и растущими ограничениями, такими как сокращение размеров гаджетов (от ноутбуков до мобильных телефонов), сокращение экранных объектов (смарт-часы), и необходимостью того, чтобы интерфейсы стали невидимыми (умный дом и роботы), диалоговые пользовательские интерфейсы снова становятся реальностью.

Например, лучшим способом взаимодействия с мобильными роботами, используемыми в умных домах, будет голос. Таким образом, система должна распознавать запросы пользователей и отвечать на естественном человеческом языке. Такие возможности систем помогут уменьшить наши усилия в изучении и понимании современных сложных интерфейсов.

Диалоговые пользовательские интерфейсы известны под несколькими названиями: интерфейсы естественного языка, разговорные диалоговые системы, разговорные чат-боты, интеллектуальные виртуальные агенты, виртуальные помощники и т. д. Фактическая разница между этими системами связана с интеграцией в них серверных элементов (например, баз данных или модулей задач/управления), модальностями (например, текстовыми, голосовыми и визуальными аватарами) и каналами, на которых они развернуты. Однако одним из общих моментов для этих систем является их способность взаимодействовать с пользователями в разговорной манере с использованием естественного языка.

Краткая история чат-ботов

Истоки современных чат-ботов можно проследить до 1964 года, когда Джозеф Вайзенбаум в Массачусетском технологическом институте (MTI) разработал чат-бота под названием Eliza. Он использовал простые правила разговора и перефразировал бóльшую часть того, что пользователи сказали, чтобы имитировать роджерианского терапевта. Хотя это показало, что наивных пользователей можно обмануть, если они думают, что разговаривают с фактическим терапевтом, сама система не понимала проблемы пользователя. После этого в 1991 году была присуждена премия Лобнера, которая призвала исследователей ИИ создавать чат-ботов, которые смогут пройти испытание Тьюринга и помочь в продвижении ИИ. Несмотря на то что они не прошли тест вплоть до 2014 года, многие известные чат-боты выиграли призы за победу в других ограниченных задачах. К ним относятся ALICE, JabberWacky, Rose и Mitsuku. Однако в 2014 году на конкурсе Тьюринга, посвященном 60-летию смерти Алана Тьюринга, чат-бот под названием Eugene Goostman, имитирующий 13-летнего ребенка, сумел обмануть 33% судей, тем самым выполнив тест. Язык разметки искусственного интеллекта (Artificial Intelligence Markup Language – AIML) и ChatScript были разработаны как способ для обозначения знаний и диалогового

контента для большинства этих чат-ботов. Скрипты, разработанные с использованием этих сценарных языков, могут затем передаваться в интерпретаторы для создания диалогового поведения. Чат-боты, разработанные для прохождения теста Тьюринга, были в основном болтливыми с одной целью – победить тест Тьюринга. В большинстве они не рассматривались как какое-либо продвижение в ИИ или шаг к созданию полезных диалоговых помощников.

С другой стороны, исследования в области искусственного интеллекта, в частности в области машинного обучения и обработки естественного языка, привели к появлению различных диалоговых интерфейсов, таких как системы ответов на вопросы, интерфейсы естественного языка для баз данных и разговорные диалоговые системы. В отличие от чат-ботов, построенных для прохождения теста Тьюринга, эти системы имели очень четкие цели. Системы ответа на вопросы обрабатывали естественные языковые вопросы и находили ответы в наборах неструктурированных текстовых данных. Интерфейсы естественного языка для баз данных (Natural Language Interfaces to Database Systems – NLIDBS) были интерфейсами для больших баз данных SQL, которые интерпретировали запросы к базе данных, создаваемые на естественном языке, таком как английский, преобразовывали их в SQL и возвращали ответ. Разговорные системы (Spoken Dialogue Systems – SDS) – это системы, которые могли бы поддерживать контекстные разговоры с пользователями для обработки диалоговых задач, таких как бронирование билетов, контроль других систем и обучение учащихся. Это были предшественники современных чат-ботов и диалоговых интерфейсов.

Последние разработки

В 2011 году компания Apple выпустила интеллектуального помощника под названием Siri в составе своих устройств iPhone/iPad. Siri была смоделирована как персональный помощник пользователя, выполняя такие задачи, как вызов, чтение сообщений и настройка аварийных сигналов и напоминаний. Это одно из самых значительных событий в недавнем прошлом, которые перезагрузили историю диалоговых интерфейсов. В первые дни работы пользователи Siri использовали ее всего несколько раз в месяц для выполнения таких задач, как поиск в Интернете, отправка SMS и совершение телефонных звонков. Несмотря на новизну, Siri предназначалась для работы с гораздо бóльшим количеством функций, которые были добавлены

в последующие годы. В первые дни у Siri было много клонов и последователей на Android и других платформах для смартфонов. Большинство из них было смоделировано в качестве помощников и доступно в виде мобильных приложений.

В том же году (2011 г.) IBM представила Watson, систему ответов на вопрос, которая участвовала в игровом шоу под названием Jeopardy и выиграла его у предыдущих победителей, Брэда Раттера и Кена Дженнингса. Это ознаменовало собой веху в истории ИИ, поскольку Watson смог обрабатывать вопросы естественного языка и отвечать на них в режиме реального времени. С тех пор Watson был переработан в набор инструментальных средств с набором инструментов когнитивного обслуживания для распознавания естественного языка, анализа настроений, управления диалогами и т. д.

Следуя Siri и Watson, следующее крупное событие появилось в Microsoft в 2013 году, когда они представили Cortana как стандартную функцию на смартфонах под управлением операционной системы Windows, а затем в 2015 году и на платформе Windows 10. Как и Siri, Cortana была персональным помощником, который управлял такими задачами, как настройка напоминаний, ответы на вопросы и т. д.

В ноябре 2014 года Amazon пригласил своих премьер-членов опробовать своего персонального помощника по имени Alexa. Alexa была доступна на собственном продукте Amazon под названием Echo. Echo был первым в своем роде умным динамиком, который размещал в нем помощника, как «призрак» в машине. Хотя он назывался динамиком, на самом деле это был крошечный компьютер с голосом в качестве единственного интерфейса, в отличие от смартфонов, планшетов и персональных компьютеров. Пользователи могли говорить с Alexa, используя голос, просить ее выполнять такие задачи, как установка напоминаний, воспроизведение музыки и т. д.

В апреле 2016 года социальная сеть Facebook объявила, что открывает свою популярную платформу Messenger для чат-ботов. Это был радикально иной подход к диалоговым интерфейсам, по сравнению с Siri, Alexa и Cortana. В отличие от этих персональных помощников, объявление Facebook привело к созданию пользовательских и фирменных чат-ботов. Эти боты очень похожи на Siri, Cortana и Alexa, но могут быть настроены на потребности их создающего бизнеса. Теперь чат-боты могли нарушить работу нескольких рынков, включая обслуживание клиентов, продажи, маркетинг, техническую поддержку и т. д. Многие платформы обмена сообщениями, такие

как Skype, Telegram и другие, также стали доступны для интеграции в чат-боты в то же время.

В мае 2016 года Google анонсировала Assistant, свою версию персонального чат-бота, доступную на нескольких платформах, таких как приложение Allo и Google Home (умный динамик, такой как Echo). Все помощники, такие как Siri, Cortana, Alexa и Google Assistant, также открылись в качестве каналов для диалоговых возможностей сторонних разработчиков. Таким образом, теперь можно персонализировать программное обеспечение Alexa и Google Assistant, добавив диалоговые возможности (называемые навыками или действиями) из библиотеки решений сторонних разработчиков. Так же, как бренды могут создавать своих собственных чат-ботов для различных служб обмена сообщениями (например, Skype и Facebook Messenger), они могут развивать навыки для Alexa или действия для Ассистента Google. В 2018 году был выпущен смарт-динамик компании Apple, Homepod, работающий под управлением голосовой платформы Siri.

Параллельно с этими событиями также наблюдается значительный рост с точки зрения инструментов, доступных для создания и размещения чат-ботов. В последние два года отмечен экспоненциальный рост инструментов для разработки, моделирования, сборки, развертывания, управления и монетизации чатов. Это привело к созданию экосистемы, которая разрабатывает и создает пользовательские диалоговые интерфейсы для предприятий, благотворительных организаций, правительственных и других организаций по всему миру.

Архитектура диалогового пользовательского интерфейса

В этом разделе мы рассмотрим базовую архитектуру диалогового интерфейса (рис. 1.1.).

Основным модулем диалогового интерфейса является менеджер диалогов. Этот модуль контролирует поток разговора. Он берет семантическое представление того, что пользователь говорит, как ввод и решает, каков должен быть ответ системы. Он поддерживает представление диалогового контекста в той или иной форме, например набор пар значений ключа для выполнения содержательного разговора в течение нескольких обменов между пользователем и системой.

Семантическое представление пользовательского ввода возможно напрямую по нажатии кнопки. В системах, которые распознают язык,

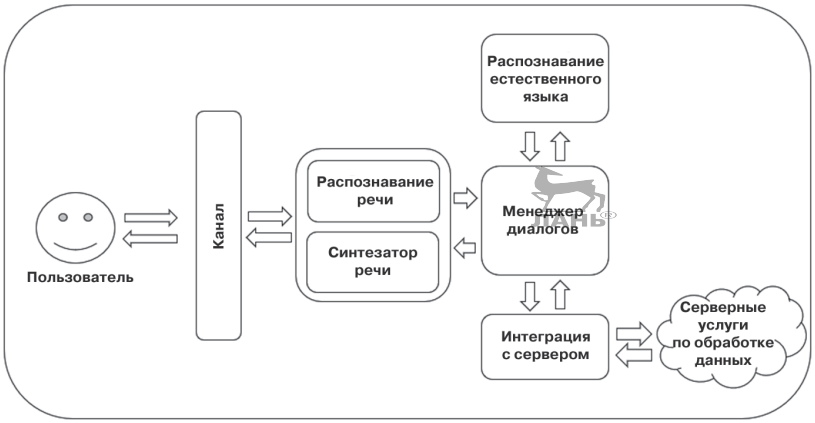


Рис. 1.1

пользовательские высказывания переводятся в семантическое представление, состоящее из пользовательских целей и параметров (слотов и сущностей), с помощью модуля распознавания естественного языка. Возможно, этот модуль необходимо предварительно обучить, чтобы распознать набор пользовательских целей, определенных разработчиком, относящихся к доступным диалоговым задачам.

Интерфейсы с поддержкой голоса, которые принимают голосовые инструкции пользователя, также нуждаются в модуле распознавания речи, который может транслировать речь в текст, прежде чем передавать его в модуль распознавания естественного языка. Симметрично, с другой стороны, необходим модуль синтеза речи (или модуль преобразования текста в речь), который преобразует текстовый ответ системы в речь.

Менеджер диалогов должен взаимодействовать с серверными модулями. Это может быть база данных или онлайновый источник данных, который запрашивается для ответа на вопрос пользователя (например, телевизионное расписание), или онлайн-служба для выполнения инструкций пользователя (например, бронирование билета).

Канал – это место, где чат-бот ожидает пользователя. В зависимости от канала может быть один или несколько модулей, из которых состоит этот слой. Например, если чат-бот находится в Facebook Messenger, этот слой состоит из Facebook-страницы и Facebook-приложения, которое соединяется с остальными модулями чат-бота, оформленными в виде веб-приложения.

Классификация

Диалоговые пользовательские интерфейсы применяются в различных сценариях. Их приложения можно разделить на две категории: корпоративные помощники и персональные помощники.

Корпоративные помощники – это чат-боты и другие диалоговые пользовательские интерфейсы, смоделированные после опроса представителей службы поддержки клиентов и помощников магазина. Как и представители обслуживания клиентов, боты привлекают клиентов к разговору, выполняя маркетинговые, продажные и вспомогательные задачи. Большинство чат-ботов, развернутых по каналам, таким как Facebook Messenger, Skype, Slack и многие другие, являются корпоративными помощниками. Они спроектированы и построены для выполнения задач, которые будут выполнять помощники магазина и представители по обслуживанию клиентов. Корпоративные помощники разрабатываются во многих секторах бизнеса, автоматизируя различные диалоговые задачи.

С другой стороны, персональные помощники, например Alexa, Siri и Cortana, которые играют роль личного помощника пользователя, выполняют такие задачи, как управление календарем, отправка текстов, прием вызовов и воспроизведение музыки. Возможности этих личных помощников могут быть расширены. Например, Alexa допускает такое расширение, позволяя разработчикам создавать навыки, которые пользователи могут выбрать для добавления в свою собственную Alexa. Таким образом, бренды могут развивать навыки Alexa или действия для Ассистента Google, которые позволят Alexa и Ассистенту взаимодействовать с ИТ-услугами бренда и выполнять такие задачи, как размещение заказов, проверка статуса доставки и многое другое. Например, популярные бренды, такие как PizzaHut, Starbucks и Domino, разработали навыки, которые могут быть задействованы на Alexa.

Приложения

Хотя чат-боты разрабатывались в течение, по крайней мере, нескольких десятилетий, они не стали основными каналами для взаимодействия с клиентами до недавнего времени. За последние два года благодаря серьезным усилиям таких гигантов индустрии, как Apple, Google, Microsoft, Facebook, IBM и Amazon, а также их последующим инвестициям в разработку наборов инструментов,

чат-ботов и диалоговых интерфейсов они стали серьезным соперником других каналов связи с клиентами. За это время чат-боты были применены в различных секторах и в различных диалоговых сценариях в этих секторах: розничном, банковском и финансовом, правительственном, медицинском, юридическом и независимом секторах и многом другом.

В розничной торговле чат-боты применяются для диалогов по поводу маркетинга продукта, привлечения бренда, поддержки продукта, продаж и поддержки. Браузерные чат-боты предлагают советы и рекомендации лояльным клиентам бренда, связанные с использованием продуктов, продаваемых брендом. Например, чат-бот Sephora советует пользователям подобрать идеальную помаду. Точно так же чат-бот TK-Maxx помогал пользователям выбирать подарки для своих друзей и семьи во время Рождества 2016 года. Одним из первых розничных продавцов для изучения чатов для продаж был H&M. Чат-бот H&M помог пользователям просмотреть каталог продуктов и добавить продукты в свои корзины. Производители автомобилей, такие как Tesla, Kia и Mercedes, разработали чат-ботов, которые помогают водителям получить информацию о своих автомобилях.

Чат-боты очень эффективны в банковской и финансовой индустрии. Банкинг был одним из первых секторов, которые экспериментировали с диалоговыми интерфейсами. Банковские чат-боты отвечают на общие вопросы о финансовых продуктах, безопасном банковском обслуживании и т. д., а также предоставляют конкретную и персонализированную информацию по счетам пользователей. Многие крупнейшие банки и поставщики финансовых услуг, включая Bank of America, ICICI bank, HSBC, Royal Bank of Scotland, Capital One, MasterCard и т. д., развернули чат-боты, чтобы помочь своим клиентам. Многие финансовые институты создают чат-ботов, выступающих в качестве финансового помощника для пользователей. Ernest.ai и Cleo – это чат-боты, которые соединяют вас с вашими банковскими счетами и говорят о ваших расходах, балансах, а также дают советы по экономии денег. Чат-боты широко используются в страховом секторе, где выступают в качестве ассистентов, способных получить для вас индивидуальные котировки (например, SPIXII[1]).

Чат-боты применяются в юридическом, медицинском, правительственном и независимом секторах. Чат-бот под названием DoNotPay

[1] www.insly.com/en/blog/chatbot-is-the-future-of-automated-insurance/.

помог людям оспорить парковочные билеты в Лондоне и Нью-Йорке в более чем 160 000 случаев. После этого было разработано много новых чат-ботов, позволяющих помочь людям получить доступ к судебным и юридическим услугам: оценка преступности (LawBot), регистрация бизнеса (LawDroid), помощь арендаторам (RentersUnion), помощь в юридических вопросах и документации (Lisa, LegaliBot, Lexi, DocuBot), поиск адвокатов (BillyBot).

В независимом секторе чат-боты использовались для распространения информации о проблемах благотворительности. Stoptober[1] – это чат-бот Facebook, который был разработан Национальными службами здравоохранения (NHS) в Великобритании, чтобы помочь курильщикам бросить курить. Другой чат-бот, Yeshi[2], был разработан, чтобы привлечь внимание к водному кризису в Эфиопии. Чат-боты начинают вносить свой вклад в здравоохранение. Чат-боты Your.MD и HealthTap были разработаны для диагностики проблем со здоровьем на основе симптомов. Emily[3] – это чат-бот, разработанный LifeFolder, чтобы помочь принять решения после смерти (например, юридическая документация, поддержка жизнеобеспечения, донорство органов и многое другое).

Чат-боты используются не только для того, чтобы помочь клиентам, но и внутри, для сотрудников. Чат-боты в некотором смысле становятся коллегами, помогая сотрудникам в работе с повторяющимися, рутинными и скучными задачами. Службы обмена сообщениями, такие как Slack и Microsoft Teams, поощряют чат-ботов на своих платформах для автоматизации служебной связи. Эти боты нацелены на привлечение сотрудников в чат для решения интересных и важных задач. Например, есть боты, способные помочь коллегам поделиться знаниями (Obie.ai), получить доступ к другим службам, таким как GDrive (WorkBot), настроить встречи (Meekan), обсудить обед (LunchTrain) и даже помочь в принятии решений (ConcludeBot, SimplePoll).

Если вам интересно узнать больше вариантов использования, я бы порекомендовал вам взглянуть на некоторые сайты с каталогами ботов, такие как botlist.co и www.chatbots.org, где вы найдете больше информации и вдохновения.

---

[1] www.marketingweek.com/2016/09/20/stoptober-uses-facebook-messenger-bot-to-help-people-quit-smoking/.

[2] www.akqa.com/work/lokai/walk-with-yeshi/.

[3] medium.com/life-folder/introducing-emily-the-chatbot-that-talks-about-death-97b390119cce.

Инструменты разработчика

За последние несколько лет экосистема инструментов и услуг вокруг идеи диалоговых интерфейсов выросла. Существует множество инструментов, которые мы можем использовать для дизайна, разработки и управления чат-ботами.

Инструменты прототипирования

Инструменты прототипирования используются, чтобы показать клиентам, как будет выглядеть и вести себя чат-бот. Это инструменты, которые вы можете использовать во время конструирования диалога, после того как запишете идею и примеры диалогов между пользователем и ботом на обратной стороне салфетки. Инструменты прототипирования позволяют визуализировать беседу между пользователем и ботом и продемонстрировать динамику разговора в обоих направлениях. BotSociety.io **(botsociety.io)** и BotMock.com **(botmock.com)** – примеры популярных инструментов прототипирования. Некоторые из этих инструментов позволяют экспортировать прототип и создавать видео.

Каналы

Каналы – это места, где пользователи могут взаимодействовать с чат-ботом. Существует несколько каналов развертывания, по которым ваши боты могут быть доступны пользователям. К ним относятся службы обмена сообщениями, такие как Facebook Messenger, Skype, Kik, Telegram, WeChat и Line; офисные и командные чаты, такие как Slack, Microsoft Teams и многие другие; традиционные каналы, такие как веб-чат, SMS и голосовые звонки; и умные динамики, такие как Amazon Echo и Google Home. Выберите канал, базируясь на ваших пользователях и требованиях проекта. Например, если вы создаете чат-бота для потребителей, Facebook Messenger может быть лучшим выбором из-за растущего числа пользователей, которые применяют услугу, чтобы поддерживать связь с друзьями и семьей. Добавить вашего чат-бота в их список контактов может быть проще, чем заставить их загрузить ваше приложение. Если пользователю необходимо взаимодействовать с ботом голосом в домашней или офисной среде, каналы умных динамиков могут быть идеальным выбором. И наконец, есть инструменты, которые могут подключать чат-ботов ко многим каналам одновременно (например, интеграция Dialogflow, служба MS Bot и Smooch.io и т. д.).

Средства разработки чат-ботов

Существует множество инструментов, которые можно использовать для создания чат-ботов без необходимости работы с кодом напрямую: Chatfuel, ManyChat, Dialogflow и т. д. Chatfuel позволяет дизайнерам создавать поток диалога с использованием визуальных элементов. С помощью ManyChat вы можете выстроить поток, используя визуальную карту, называемую FlowBuilder. Диалоговые элементы, такие как высказывания бота и кнопки ответа пользователя, могут быть настроены путем простого перетаскивания элементов интерфейса. Dialogflow можно использовать для создания чат-ботов, которые требуют расширенных методов распознавания естественного языка для взаимодействия с пользователями.

С другой стороны, существуют языки сценариев, такие как язык разметки искусственного интеллекта (Artificial Intelligence Markup Language – AIML), ChatScript и RiveScript, которые применяются для создания чат-ботов. Эти сценарии будут содержать диалоговый контент и поток, которые затем должны быть отправлены программе интерпретатора или механизму правил, чтобы оживить чат-бота. Интерпретатор решает, как продолжать диалог, сопоставляя пользовательские высказывания с шаблонами в сценариях. Хотя с помощью этого подхода легко создавать диалоговых чат-ботов, становится сложно создавать транзакционные чаты, не генерируя явных семантических представлений пользовательских высказываний. PandoraBots – популярная веб-платформа для создания чат-ботов AIML.

В качестве альтернативы существуют библиотеки SDK, которые можно использовать для создания чатов: MS Bot Builder, BotKit, BotFuel и т. д. SDK предоставляется на одном или нескольких языках программирования, чтобы помочь разработчикам в создании основного диалогового модуля управления. Возможность настройки диалогового менеджера позволяет разработчикам гибко форматировать беседы и интегрировать в бота задачи резервного копирования, без излишнего кода и сценарных платформ. После создания менеджер диалогов может быть подключен к другим службам, таким как распознавание естественного языка, чтобы воспринимать пользовательские высказывания.

Аналитика

Как и другие цифровые решения, чат-боты могут извлечь выгоду из сбора и анализа статистики их использования. Хотя вы можете

создать особую платформу аналитики с нуля, вы также можете использовать готовые инструменты, которые сейчас широкодоступны. Существуют многие готовые инструменты для аналитики, которые можно подключить к чат-боту, используя входящие и исходящие сообщения, которые могут быть зарегистрированы и изучены. Эти инструменты сообщают разработчикам и менеджерам чат-бота о типах разговоров, которые фактически происходят между пользователями и чат-ботом. Данные предоставят полезную информацию, такую как популярные диалоговые инструкции, места, где диалог прерывается, высказывания, которые бот не распознал, и запросы, для которых по-прежнему требуется масштабирование чат-ботов. Dashbot.io, BotAnalytics и Chatbase Google – это несколько аналитических инструментов, которые вы можете использовать для анализа производительности вашего чат-бота.

Распознавание естественного языка

Чат-боты могут быть построены без необходимости распознавать высказывания пользователя. Однако добавление возможностей распознавания естественного языка не очень сложно. Это одно из отличительных качеств, которое отличает чат-ботов от их цифровых аналогов, таких как веб-сайты и приложения с визуальными элементами. Существует множество модулей распознавания естественного языка, которые доступны как облачные службы. Крупные ИТ-игроки рынка, такие как Google, Microsoft, Facebook и IBM, создали инструменты, которые вы можете подключить к своему чат-боту. Google Dialogflow, Microsoft LUIS, IBM Watson, SoundHound и Facebook Wit.ai – вот некоторые из NLU-инструментов (Natural Language Understanding – распознавание естественного языка), которые вы можете протестировать. Мы рассмотрим инструмент Dialogflow в последующих главах.

Службы каталогов

Одна из задач бота – заставить пользователей найти и использовать его. Чат-боты не так популярны, как веб-сайты и мобильные приложения, поэтому потенциальный пользователь может не знать, куда перейти, чтобы найти бота. Как только ваш чат-бот будет развернут, вы должны помочь пользователям найти его. Есть каталоги, в которых перечислены боты в разных категориях. Chatbots.org – одна из

старейших служб каталогов, которая с 2008 года публикует списки чат-ботов и виртуальных помощников. Другими популярными службами являются Botlist.co, BotPages, BotFinder и ChatBottle. Эти каталоги классифицируют ботов с точки зрения цели, сектора, поддерживаемых языков, стран и т. д. В дополнение к этим такие каналы, как Facebook и Telegram, имеют свои собственные каталоги для ботов, размещенных на их канале. В случае с Facebook вы можете помочь пользователям найти свой бот Messenger, используя службу **Discover** (Интересное).

Монетизация

Чат-боты созданы для многих целей: для осведомления, поддержки клиентов после продажи товаров/услуг, предоставления платных услуг и многих других. В дополнение ко всему этому чат-боты с интересным контентом могут долгое время привлекать пользователей и применяться для того, чтобы заработать деньги за счет целенаправленной персонализированной рекламы. Такие услуги, как CashBot.ai и AddyBot.com, могут интегрироваться в вашего чат-бота для отправки таргетированных рекламных объявлений и рекомендаций пользователям, и когда пользователи будут участвовать, ваш чат-бот будет зарабатывать деньги.

Вышеупомянутое – не исчерпывающий список инструментов, а также услуг. Эти инструменты развиваются со временем, по мере того как чат-боты находят свою нишу на рынке. Этот список должен дать вам представление о том, как многомерна экосистема, помочь вам исследовать ее и дать пищу вашему уму.

Выгоды

Диалоговые пользовательские интерфейсы дают лучшее из обоих миров: человекоподобное естественное взаимодействие в сочетании с преимуществами цифровых технологий.

- **Доступность**: как и любые другие автоматизированные цифровые технологии, диалоговые интерфейсы недороги и доступны круглосуточно и без выходных. Это похоже на то, что кто-то постоянно сидит в веб-чате, чтобы клиенты всегда имели возможность получить ответы.
- **Персонализированный опыт**: в отличие от веб-сайтов и приложений для смартфонов, чат-боты обеспечивают крайне

персонализированный подход благодаря разговорной природе взаимодействия. Индивидуальные настройки беседы предоставляют широкие возможности для распознавания и адаптации к целям, предпочтениям и ограничениям пользователя.

- **Низкая стоимость**: чат-боты – это цифровые решения, следовательно, они предоставляют услуги поддержки клиентов, по меньшей мере, в десять раз дешевле, чем люди, выполняющие те же самые задачи.
- **Стабильность**: чат-боты могут непрерывно работать в службах, чего трудно достичь с помощью операторов-людей и что может быть очень важно в определенных секторах.
- **Быстрое время отклика**: в отличие от человеческих систем, время ответа в чат-ботах намного меньше. Пользователям больше не нужно ждать, пока их звонок будет принят, и во время разговора чат-бот будет отвечать быстрее человека, особенно когда человеческим операторам поручено одновременно вести более одного чата (иногда до пяти). Способность чат-ботов обрабатывать одновременные диалоги также устраняет узкое место ограниченной пропускной способности поддержки пользователей и, следовательно, помогает предприятиям масштабироваться.
- **Масштабирование**: чат-боты могут легко масштабироваться, чтобы обрабатывать растущий и сезонный трафик, что нелегко сделать при использовании штата людей-советников. Праздничный сезон может особенно повысить спрос на поддержку клиентов. В такие моменты чат-боты могут использоваться для обработки низкоприоритетных и простых задач, тем самым уменьшая нагрузку на живых консультантов, и применяться разумно для обработки особо ценных разговоров.

Чат-боты никуда не денутся

Технологии диалогового пользовательского интерфейса в настоящее время являются одной из главных тенденций в технологическом бизнесе. Большинство крупных брендов начали формулировать свою стратегию чат-ботов в рамках своей более крупной стратегии ИИ и автоматизации. Такие инновационные решения, как чат-боты, умные динамики и беспилотные автомобили, приводят к важным политическим решениям. Мир готовится к наступлению технологий автоматизации, которые готовы заменить людей в повторяющихся и структурированных задачах.

Недавний рост популярности чат-ботов был вызван многими факторами:

- представители поколения Y неуклонно двигались к чатам в качестве своего предпочтительного канала для взаимодействия с брендами. Опросы клиентов показывают, что люди чаще хотят использовать каналы веб-чата, когда они доступны, по сравнению с другими традиционными каналами, такими как электронная почта и телефон, для связи с предприятиями;
- рост приложений обмена сообщениями в чатах на смартфонах и других устройствах превзошел использование приложений для социальных сетей, таких как Facebook и Twitter. Теперь люди тратят больше времени на приложения для обмена сообщениями, беседуют с друзьями, семьей, коллегами и даже с сотрудниками компаний;
- растущий покупательский спрос на чаты оказывает огромное давление на бренды. Отсутствие достаточных квалифицированных человеческих ресурсов для обработки растущего трафика чатов также является важным фактором роста популярности чат-ботов;
- наличие инструментов виртуального обслуживания для распознавания естественного языка, распознавания речи, синтеза речи, управления диалогами, аналитики и т. д. сделало дизайн и развитие чат-ботов проще, чем это было несколько лет назад;
- открытие каналов обмена сообщениями и инновационные новые возможности, такие как интеллектуальные динамики, сделали доставку услуг через чат-боты реальностью. Тот факт, что растет интерес к приложениям и устройствам обмена сообщениями, таким как умные динамики, дает привлекательную возможность для брендов создавать чат-боты пользователям, которые уже используют эти каналы.

Существует несколько обзоров и статистические данные, которые демонстрируют, что диалоговые интерфейсы никуда не денутся. В списке ниже перечислены некоторые из самых убедительных результатов опросов и прогнозов, которые показывают, что у чат-ботов долгосрочная перспектива:

- компания Gartner прогнозирует, что к 2021 году бренды, которые разрабатывают свои веб-сайты с учетом голосового и визуального поиска, повысят свой доход на 30% и что более 50% компаний будут тратить больше на чат-боты, чем на традиционные мобильные приложения[1];

[1] www.gartner.com/smarterwithgartner/gartner-top-strategic-predictions-for-2018-and-beyond/.

- в опросе Oracle 80% респондентов (руководители уровня C) заявили, что планируют внедрить чат-ботов к 2020 году, если они еще не были введены[1];
- компания Juniper прогнозирует, что использование чат-ботов приведет к ежегодной экономии затрат в 8 млрд долларов США к 2022 году, по сравнению с 20 миллионами долларов США в 2017 году[2];
- опрос Hubspot показал, что около 47% потребителей готовы делать покупки предметов через чат-бот, а около 40% не волнует, разговаривают ли они с чат-ботом или человеком, пока они получают помощь легко и быстро[3];
- наконец, согласно недавнему отчету Grand View Research, глобальный рынок чат-ботов достигнет ошеломляющих 1,25 млрд долл. США к 2025 году, увеличив совокупный среднегодовой темп роста на 24,3% с 190 млн долл. США в 2016 году[4].

Приступим!

Итак, вы готовы начать работу и создать пару чат-ботов? Надеюсь, я дал вам хорошее представление о мире чат-ботов в этой главе. Мы прошлись по историческим и последним событиям, поговорили о классификации чат-ботов, их применении в различных секторах, преимуществах, будущем и базовой архитектуре. В следующих восьми главах я расскажу вам о нескольких инструментах, методах и концепциях, которые позволят вам создавать удивительные диалоговые интерфейсы. Поехали!

> Инструменты, которые мы собираемся изучить в этой книге, постоянно развиваются, и вы можете встретиться с их версией при работе над проектами. Я постарался объяснить основные понятия в каждой главе, чтобы вы могли решить, как действовать на основе вашего понимания концепции и использовать собственный подход, а не просто выполнять инструкции дословно. Для получения последней информации о развитии темы обратитесь к ссылкам на документацию, которые указаны в конце каждой главы.

[1] uk.businessinsider.com/80-of-businesses-want-chatbots-by-2020-2016-12.
[2] www.juniperresearch.com/researchstore/innovation-disruption/chatbots/retail-ecommerce-banking-healthcare.
[3] research.hubspot.com/reports/artificial-intelligence-is-here.
[4] www.grandviewresearch.com/industry-analysis/chatbot-market.

Глава 2. Путеводитель по городу

Познакомившись с чат-ботами и узнав их краткую историю, в этой главе мы создадим первого чат-бота. Он будет действовать как путеводитель по городу для пользователя. Мы используем популярные туристические диалоги, такие как изучение города, выяснение того, что нужно делать, и поиск мест, где можно поесть, как примерные задачи. Мы рассмотрим, как использовать различные инструменты в наборе инструментов для создания чата, способного выполнять эти диалоговые задачи.

Рассмотрим популярный инструмент Chatfuel и научимся создавать чат-ботов с нуля. Chatfuel – это инструмент, который позволяет создать чат-бота без необходимости трогать исходный код. Это веб-инструмент с графическим редактором, который позволяет пользователю создать чат-бота модульным способом. В дополнение к созданию чат-ботов он также позволяет разработчикам анализировать и управлять производительностью бота с помощью инструментов управления и аналитики.

В этой главе мы рассмотрим строительные блоки чат-бота. Затем мы построим путеводительного чат-бота с нуля и разместим его на платформе Facebook Messenger. Мы рассмотрим различные функции, предоставляемые Chatfuel, чтобы расширить возможности чат-бота, такие как интеграция в него базы данных и настройка широковещательных сообщений. Наконец, мы кратко рассмотрим пакет аналитики. В ходе этой главы вы:

- поймете основы Chatfuel;
- усвоите концепцию потока диалога и навигации;
- разберете строительные блоки: блоки, карты, плагины и атрибуты;
- научитесь управлять контекстами с использованием пар ключ/значение и обрабатывать пользовательские высказывания;
- узнаете, как создавать чат-ботов и интегрировать в них веб-данные для обработки на стороне сервера;
- узнаете, как интегрировать чат-бота в Facebook Messenger.

Начало работы

Приступим. Чтобы создать учетную запись в Chatfuel, перейдите на страницу **chatfuel.com**.

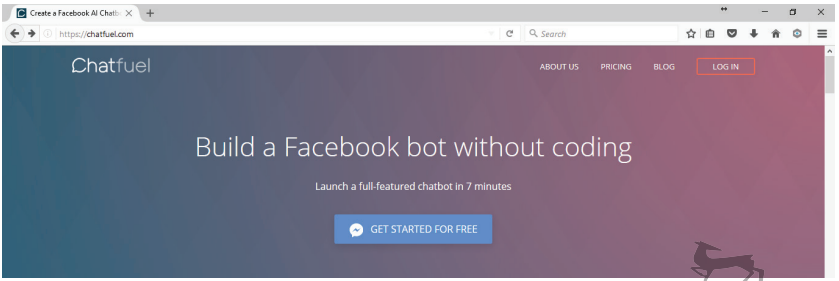


Рис. 2.1

Щелкните мышью по ссылке **Get started for free** (Начать бесплатно). Помните, что инструментарий Chatfuel в настоящее время бесплатен. Далее доступен один из следующих двух вариантов:

- если вы авторизованы на сайте Facebook, появится запрос на разрешение связать учетную запись Chatfuel с вашей учетной записью Facebook рис. 2.2;

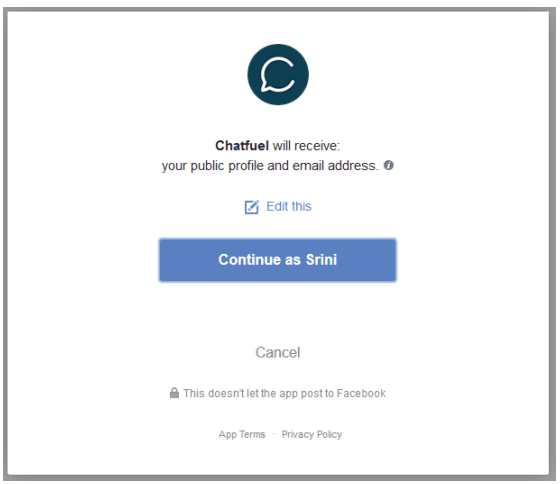


Рис. 2.2

- если вы не авторизованы на сайте Facebook, появится запрос авторизации на сайте Facebook, прежде чем будет запрошено разрешение.

Chatfuel запросит разрешение на связь с Facebook для развертывания ботов. Для этого требуется разрешение на использование вашей учетной записи Facebook – см. рис. 2.3.

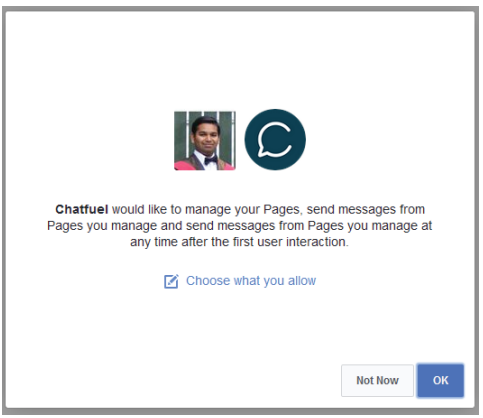


Рис. 2.3

Вы все настроили для создания своего первого бота – рис. 2.4.

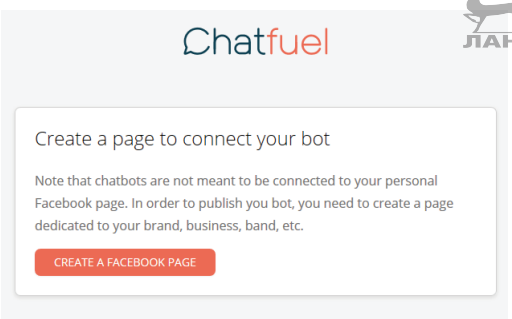


Рис. 2.4

Создание вашего первого бота

Боты Chatfuel могут быть размещены на двух платформах развертывания: Facebook Messenger и Telegram. Сначала создадим чат-бота

для платформы Facebook Messenger. Для этого нам нужно создать страницу Facebook. Любой чат-бот на платформе Facebook Messenger должен быть прикреплен к странице. Вот как можно создать страницу в Facebook:

1. Перейдите на страницу **www.facebook.com/pages/create/**.
2. Выберите категорию, соответствующую содержимому страницы. Я выбрал пункт **Brand or Product** (Компания или бренд).
3. Присвойте странице название. В нашем случае укажем текст **Get_Around_Edinburgh**. Обратите внимание, что Facebook не позволяет легко изменять названия страниц. Поэтому выбирайте название осмысленно.
4. Как только страница будет создана, вы увидите, что Chatfuel запрашивает разрешение на подключение к странице (рис. 2.5.).

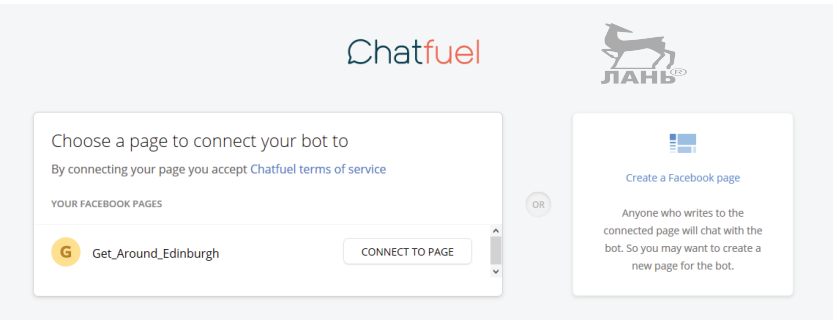


Рис. 2.5

5. Выберите пункт **Connect to page** (Подключить к странице). Откроется страница редактора ботов.
6. Указано название бота **My First Bot**. Ему присвоен URL-адрес на платформе Messenger, который вы можете увидеть рядом с названием. URL-адреса на платформе Messenger начинаются с букв m.me. Обратите внимание, что бот имеет встроенное приветственное сообщение. Слева вы видите главное меню с несколькими вкладками, причем команда **Build** (Конструирование) выбрана по умолчанию. Мы рассмотрим другие вкладки в следующих разделах главы – рис. 2.6.
7. Щелкните мышью по URL-адресу бота, чтобы начать свой первый диалог с ботом. Откроется страница Facebook Messenger во вкладке вашего браузера – рис. 2.7.

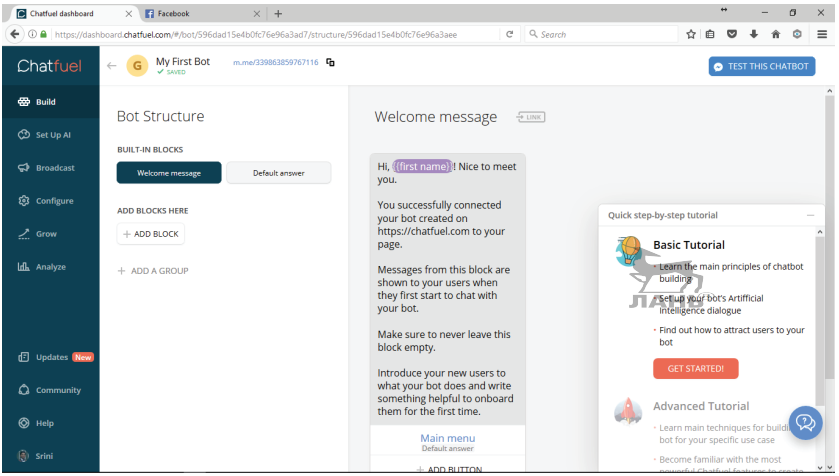


Рис. 2.6

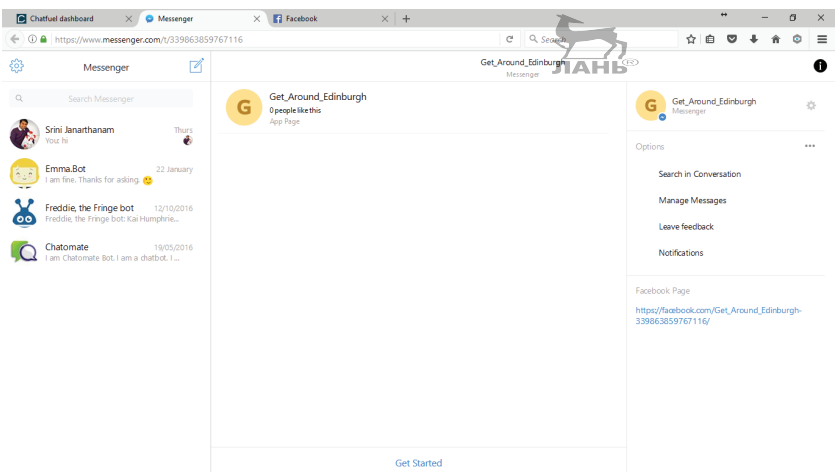


Рис. 2.7

8. Чтобы начать диалог, щелкните мышью по ссылке **Get Started** (Начать) внизу окна чата.
9. Ваш диалог только что начался. Бот отправил вам приветственное сообщение – рис. 2.8.

Обратите внимание, что бот приветствует вас вашим именем. Это потому, что вы предоставили боту доступ к вашей информации на Facebook.

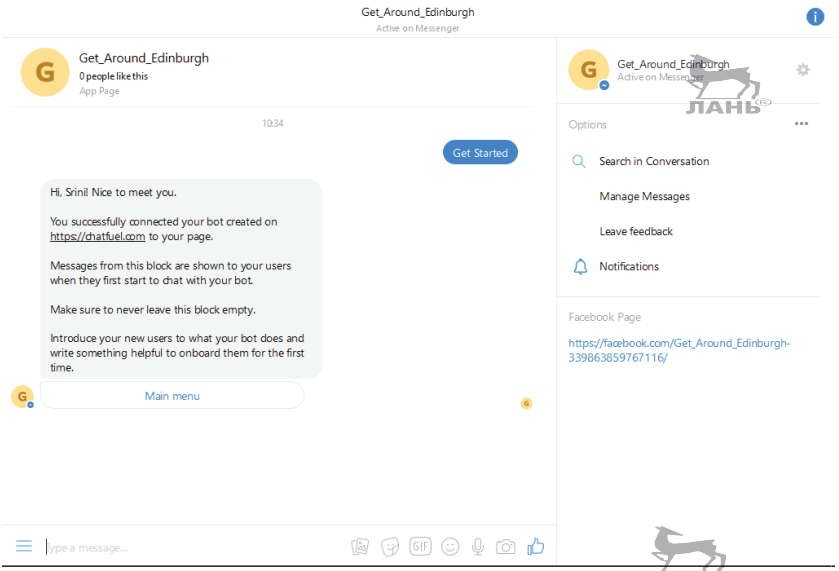


Рис. 2.8

Теперь, когда вы создали своего первого бота и поговорили с ним, добро пожаловать в мир чат-ботов!

Базовые элементы

Прежде чем мы перейдем к созданию нашего бота, взглянем на основные строительные кирпичики:

- блоки;
- карты;
- кнопки;
- плагины;
- атрибуты.

Блоки

Вернитесь в редактор и посмотрите вкладку **Bot Structure** (Структура бота). В разделе **Bot Structure** (Структура бота) вы найдете два типа блоков – встроенные и пользовательские. **Welcome message** (Приветственное сообщение) и **Default answer** (Ответ по умолчанию) – это два встроенных блока. Мы создадим пользовательские

блоки для реализации возможности беседы с чат-ботом. Каждый блок можно рассматривать как сегмент ответа чат-бота. Каждый блок может содержать одну или несколько карт.

Карты

Карты – это конструкции, которые используются для отправки сообщений на платформу развертывания. Существует много типов карточек. Текстовые карты являются основными; они содержат текстовые сообщения и дополнительные кнопки. Другие карты включают изображения, быстрые ответы (массивы кнопок), списки и галерею. Мы рассмотрим каждый тип карт позже.

Кнопки

В большинстве карт присутствуют кнопки. Кнопки используются для предоставления пользователям вариантов ответа. Пользователи могут отвечать на вопросы и запросы чат-бота, нажимая кнопки. Это альтернативный подход, позволяющий пользователям вводить свои запросы и ответы с использованием текста на естественном языке.

Плагины

Плагины – это части блоков, где чат-бот должен выполнять специальные задачи, а не просто отвечать на запросы пользователя. Например, вы можете использовать плагин для отправки электронной почты себе со всей информацией, полученной чат-ботом от пользователя. Мы рассмотрим несколько плагинов позже.

Атрибуты

Атрибуты – это переменные на языке программистов. Это места, в которых мы можем временно хранить данные во время разговора. Существуют предопределенные атрибуты и те, которые определены пользователем. Атрибуты используются для управления диалогом или применяются для сбора данных от пользователя для дальнейшей обработки в модуле работы с сервером.

Блоки по умолчанию

Предусмотрены два предопределенных блока: **Welcome message** (Приветственное сообщение) и **Default answer** (Ответ по умолчанию).

Welcome message (Приветственное сообщение) содержит карту с приветственным текстом по умолчанию. Блок приветствия запускается, когда пользователь впервые встречается с чат-ботом. В Facebook Messenger это срабатывает, когда пользователь нажимает кнопку **Get Started** (Начать). Блок ответа по умолчанию срабатывает, когда бот не знает, как обработать входное сообщение пользователя. Попробуйте поздороваться с ботом на Facebook Messenger и посмотрите, как он реагирует, – рис. 2.9.

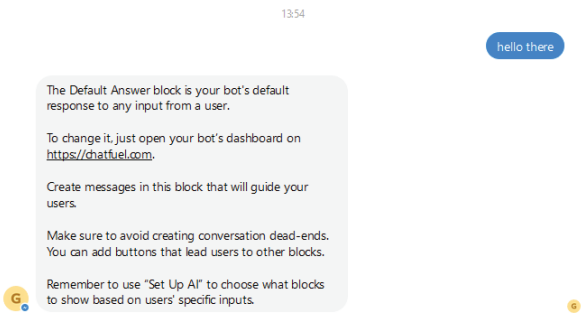


Рис. 2.9

Настройка бота

Теперь, когда мы разобрались в основах, начнем разработку чат-бота.

1. В блоке приветствия щелкните текст по умолчанию и отредактируйте его. Если навести на блок указатель мыши, вы увидите команды, позволяющие удалять, переупорядочивать и добавлять новые карты к существующим. Удалите кнопку **Main menu** (Главное меню) – рис. 2.10.

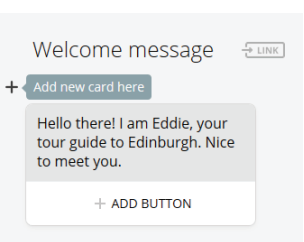


Рис. 2.10

2. Добавьте карту **Text** (Текст). Мы добавим следующую текстовую карту и зададим пользователю вопрос.
3. Добавьте кнопки для ответов пользователей. Щелкните мышью по ссылке **Add button** (Добавить кнопку) и введите название кнопки. Пока что игнорируйте названия блоков. Поскольку они неполны, то будут отмечены красным. Помните, что вы можете добавить не более трех кнопок на текстовую карту, – рис. 2.11.

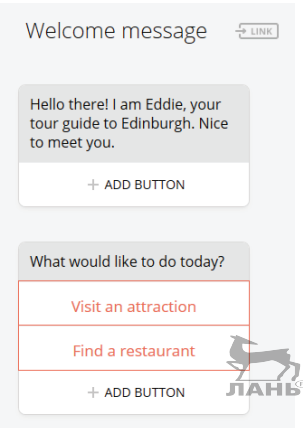


Рис. 2.11

4. Кнопки ответа должны быть привязаны к блокам, поэтому, когда пользователи нажимают кнопку, чат-бот будет знать, что делать или говорить. Мы добавим несколько блоков. Чтобы добавить новый блок, щелкните мышью по ссылке **Add block** (Добавить блок) на вкладке **Bot Structure** (Структура бота). Так вы создадите новый безымянный блок. В правой части введите имя блока. Повторите то же самое для каждого блока, который вы хотите создать, – рис. 2.12.

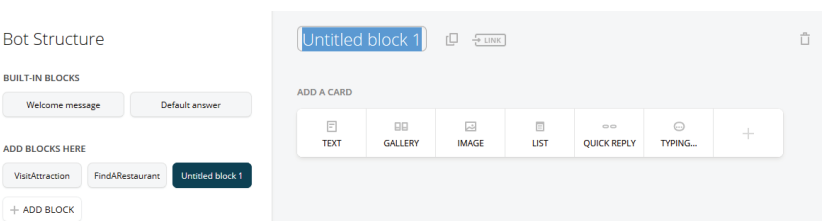


Рис. 2.12

5. Теперь вернитесь к кнопкам и укажите имена блоков для подключения. Нажмите соответствующую кнопку, выберите пункт **Blocks** (Блоки) и укажите название блока – рис. 2.13.

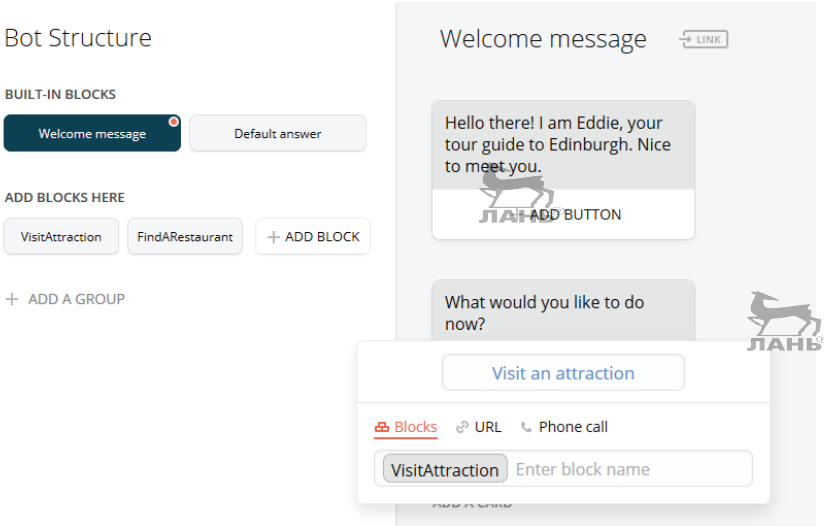


Рис. 2.13

6. Для каждого созданного вами блока добавьте контент, добавив соответствующие карты. Помните, что каждый блок может иметь более одной карты. Каждая карта будет отображаться в виде ответа, последовательно – рис. 2.14.

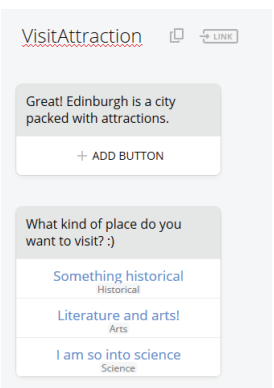


Рис. 2.14

Повторите предыдущие шаги, чтобы добавить больше блоков и подключить их к кнопкам других блоков. Когда вы закончите, можно проверить, нажав кнопку Test this chatbot (Протестировать этого чат-бота) в правом верхнем углу редактора. Теперь вы должны увидеть новое приветственное сообщение с кнопками ответов. Нажмите одну из них, чтобы поговорить, – рис. 2.15.

Прекрасно! Теперь у вас есть бот с потоком диалога.

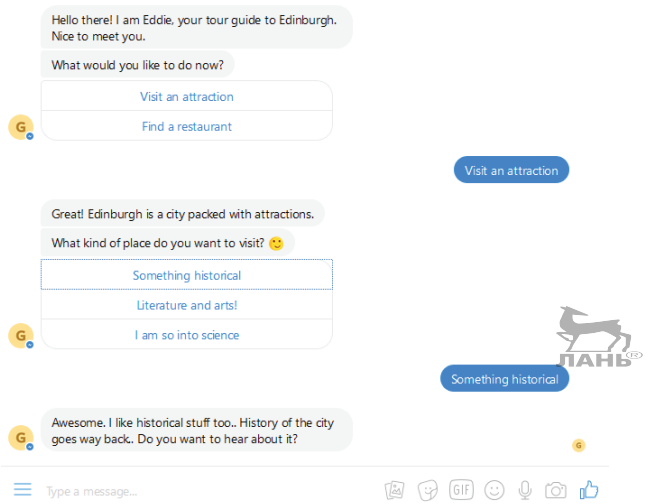


Рис. 2.15

Дополнительные карты

Помимо текстовых, есть другие типы карт, которые можно использовать для предоставления контента. Эти карты также относятся к платформе доставки. Только некоторые платформы, наапример Facebook Messenger, поддерживают расширенные карты, такие как карусели. Рассмотрим типы доступных карт.

Графика

Чтобы добавить карту изображения, щелкните мышью по значку **Image** (Изображение) в разделе **Add a card** (Добавить карту). Выберите пункт **Upload Image** (Загрузить изображение) и выберите файл изображения – рис. 2.16.

Рис. 2.16

Аудио

Вы можете отправлять аудиофайлы по чату, которые могут быть мгновенно воспроизведены пользователем. Чтобы отправить аудиофайл, щелкните мышью по значку + в разделе **Add a card** (Добавить карту). Выберите пункт **Audio** (Аудио) и укажите URL-адрес аудиофайла. Поддерживаются форматы MP3, WAV и OGG – рис. 2.17.

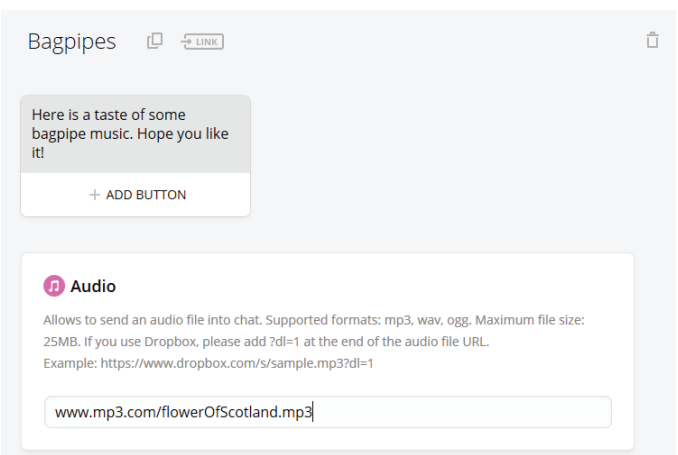


Рис. 2.17

Видео

Добавление видеокарты аналогично аудиокарте. Щелкните мышью по значку **+**, выберите пункт **Video** (Видео) и укажите URL-адрес видеофайла. Поддерживается только формат MP4, а максимальный размер видео ограничен до 25 Мб. Вот как видеоролик выглядит в чате.

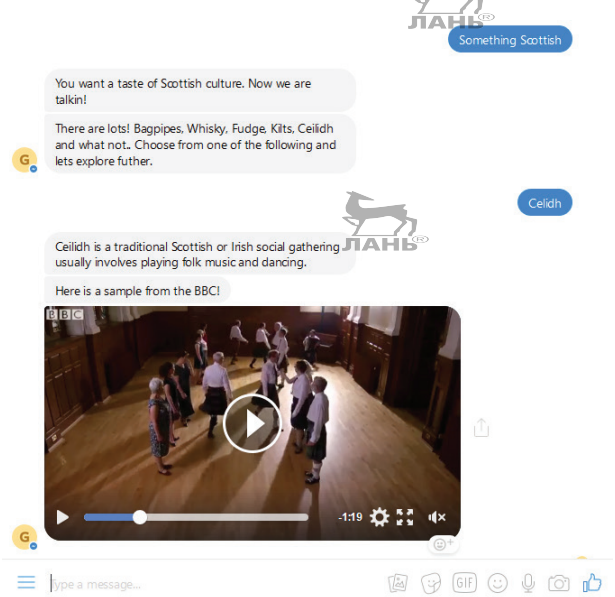


Рис. 2.18

Быстрые ответы

Быстрые ответы – это набор кнопок, отображаемых по горизонтали. Они могут использоваться для получения ответов от пользователя так же, как и кнопки, прикрепленные к текстовым картам. Разница в том, что так можно разместить до 10 кнопок, и они не прикреплены к карте. Кнопки исчезнут, как только пользователь нажмет одну из них или наберет свой запрос/ответ, вместо того чтобы нажать кнопку. Это чрезвычайно полезно, если доступно свыше трех вариантов ответа или параметров, которые не должны задерживаться, поскольку потеряют контекст.

Чтобы добавить массив быстрых ответов, выберите пункт **Quick Reply** (Быстрый ответ), добавьте названия кнопок. Для каждой

кнопки вы можете указать необходимый ей блок. Если отклики должны рассматриваться как ответы, они также могут храниться в виде пользовательских атрибутов. Далее мы рассмотрим эти атрибуты.

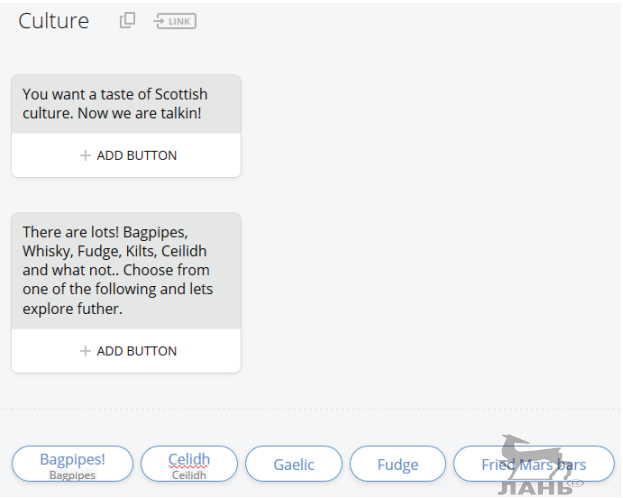


Рис. 2.19

Галерея

Галерея – это тип карты, которая позволяет создавать богатый способ представления контента. Вместо простого набора кнопок в качестве быстрых ответов один и тот же контент может быть представлен обогащенными изображениями и текстовыми описаниями. Чтобы добавить карту галереи, выберите пункт **Gallery** (Галерея) в разделе **Add a card** (Добавить карту). Карту галереи можно считать картой с несколькими мини-картами внутри.

Перейдите к добавлению изображения, заголовка, подзаголовка, дополнительного URL-адреса и кнопок (не более трех) для мини-карты по умолчанию. Закончив, добавьте больше мини-карт, щелкнув мышью по значку **+** справа, – рис. 2.20.

Списки

Карта списка очень похожа на карту галереи с точки зрения контента. Он предоставляет возможность доставить пользователю

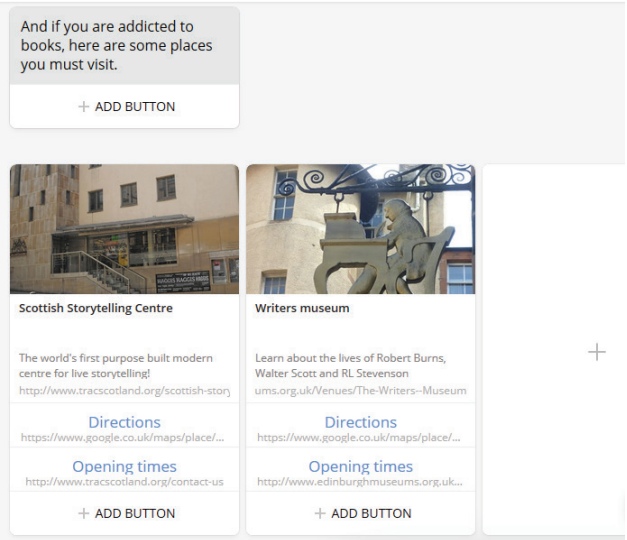


Рис. 2.20

богатый контент с изображениями и URL-адресами. Однако, в отличие от карты галереи, элементы в ней располагаются вертикально. Чтобы добавить список, щелкните мышью по значку **List** (Список) на вкладке **Add a card** (Добавить карту).

Добавьте изображение обложки, заголовок, URL, подзаголовок и кнопку побуждения к действию и создайте заголовок группы. Повторите то же самое для каждого элемента в списке. Фактически верхний элемент не обязательно должен быть заголовком группы, вместо этого может быть первым элементом в списке на фоне изображения – рис. 2.21.

В показанном примере у нас есть заголовок группы с названиями индийских ресторанов. Однако если вы не хотите использовать заголовок, то можете указать точку для первого элемента в списке. В списке карт допускается не менее двух и не более четырех наименований. Вы также можете добавить необязательную кнопку в конце списка, чтобы указать пункт «больше» для отображения большего количества элементов.

Навигация

Как пользователь и чат-бот могут перемещаться по цепочке общения? Как они реагируют друг на друга, и как продвигается разговор?

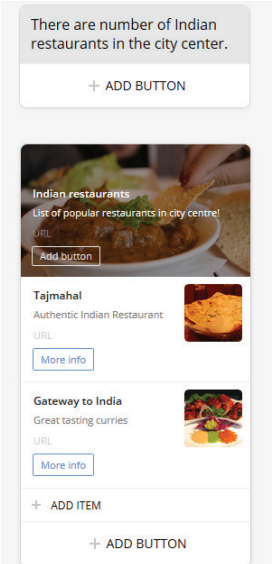


Рис. 2.21

В этом разделе мы рассмотрим инструменты для содействия потоку диалога.

Кнопки

Кнопки – это способ дать возможность пользователям отвечать чат-боту однозначно. Вы можете добавлять кнопки в текст, галерею и список карт. Кнопки имеют метку и полезную нагрузку. Метка – это то, что пользователь видит. Полезная нагрузка – это то, что происходит на стороне сервера, когда пользователь нажимает кнопку, – рис. 2.22.

Кнопка может иметь полезную нагрузку одного из четырех типов: следующий блок, URL-адрес, номер телефона или общий доступ. Следующий блок идентифицируется по имени блока. Он сообщает

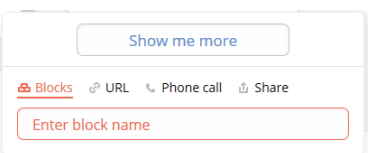


Рис. 2.22

чат-боту, какой блок выполнить при нажатии кнопки. URL-адрес можно указать, если чат-бот должен открыть веб-страницу во встроенном веб-браузере. Поскольку браузер встроенный, размер окна также можно указать. Можно указать номер телефона, если чат-бот должен сделать голосовой вызов (например, позвонить, чтобы зарезервировать столик). И наконец, общий доступ можно использовать в картах, таких как списки и галереи, чтобы поделиться картой с другими контактами пользователя.

Переход к блоку

Кнопки могут использоваться для навигации пользователя с одного блока на другой, однако пользователю необходимо нажать кнопку, чтобы включить навигацию. Но могут возникнуть обстоятельства, при которых навигация должна произойти автоматически. Например, если чат-бот дает пользователю пошаговые инструкции о том, как что-то сделать, его можно построить, поместив все карты (один шаг информации на карту) в один блок. Однако было бы неплохо поместить их в разные блоки для модульности. В таком случае нам нужно предоставить пользователю кнопку «*следующий шаг*», чтобы перейти к следующему шагу.

В Chatfuel мы можем использовать карту **Go to Block** (Переход к блоку) для решения этой проблемы. **Go to Block** (Переход к блоку) можно поместить в конец любого блока, чтобы перенаправить чат-бота к другому блоку. Как только чат-бот выполнит все карты в блоке, он автоматически перемещается в другой блок без вмешательства пользователя. Используя карты **Go to Block** (Переход к блоку), мы можем создать модульного чат-бота. Чтобы добавить карту **Go to Block** (Переход к блоку) в конце блока, выберите пункт **ADD A CARD** (Добавить карту), щелкните мышью по значку **+** и выберите карту **Go to Block** (Переход к блоку). Введите название блока для перенаправления – рис. 2.23.

Перенаправления могут быть случайными и условными. Выбирая случайную опцию, мы можем разрешить чат-боту случайно выбрать один из упомянутых блоков. Это добавляет некоторую неопределенность в разговор. Однако такой способ нужно использовать очень осторожно, потому что контекст разговора может оказаться сложным для его поддержания.

Условные перенаправления удобны, если необходимо проверить контекст до того, как будет выполнено перенаправление. Мы рассмотрим этот вариант после обсуждения контекста.

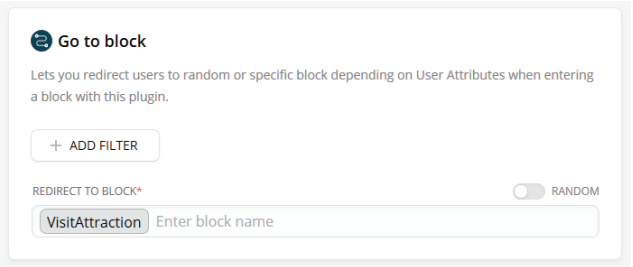


Рис. 2.23

Учет контекста

В любом диалоге необходимо управлять контекстом разговора. Контекст можно поддерживать, создавая локальный кеш, где может храниться информация, передаваемая между двумя партнерами диалога. Например, пользователь может сообщить чат-боту о своих предпочтениях в отношении пищи, и эта информация может быть сохранена в контексте для дальнейшего использования, если не используется немедленно. Другой пример – в диалоге, где пользователь задает исторические вопросы. Эти вопросы могут быть неполными и, возможно, должны толковаться с точки зрения информации, доступной в контексте.

В этом разделе мы рассмотрим, как контекст может быть записан и использован во время диалога в Chatfuel. Мы рассмотрим задачу поиска ресторана как часть вашего чат-бота путеводителя. Диалог между чат-ботом и пользователем может выглядеть следующим образом:

```
Пользователь: Поиск ресторана
Бот: Ok. Где?
Пользователь: Центр города.
Бот: Ok. Какая кухня вам нравится?
Пользователь: Индийская.
Бот: Ok. Давайте посмотрим. Я нашел пару индийских ресторанов в центре города. Вот они.
```

В показанном диалоге до последнего своего высказывания бот должен сохранять информацию локально. Когда он собрал всю необходимую информацию, он может отключиться и выполнить поиск в базе данных с соответствующими параметрами. Обратите

внимание, что он также должен использовать эту информацию для динамического генерирования высказываний. Рассмотрим, как реализовать обе эти задачи – динамически генерировать высказывания и искать в базе данных.

Во-первых, нам нужно выстроить поток диалога, чтобы пользователь мог вести беседу, как мы обсуждали в разделе *«Настройка бота»*. Предположим, что пользователь нажимает кнопку **Find_a_restaurant** в блоке приветствия. Мы построим базовые блоки с текстовыми сообщениями и кнопками для навигации по цепочке диалога.

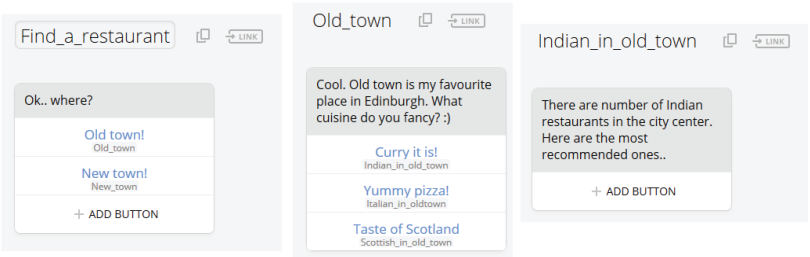


Рис. 2.24

Пользовательский ввод

Как вы можете себе представить, создание блоков для каждой кухни и комбинации мест может стать трудоемкой задачей. Мы построим ту же функциональность по-другому – с помощью форм. Чтобы использовать формы, необходимо использовать карту *ввода пользователя*. Мы создадим новый блок, **Restaurant_search**, и к нему добавим карту ввода пользователя. Чтобы добавить карту ввода пользователя, выберите пункт **Add a card** (Добавить карту), щелкните мышью по значку **+** и выберите карту **User Input** (Ввод пользователя).

Добавьте все вопросы, которые вы хотите задать пользователю, в поле **Message to user** (Сообщение для пользователя). Ответы на каждый из этих вопросов можно сохранить в переменных. Дайте имена переменным по каждому вопросу. Эти переменные всегда обозначаются двойными фигурными скобками (например, `{{restaurant_location}}`).

Информация, предоставленная пользователем, также может быть подтверждена до того, как будет принята. Если требуемая

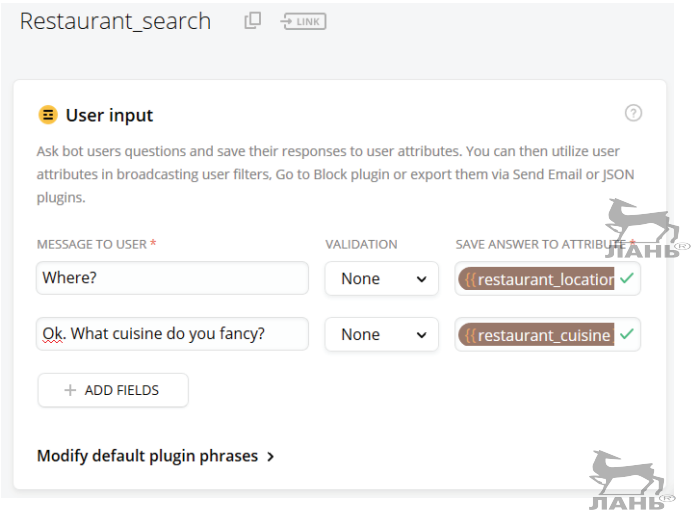


Рис. 2.25

информация – номер телефона, адрес электронной почты или номер, они могут быть подтверждены путем выбора соответствующего формата поступающей информации. После пользовательской карты ввода добавим карту **Go to Block** (Переход к блоку), чтобы перенаправить поток на страницу результатов (рис. 2.26).

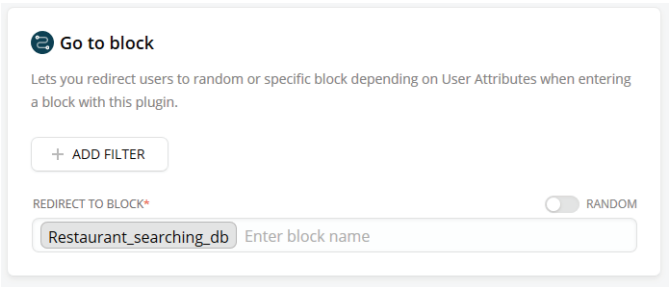


Рис. 2.26

И добавьте блок, в котором мы представим результаты. Как видно из рисунка, переменные, содержащие информацию, могут использоваться в высказываниях чат-бота. Они будут динамически заменены из контекста по мере разговора – рис. 2.27.

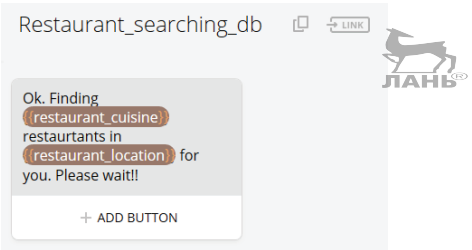


Рис. 2.27

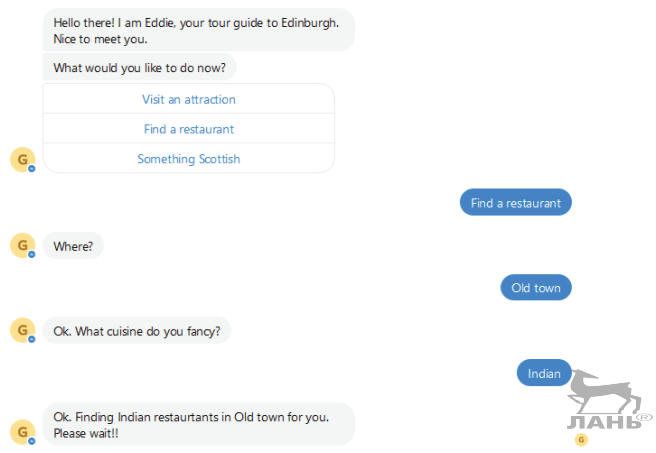


Рис. 2.28

Рисунок 2.28 демонстрирует диалог на платформе Messenger.

Настраиваемые пользовательские атрибуты

В дополнение к картам ввода пользователя существует еще один способ сохранить информацию в контексте. Это можно сделать, используя карту *настраиваемых пользовательских атрибутов*. Используя эту карту, вы можете установить контекстно-зависимую информацию в любой момент во время разговора. Посмотрим, как это сделать. Чтобы добавить эту карту, выберите пункт **Add a card** (Добавить карту), щелкните мышью по значку **+** и выберите карту **Set Up User Attribute** (Настраиваемые пользовательские атрибуты) – рис. 2.29.

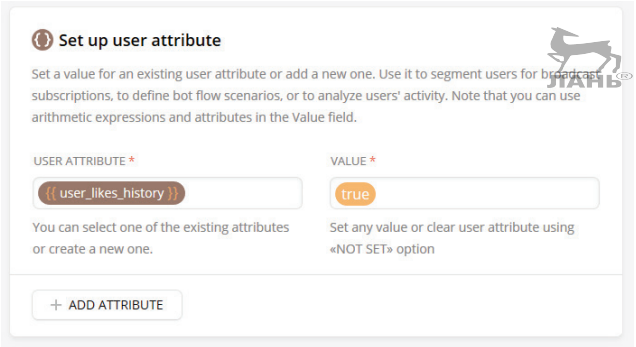


Рис. 2.29

На предыдущем рисунке показана переменная `user-likes-history`, которой присваивается значение `true`, когда пользователь запрашивает информацию об исторических достопримечательностях. Эта информация позже может использоваться для ведения разговора (как используется в карте **Go to Block** (Переход к блоку)) или для предоставления рекомендаций. Переменные, которые уже находятся в контексте, могут быть сброшены на новые значения или вообще не иметь значений. Чтобы очистить значение переменной, используйте специальное значение **Not set** (Не задано) в раскрывающемся меню, которое появляется, если вы попытаетесь заполнить значение для переменной. Кроме того, вы можете установить/перезагрузить сразу несколько переменных в карте.

Контекстные переменные по умолчанию

Помимо определения собственных контекстных переменных, вы также можете использовать список предопределенных переменных. Информация, содержащаяся в этих переменных, включает следующее:

- информация, полученная с платформы развертывания (то есть Facebook), включая имя пользователя, пол, часовой пояс и локаль;
- контекстная информация – последняя нажатая кнопка, название последнего посещенного блока и т. д.

Чтобы получить полный список переменных, создайте новую текстовую карту и введите {{. Вы увидите раскрывающееся меню со списком переменных, которые можете выбрать. Этот список также будет содержать созданные вами переменные – рис. 2.30.

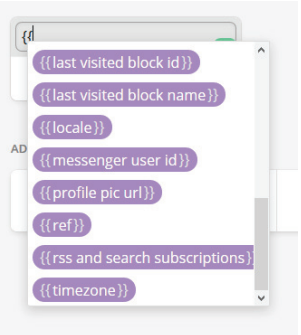


Рис. 2.30

Как и в случае с определенными разработчиком переменными, эти встроенные переменные также могут использоваться в текстовых сообщениях и в условных перенаправлениях с использованием карт **Go to Block** (Переход к блоку).

Распознавание естественного языка

До сих пор мы видели, как происходит поток диалога с помощью кнопок. В каждом случае пользователь должен нажать кнопку или ввести информацию, чтобы заполнить поле, что дает возможность разговору продолжаться. Тем не менее пользователи могут инициировать беседу и использовать навигацию с использованием **естественного языка (natural language – NL)**, а также в Chatfuel. В этом разделе мы рассмотрим, как научить чат-бота распознавать естественные языковые вводы пользователя и выполнять необходимые действия.

Блок по умолчанию

Наряду с блоком **Welcome message** (Приветственное сообщение) есть блок **Default answer** (Ответ по умолчанию). Этот блок используется

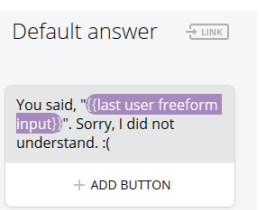


Рис. 2.31

чат-ботом для ответа на любой ввод пользователя, который он не распознает. Измените содержимое блока следующим образом:

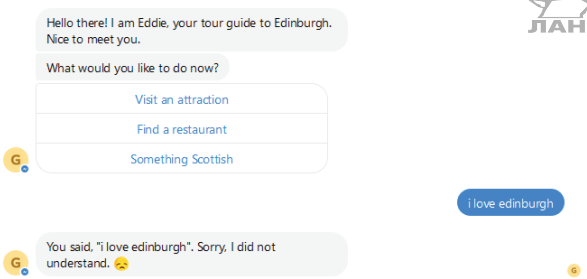


Рис. 2.32

Теперь проверьте работу бота, введя вопрос в окне Messenger. Вы увидите, как выводится сообщение по умолчанию.

Теперь рассмотрим, как обрабатывать вход на естественном языке, чтобы пользователи не получали этот ответ по умолчанию. Чтобы настроить чат-бота на распознавание ввода на естественном языке, щелкните мышью по ссылке **Set Up AI** (Настройка ИИ) в меню в крайнем левом углу страницы. Здесь мы укажем шаблоны

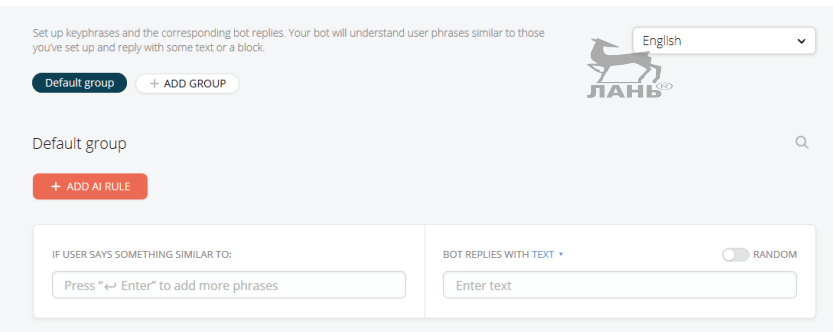


Рис. 2.33

ввода на естественном языке и соответствующие ответы на них чат-бота.

Щелкните мышью по ссылке **ADD AI RULE** (Добавить правило ИИ). Вы увидите два поля. Введите пользовательские высказывания и ответы бота в соответствующих полях рис. 2.34:

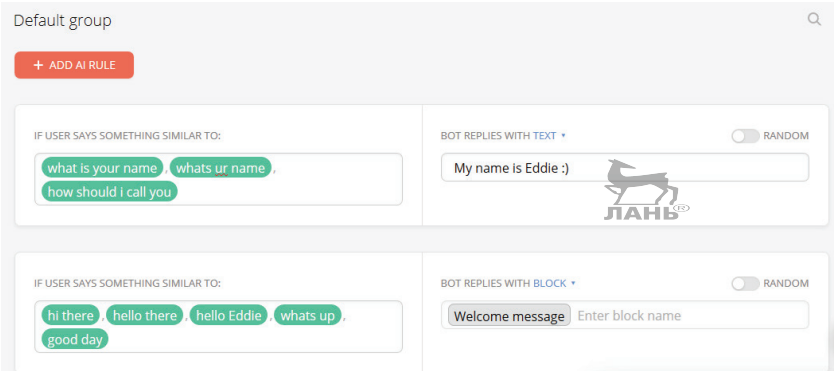


Рис. 2.34

Ответы могут быть текстовыми или блочными. Предоставляя имена блоков, чат-бот может быть перенаправлен на соответствующие блоки, когда пользователь вводит ответы, а не нажимает кнопки. Добавив несколько правил, вернитесь в Messenger и протестируйте бота, вводя текстовые запросы на естественном языке, – рис. 2.35.

Вы также можете добавить элемент неопределенности и случайности, выбрав флаг случайности в настройках ответа бота и добавив несколько ответов. Бот выберет один ответ случайным образом.

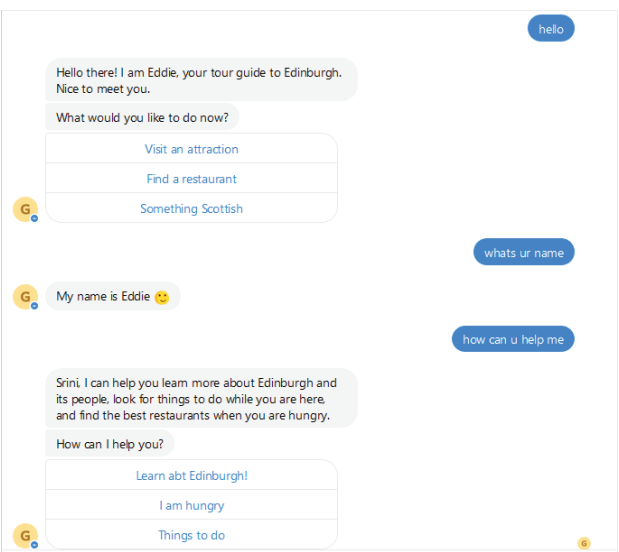


Рис. 2.35

Обработка на стороне сервера

Задачи ведения диалога, как правило, должны быть подкреплены задачами на стороне сервера. Например, бронирование стола в ресторане – это не просто разговор, но также и действие по бронированию стола. Это серверная задача, когда информация о бронировании отправляется на сервер бронирования в качестве HTTP-запроса. В этом разделе мы рассмотрим, как использовать карту JSON API для включения серверных задач.

Прежде чем мы начнем использовать карту API JSON, нам нужен URL-адрес, который может принимать несколько параметров и выполнять серверную задачу. Мы создадим фиктивную серверную службу, которая может заказать стол в определенном ресторане для определенного количества людей и на указанное время. Для этого нам нужно создать веб-приложение Node.js и разместить его в облаке.

1. Создайте каталог с именем *Eddie-bot*. В этом каталоге нам нужны три файла: *index.js*, *package.json* и *Procfile*.
2. Создайте файл пакета *package.json*, который объявляет пакеты, необходимые для приложения, как показано ниже.

```
{
  "name": "eddie-server",
  "version": "1.0.0",
  "description": "Eddie - Chatfuel",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
},
  "author": "Srini Janarthanam",
  "license": "ISC",
  "dependencies": {
    "body-parser": "^1.15.2",
    "express": "^4.14.0",
    "request": "^2.72.0"
  }
}
```

3. Создайте файл *Procfile*. *Procfile* – специальный файл, который используется нашей облачной службой, называемой Heroku, чтобы сообщить оператору о том, какая команда нужна, чтобы запустить приложение. Он должен содержать следующий код:

```
web: node index.js
```

4. Создайте файл *index.js*. Нам нужна программа, которая будет обрабатывать запрос на сервере. Вставьте следующий код:

```
// Веб-настройки EddieBot

const express = require('express')
const bodyParser = require('body-parser')
const app = express()
app.set('port', (process.env.PORT || 5000))

// Выполнение application/x-www-form-urlencoded
app.use(bodyParser.urlencoded({extended: false}))

// Process application/json Выполнение application/json
app.use(bodyParser.json())
app.use(express.static('public'))

// Контакт с сервером
app.listen(app.get('port'), function() {
    console.log('running on port', app.get('port'))
})

// Индексный маршрут
app.get('/', function (req, res) {
    res.send('Hello world, I am EddieBot webhook.')
})

app.post('/booktable/', function (req, res) {
    console.log(JSON.stringify(req.body));
    // ВАШЕ БРОНИРОВАНИЕ НАЧИНАЕТСЯ ЗДЕСЬ!!

    var out = {
            "messages": [
                {"text": "Thanks for your booking!"},
                {"text": "See you soon!!"}
            ]
    }

    var outString = JSON.stringify(out);
    console.log('Out:' + outString);

    res.send(outString);
})
```

В предыдущей программе у нас есть дескриптор **booktable**, который мы будем вызывать, если пользователь хочет заказать столик в определенном ресторане. В этот дескриптор мы отправим ряд параметров, необходимых для бронирования столика, например название ресторана, количество людей и время. После того как заказ сделан, приложение возвращает сообщение об успешном выполнении боту, которое затем перенаправляется пользователю. В предыдущем коде мы фактически не делаем бронирования. Мы просто предполагаем, что бронирование сделано.

5. Чтобы это приложение работало, нам нужно выложить его на облачный сервер. Для этого в Heroku выполните следующие команды в консоли:

```
C:\Eddie-bot> git init
C:\Eddie-bot> git add .
C:\Eddie-bot> git commit -m “Backend Eddie Bot v1”
C:\Eddie-bot> heroku create eddie-bot-backend
C:\Eddie-bot> git push heroku master
```

Последняя команда выкладывает приложение в облако, собирает его и выполняет сценарий index.js, который запускает веб-сервер. Вы узнаете больше о том, как создавать веб-приложения в Node.js и Java в последующих главах. Теперь мы можем вызвать URL-адрес **eddie-bot-backend.herokuapp.com/booktable** с необходимыми параметрами, чтобы сделать заказ.

6. Теперь, когда создан фейковый сервер бронирования, обратимся к нему и сделаем заказ из чат-бота. Это предполагает создание потока с использованием карт нескольких типов. Предположим, что пользователь нажимает кнопку **More information** (Дополнительная информация) для определенного ресторана, например Tajmahal. При этом необходимо, чтобы пользователь зашел в блок, где мы показываем информацию о ресторане, такую как кухня, меню и местоположение, с возможностью заказать стол, если пользователь заинтересован. Итак, построим базовый блок для ресторана Tajmahal – рис. 2.36.
7. Нам нужно добавить еще пару карт в этот блок. Чтобы установить контекст, нам нужно добавить карту пользовательских атрибутов, где мы задаем название ресторана, – рис. 2.37.

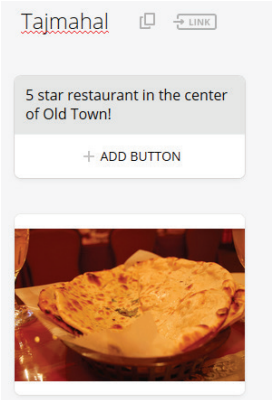


Рис. 2.36

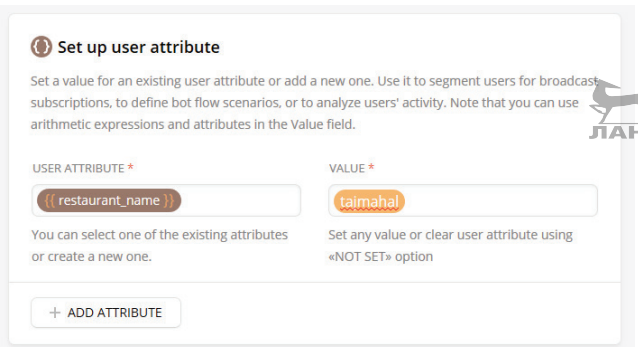


Рис. 2.37

8. Затем мы переводим диалог в общий блок бронирования столиков, где может собираться другая необходимая информация. Для этого создадим новый блок **Table_booking_form** – рис. 2.38. В этом блоке мы добавили карту ввода пользователя, в которой у пользователя запрашивается информация о количестве людей и времени. Данные сохраняются в соответствующих переменных.
9. Наконец, к этому блоку мы добавляем карту **JSON API**, которая инструктирует чат-бота обратиться к внешнему серверу для обработки, – рис. 2.39.
   В этой карте мы должны указать тип запроса (**GET** или **POST**), URL-адрес для вызова и параметры для отправки. Ответ от сервера возвращается пользователю.

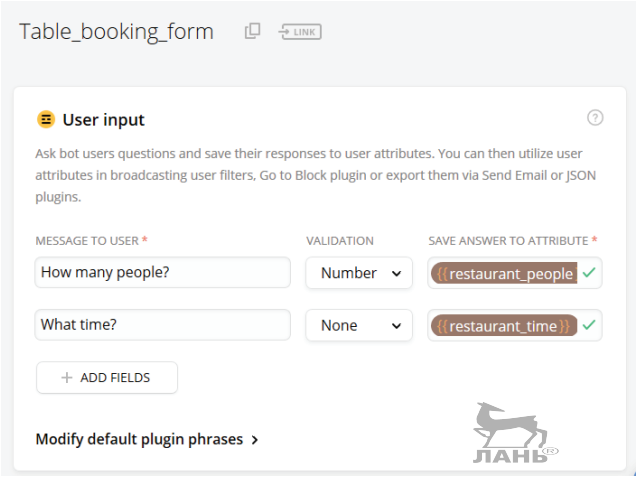


Рис. 2.38

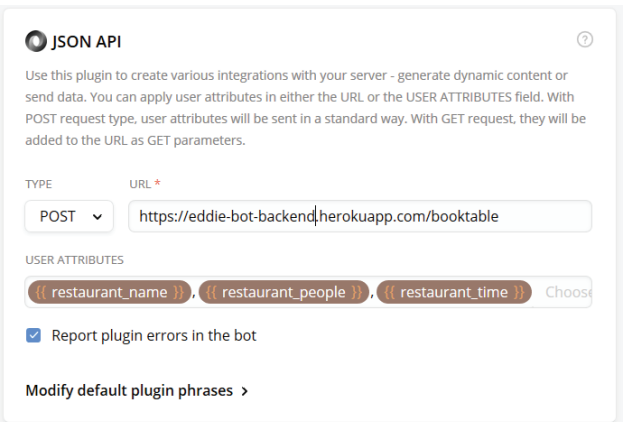


Рис. 2.39

Отлично! Проверим работу в Messenger – рис. 2.40!

Обратите внимание, как текстовое сообщение, возвращаемое с сервера, обрабатывается в Messenger. Ответ от сервера должен быть в определенном формате. В файле *index.js* мы отправили обратно простое текстовое сообщение. Однако могут быть переданы и другие типы сообщений. Например, ниже приведен пример того, как вы можете отправить текстовый блок с помощью кнопок.

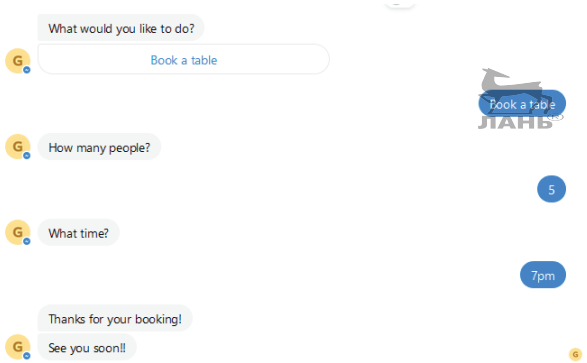


Рис. 2.40

```
{
  "messages": [
    {
      "attachment": {
        "type": "template",
        "payload": {
          "template_type": "button",
          "text": "Here are my recommendations!",
          "buttons": [
            {
              "type": "show_block",
              "block_name": "TajMahal",
              "title": "TajMahal (5 stars)"
            },
            {
              "type": "show_block",
              "block_name": "SpicyTandoori",
              "title": "Spicy Tandoori (4 stars)"
            }
          ]
        }
      }
    }
  ]
}
```

Изучите другие типы сообщений, которые вы можете отправить, для создания более динамических диалоговых карт по адресу **help.chatfuel.com/facebook-messenger/plugins/json-plugin/**.

Широковещание

Еще одна интересная функция, предоставляемая Chatfuel, – это возможность отправлять сообщения нескольким пользователям в любое время. Это средство, которое вы можете использовать для перезапуска сеансов с пользователем или отправки уведомлений периодически. Чтобы использовать эту функцию, щелкните мышью по ссылке **Broadcast** (Широковещание) в главном меню – рис. 2.41.

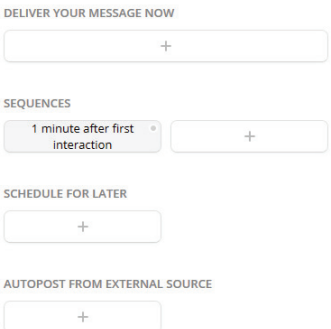


Рис. 2.41

Вы увидите четыре варианта. Мы выберем пункт **Deliver your message now** (Доставить сообщение сейчас). Есть три вещи, которые нужно сделать: составить сообщение для отправки, выбрать пользователей и нажать кнопку **Send** (Отправить) – рис. 2.42.

Выберите пункт **User filter** (Отфильтровать пользователей), чтобы выбрать подмножество пользователей, которым вы хотите отправить сообщение. Это делается с использованием переменных и значений. Например, вы можете отправить сообщение всем пользователям, которые сказали, что им нравится индийская кухня (то есть `{{restaurant_cuisine}} = 'Indian'`) и что они находятся в центре города (`{{restaurant_location}} = ' Old town'`). Это сообщение может информировать об акциях в индийском ресторане в городе – рис. 2.43.

Выбрав целевую группу пользователей, создайте блок сообщения и щелкните мышью по ссылке **Send** (Отправить). Это сообщение будет немедленно отправлено всем подходящим пользователям.

Вы также можете отправлять сообщения целевым группам пользователей по расписанию, последовательно создавать автоматические сообщения после взаимодействия с пользователем и отправлять

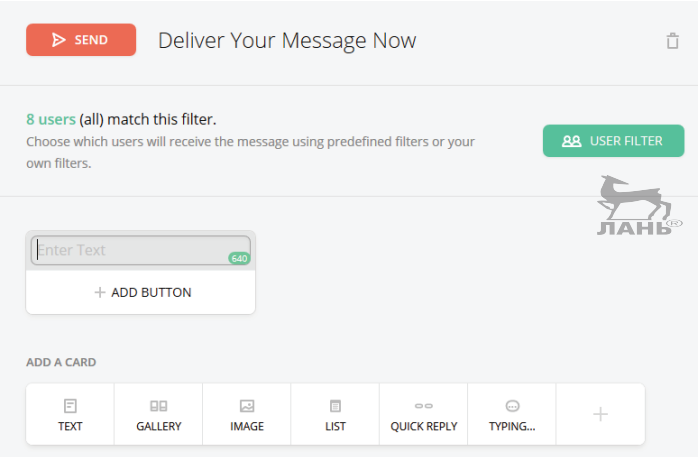


Рис. 2.42

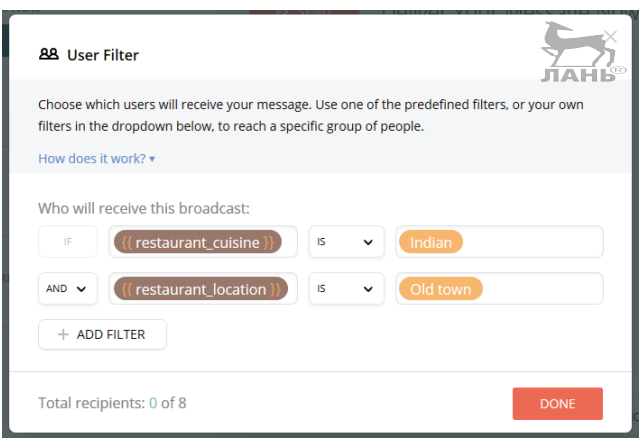


Рис. 2.43

сообщения из внешних источников, а также использовать функцию широковещания.

Шаблоны

Чаты могут быть созданы из шаблонов. Это простые предварительно разработанные чат-боты, которые легко отредактировать и преобразовать в чат-бота, который вам нужен. Для этого есть несколько

шаблонов. Перейдите на страницу **dashboard.chatfuel.com/#/bots**. В разделе **Tutorial templates** (Демонстрационные шаблоны) щелкните мышью по ссылке **View All Templates** (Просмотреть все шаблоны) – рис. 2.44.

Chatfuel предоставляет ряд шаблонов со множеством вариантов использования, таких как рестораны, ассистент звезды, викторина и электронная коммерция. Изучите их, чтобы узнать, как структурирована беседа и как для поддержки этих диалогов использовались различные инструменты.

Аналитика

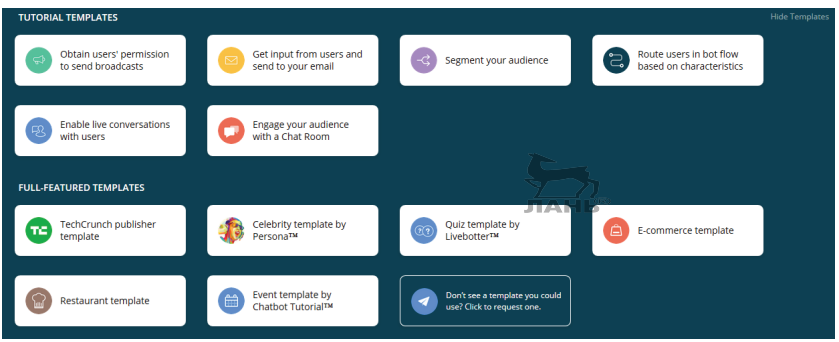


Рис. 2.44

Chatfuel также предоставляет инструменты аналитики в вашем чат-боте. Чтобы получить доступ к аналитике, перейдите на панель мониторинга чата и выберите пункт Analyze (Аналитика) в главном меню. Страница аналитики суммирует статистику использования бота с точки зрения роста количества пользователей, активности пользователей и удержания пользователей. Также освещаются шаблоны использования, такие как блоки, которые популярны у пользователей, и проблемы, с которыми сталкивается ИИ, с точки зрения распознавания высказываний пользователей. Наконец, также сообщается, где в мире находятся ваши пользователи.

Заключение

Поздравляю! В этой главе мы отправились в путь для создания потрясающих чат-ботов. Надеюсь, при помощи Chatfuel у вас

получилось создать первого бота. Используя путеводитель в качестве варианта использования, мы заодно изучили различные темы конструирования и развития чат-бота: конструкции потока сообщений, блоки, типы содержимого сообщений, перемещение по диалогу, распознавание основных высказываний на естественном языке и многое другое. Мы также изучили различные строительные блоки, предоставляемые Chatfuel для создания, мониторинга и управления чат-ботами.

В следующих главах мы рассмотрим множество других вариантов использования и более сложные темы при создании чат-ботов, такие как распознавание нюансов в высказываниях на естественном языке и управление более сложными диалоговыми контекстами.

Ссылки

- Документация Chatfuel: **docs.chatfuel.com**.

Глава 3. Поговорим о погоде

Facebook Messenger – одна из самых популярных общедоступных платформ обмена сообщениями. Вы можете получить к ней доступ, используя свою учетную запись Facebook. Messenger доступен отдельно для Facebook в виде веб-сайта (**messenger.com** и **m.me**) и в качестве приложения для мобильных устройств. Благодаря этому платформа чрезвычайно доступна. В сети Facebook свыше 1 / миллиарда пользователей по всему миру, и все эти пользователи социальных сетей имеют доступ к Messenger; это ошеломляющее число растет с каждым днем. Поэтому Facebook Messenger – популярный выбор в качестве платформы развертывания среди разработчиков чат-ботов. В дополнение к количеству пользователей компания вкладывает огромные средства в развитие чат-ботов. Последние функции, такие как вкладка **Discover** (Интересное) и обновление SDK, служат свидетельством приверженности к этому подходу.

В этой главе мы разработаем и внедрим чат-бота, с которым можно говорить о погоде. Мы создадим чат-бота и интегрируем в него функционал серверной службы, которая будет предоставлять нам информацию о погоде в режиме реального времени. Затем мы перейдем к интеграции чат-бота в Facebook Messenger. И наконец, посмотрим на эксклюзивные функции Messenger, которые можно использовать, чтобы сделать чат-бота более привлекательным и контактным.

В ходе этой главы вы:

- спроектируете диалоги, чтобы поговорить о погоде;
- интегрируете серверную службу API OpenWeatherMap;
- создадите чат-бота на языке Java 8 и платформе Node.js и опубликуете его в облаке с помощью Heroku;
- узнаете, что такое Facebook Messenger SDK и каковы типы шаблонов сообщений;
- интегрируете чат-бота в Messenger.

Задачи чат-бота

Сначала определим задачи, которые должен выполнять чат-бот. Я полагаю, что чат-бот, настроенный поговорить о погоде, должен уметь сообщать следующее:

- погода за окном;
- прогноз на сегодня;
- прогноз на неделю;
- прогноз на выходные;
- долгосрочный прогноз.

Все упомянутые отчеты основаны на географическом местоположении. Эта информация может быть предоставлена в разных форматах. Большинство людей укажут название города или ближайшего районного центра либо почтовый индекс. Но, учитывая тот факт, что мобильные устройства обладают возможностью геолокации, а мессенджеры допускают общий доступ к местоположению, другой способ определения положения пользователя – это использовать географические координаты (широту и долготу). Мы рассмотрим все эти варианты. В дополнение к этим задачам и типам вводной информации нам нужно разобраться, как чат-бот отреагирует на запросы, если информация о местоположении неоднозначна.

Конструирование диалога

Теперь, когда мы определили задачи, которые должен выполнить наш чат-бот, создадим поток диалога. Поток диалога – это основная схема взаимодействия между пользователем и ботом.

Представим несколько простых воображаемых диалогов между пользователем и нашим ботом погоды.

Пример 1:

Пользователь: Привет.

Бот: Привет! Я - WeatherMan, погодный робот. Что ты хочешь узнать? Погоду за окном или прогноз?

Пользователь: Погоду за окном.

Бот: Ок. Для какого города?

Пользователь: Лондон, Англия.

Бот: Ок. Текущая погода в городе Лондон, Англия. Температура 10 градусов Цельсия. Ясно.

Пользователь: Спасибо.

Бот: Пожалуйста! :)

Представим еще один, немного сложнее.

Пример 2:

Пользователь: Привет.

Бот: Привет! Я – WeatherMan, погодный робот. Что ты хочешь узнать? Погоду за окном или прогноз?

Пользователь: Прогноз.

Бот: Ок. На завтра или на выходные?

Пользователь: На завтра.

Бот: Ок. Для какого города?

Пользователь: Эдинбург.

Бот: Эдинбург, США или Эдинбург, Англия?

Пользователь: Эдинбург, Англия.

Бот: Ок. Прогноз на завтра в городе Эдинбург, Англия. Температура 12 градусов Цельсия. Облачно.

Ладно, при диалогах о погоде возможны и более сложные варианты. Однако мы будем работать с предыдущими примерами, чтобы показать, как вы можете создать чат-бота и развернуть его на Facebook Messenger в принципе. Рассмотрим более сложные высказывания и беседы позже.

Задачи на стороне сервера

Прежде чем перейти к реализации чат-бота, взглянем на источник информации для данных о погоде, OpenWeatherMap (**openweathermap.org**). Это облачная служба, предоставляющая информацию о погоде примерно для 200 тыс. городов по всему миру. Она содержит как сведения о текущей погоде, так и прогнозы, а также исторические данные. Доступны бесплатные и платные подписки. Перейдите на сайт, введите название своего города и узнайте погоду.

Получение данных о погоде

Создадим учетную запись в службе OpenWeatherMap для получения данных о погоде.

Получение ключа API

В отличие от предыдущей главы, где мы создавали контент для чат-бота вручную, мы будем использовать данные, полученные в реальном времени с сервера службы. Чтобы это сделать, нам нужно

оформить подписку. Чтобы зарегистрироваться, перейдите на страницу **home.openweathermap.org** и создайте новую учетную запись.

Чтобы начать работу с данными, нам нужно будет подписаться на соответствующую службу API. Для этого перейдите по ссылке **API** в строке меню на главной странице веб-сайта.

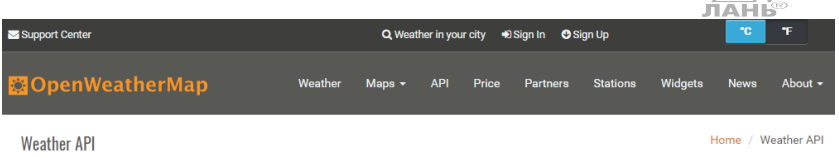


Рис. 3.1

На открывшейся странице вы увидите список всех доступных API. Начнем с самого первого интерфейса – **Current weather data** (Информация о текущей погоде). Щелкните мышью по ссылке **Subscribe** (Подписаться).

Price

Home / Price

Subscribe to current weather, forecasts, and historical data collections and enjoy our fast simple API!

Current weather and forecasts collection

	Free	Startup	Developer	Professional	Enterprise
Price Price is fixed, no other hidden costs.	Free	40 USD / month	180 USD / month	470 USD / month	2,000 USD / month
Subscribe	Get API key and Start	Subscribe	Subscribe	Subscribe	Subscribe
Calls per minute (no more than)	60	600	3,000	30,000	200,000
Current weather API	✓	✓	✓	✓	✓
5 days/3 hour forecast API	✓	✓	✓	✓	✓
16 days/daily forecast API	-	-	✓	✓	✓
Weather maps API	✓	✓	✓	✓	✓

Рис. 3.2

Щелкните мышью по ссылке **Get API key and Start** (Получить API-ключ и начать работу). Откроется страница идентификатора приложения (**openweathermap.org/appid**), где вы найдете документацию по API (то есть как использовать ключ и ограничивать к нему доступ).

Чтобы получить свой ключ, перейдите на страницу **API keys**, **home.openweathermap.org/api_keys**. Вы увидите ключ по умолчанию. Если он не указан, создайте его.

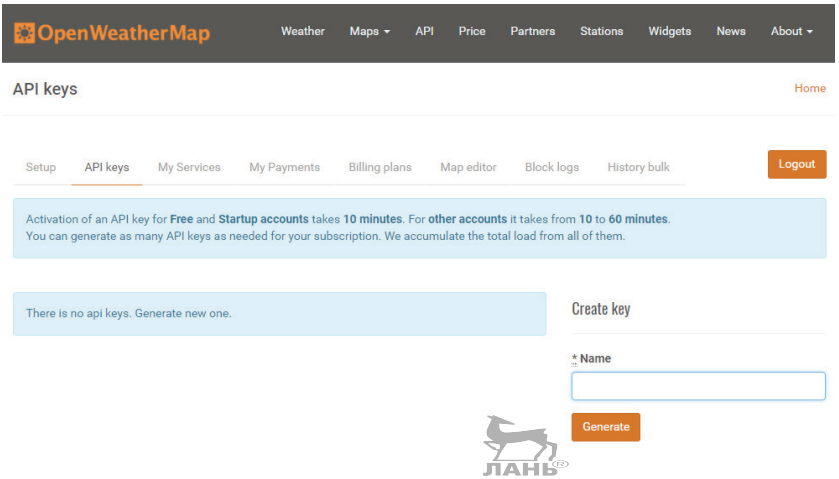


Рис. 3.3

Теперь, заимев ключ, испробуем его в деле!

Проверка вашего ключа

Мы должны сконструировать URL-адрес, чтобы получить нужные нам данные. Ниже показан синтаксис:

```
http://api.openweathermap.org/data/2.5/forecast?id=КОД_ГОРОДА&APPID=ВАШ_API_КЛЮЧ
```

Чтобы получить код города, перейдите по URL-адресу **openweathermap.org/find?q=**, введите название своего города и щелкните мышью по кнопке **Search** (Поиск). На странице результатов выберите город, а затем скопируйте код города из URL-адреса – рис. 3.4.

Например, код города для английского города Лондон – **2643743**.

Замените значение `КОД_ГОРОДА` полученным кодом города, а значение `ВАШ_API_КЛЮЧ` – своим ключом и введите полученный URL-адрес в веб-браузере. Вы получите объект JSON с текущей и прогнозируемой информацией о погоде, например так (рис. 3.5.):

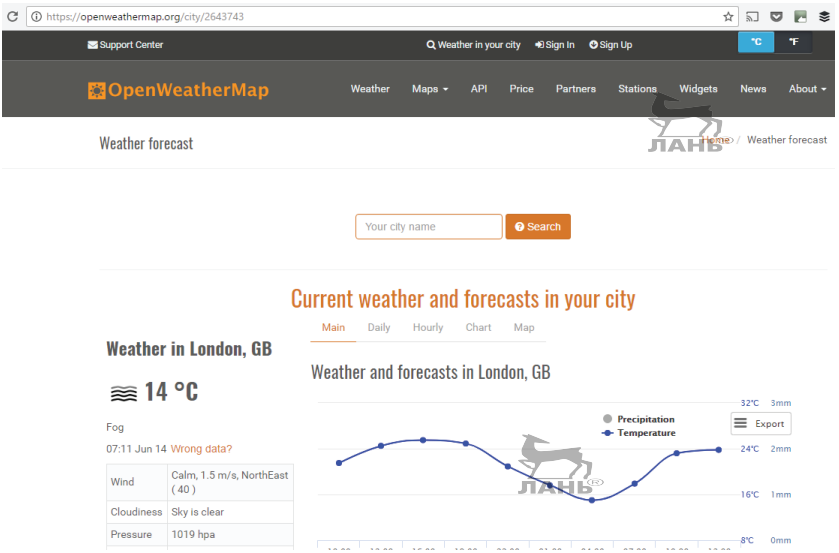


Рис. 3.4


```
{
    "cod": "200",
    "message": 0.003,
    "cnt": 37,
    "list": [ … ], // 37 items
    "city": {
        "id": 2643743,
        "name": "London",
        "coord": {
            "lat": 51.5085,
            "lon": -0.1258
        },
        "country": "GB"
    }
}
```


Рис. 3.5

Тег `list` с 37 элементами может быть развернут; в нем вы найдете текущую и прогнозируемую информацию о погоде на каждые три часа, начиная с текущего момента.

```
▼ "list": [
  ▼ {
      "dt": 1497430800,
    ▶ "main": { … }, // 8 items
    ▼ "weather": [
      ▼ {
          "id": 800,
          "main": "Clear",
          "description": "clear sky",
          "icon": "01d"
        }
      ],
    ▶ "clouds": { … }, // 1 item
    ▶ "wind": { … }, // 2 items
    ▶ "sys": { … }, // 1 item
      "dt_txt": "2017-06-14 09:00:00"
    },
  ▶ { … }, // 7 items
  ▶ { … }, // 7 items
```


Рис. 3.6

Превосходно! Теперь у нас есть источник данных для подключения к нашему чат-боту.

Построение интерфейса на стороне сервера

Начнем кодировать! Теперь мы напишем модуль на языке Java, который сможет получать данные о погоде из нашего источника данных в реальном времени. Затем мы подключим этот модуль к чат-боту, чтобы менеджер диалогов мог его использовать.

1. Откройте среду разработки Eclipse (**www.eclipse.org**).
2. Создайте новый проект Maven.
3. Выберите вариант **Create a simple project** (Создать простой проект).
4. Укажите расположение проекта и щелкните мышью по кнопке **Next** (Далее).
5. На следующей странице заполните поля **Group Id** (Идентификатор группы), **Artifact Id** (Идентификатор артефакта), **Name** (Имя) и **Description** (Описание). Нажмите кнопку **Finish** (Готово).

Теперь у нас готовый проект Maven. Прежде чем перейти к верстке серверного кода, добавим несколько пакетов, от которых зависит наш проект. Найдите POM-файл (*pom.xml*) и добавьте следующие зависимости Maven:

```
<dependencies>
<!-- https://mvnrepository.com/artifact/com.google.code.gson/gson -->
<dependency>
    <groupId>com.google.code.gson</groupId>
    <artifactId>gson</artifactId>
    <version>2.8.0</version>
</dependency>
<!-- https://mvnrepository.com/artifact/org.apache.httpcomponents/httpclient -->
<dependency>
    <groupId>org.apache.httpcomponents</groupId>
    <artifactId>httpclient</artifactId>
    <version>4.5.3</version>
</dependency>
</dependencies>
```

Возможно, нам понадобятся другие зависимости. Но для серверного кода этих пакетов достаточно. Теперь создадим класс Java `Weather.java` для доступа к данным о погоде. Показанный ниже код демонстрирует базовую структуру класса:

```
package weatherman.weather;

import com.google.gson.JsonObject;

public class Weather {
  public static void main(String[] args) {
  }

  public Weather(){}

  public JsonObject getWeather(String cityCode){ return null; }

  public JsonObject getCurrentWeather(String cityName){ return null; }

}
```

Теперь мы добавим необходимый код для получения фактических данных от службы OpenWeatherMaps.

```
package weatherman.weather;

import java.io.IOException;

import org.apache.http.HttpResponse;
import org.apache.http.client.ClientProtocolException;
import org.apache.http.client.HttpClient;
import org.apache.http.client.methods.HttpGet;
import org.apache.http.impl.client.HttpClientBuilder;
import org.apache.http.util.EntityUtils;

import com.google.gson.JsonObject;
import com.google.gson.JsonParser;

public class Weather {

  // Сюда поместите ваш ключ API
  String apiKey = <Ваш ключ API>;

  public static void main(String[] args) {
    try {
       (new Weather()).getWeather("2643743");
    } catch (ClientProtocolException e) {
        e.printStackTrace();
    } catch (IOException e) {
        e.printStackTrace();
    }
  }

  public Weather(){}

  public JsonObject getWeather(String cityCode)
               throws ClientProtocolException, IOException {

    // Шаг 1: Подготовка URL
    String url = "http://api.openweathermap.org/data/2.5/forecast?id="
      + cityCode + "&APPID=" + apiKey ;

    // Шаг 2: Создание HTTP-клиента
    HttpClient httpclient = HttpClientBuilder.create().build();

    // Шаг 3: Создание объекта HTTPGet и обращение к URL
    HttpGet httpGet = new HttpGet(url);
    HttpResponse response = httpclient.execute(httpGet);
```

```
    // Шаг 4: Обработка результата
    JsonObject json = null;
    int statusCode = response.getStatusLine().getStatusCode();
    if (statusCode == 200) {
      String response_string = EntityUtils.toString(response.getEntity());
      json = (new JsonParser()).parse(response_string).getAsJsonObject();
      Gson gson = new GsonBuilder().setPrettyPrinting().create();
      String prettyJson = gson.toJson(json);
      System.out.println(prettyJson);
    }
    return json;
  }

}
```

В методе `main` мы создали объект `Weather` для получения текущей погоды для Лондона, Англия (`2643743`). Запустите предыдущий код, и вы получите следующий результат в консоли:

```
Problems | @ Javadoc | Declaration | Search | Console | Progress
<terminated> Weather [Java Application] C:\Program Files\Java\jre1.8.0_66\bin\javaw.exe (7 Dec 2017, 10:32:24)
{
  "cod": "200",
  "message": 0.0409,
  "cnt": 40,
  "list": [
    {
      "dt": 1512648000,
      "main": {
        "temp": 284.77,
        "temp_min": 284.77,
        "temp_max": 285.379,
        "pressure": 1010.65,
        "sea_level": 1018.28,
        "grnd_level": 1010.65,
        "humidity": 99,
        "temp_kf": -0.61
      },
      "weather": [
        {
          "id": 500,
```

Рис. 3.7

Результат выводится в виде объекта JSON, который мы можем использовать для ответа на запросы погоды. В *списке* имеется 36 элементов. Каждый из этих элементов содержит метеоданные, такие как средняя температура (`temp`), минимальная температура (`temp_min`) и максимальная температура (`temp_max`) в определенное время (`dt` – метка времени). Мы реализуем два метода: `getWeatherAtTime()` и `getWeatherReport()` для генерации короткого отчета о погоде.

```
public String getWeatherReport(String cityCode, Integer i)
                    throws ClientProtocolException, IOException{

  JsonObject currentWeather = null;
  if (cityCode != null){
    currentWeather = getWeatherAtTime(cityCode, i);
  }

  String weatherReport = null;
  if (currentWeather != null){
    JsonObject weather = currentWeather.get("weather")
            .getAsJsonArray().get(0).getAsJsonObject();
    Double avgTemp = Double.valueOf(currentWeather.get("main")
    .getAsJsonObject().get("temp").getAsString()) - 273.15;
    String avgTempSt = String.valueOf(avgTemp).split("\\.")[0];

    weatherReport = "The temperature is " + avgTempSt +
                    " degrees Celsius. "
                    + weather.get("description").getAsString() + ".";
  }
  System.out.println(weatherReport);
  return weatherReport;
}

public JsonObject getWeatherAtTime(String cityCode, Integer i)
                    throws ClientProtocolException, IOException{

  JsonObject json = getWeather(cityCode);
  JsonArray list = json.get("list").getAsJsonArray();
  JsonObject weatherAtTime = list.get(i).getAsJsonObject();
  return weatherAtTime;
}
```

Возможно, вы заметили, что `getWeatherReport()` требует два параметра – код города и время. Время передается как целое число: `0` – текущее время, `1` – следующий слот и т. д. Вызов метода `getWeatherReport()` из метода `main` с именем города и временем в качестве параметров приведет к короткому текстовому отчету о погоде, примерно так:

```
Temperature is 297.8 degrees . clear sky.
```

Очевидно, что при наличии избыточных исходных данных мы можем расширить серверные задачи.

Реализация чат-бота

Теперь, когда серверные задачи сформулированы, сосредоточимся на самом чат-боте. В целом чат-бот будет принимать запросы пользователя в качестве входных данных и отвечать своими высказываниями. Однако, поскольку мы создаем чат-бота для платформы Facebook Messenger, наш чат-бот будет в основном принимать входные данные через нажатия кнопок и отвечать с использованием как высказываний, так и визуально привлекательных карт.

Начнем с реализации класса `Chatbot.java`. Приступим к разработке алгоритма обработки и реагирования на высказывания пользователей. Нам нужно:

- обработать пользовательский ввод;
- обновить контекст;
- определить цель бота;
- создать структуру высказывания и вывода;
- ответить.

Для начала работы это очень простой алгоритм. Сначала обрабатывается ввод пользователя, полученный в виде высказываний или нажатий кнопок. Затем контекст диалога обновляется. На следующем шаге мы определяем, что должен ответить бот. Как только это будет определено, мы выясним, как отреагировать, и отвечаем. Начнем с реализации базовой структуры, основанной на указанном алгоритме.

```
package weatherman.chatbot;

import com.google.gson.JsonObject;

public class Chatbot {
    JsonObject context;

    public static void main(String[] args){}

    public Chatbot(){
        context = new JsonObject();
    }

    public JsonObject process(JsonObject userInput){

        // Шаг 1: обработать ввод пользователя
        JsonObject userAction = processUserInput(userInput);
```

```
        // Шаг 2: обновить контекст
        updateContext(userAction);

        // Шаг 3: определить цели бота
        identifyBotIntent();

        // Шаг 4: вывод структуры
        JsonObject out = getBotOutput();

        return out;
    }
}
```

Теперь мы изменим метод `main()`, чтобы имитировать окно чата, в котором пользователь может вводить свои вопросы и ответы и общаться с ботом.

```
public static void main(String[] args){
    Chatbot c = new Chatbot();
    Scanner scanner = new Scanner(System.in);
    String userUtterance;

    do {
        System.out.print("User:");
        userUtterance = scanner.nextLine();
        // Заканчиваем общение

        if (userUtterance.equals("QUIT")){ break; }
        JsonObject userInput = new JsonObject();
        userInput.add("userUtterance", new JsonPrimitive(userUtterance));
        JsonObject botOutput = c.process(userInput);
        String botUtterance = "";
        if (botOutput != null && botOutput.has("botUtterance")) {
        botUtterance = botOutput.get("botUtterance").getAsString();
        }
        System.out.println("Bot:" + botUtterance);

    } while (true);
}
```

Теперь мы создадим модуль для «понимания» высказываний пользователей. Мы сделаем очень простой модуль, используя правила и *регулярные выражения* для перевода пользовательских высказываний в пользовательские цели.

Цель (намерение) *– формальное однозначное представление того, что говорит пользователь или бот. Оно передает смысл слова или жеста.*

Сначала определим исходный список пользовательских целей для задач, которые мы сформировали в примерах:

- greet
- request_current_weather
- inform_city
- thank

Цель greet представляет собой множество разных приветствий, которые пользователь может использовать. Поэтому мы переведем несколько высказываний, таких как hi, hi there, hello, hello there, в цель greet. Аналогичным образом цель thank представляет все способы, которыми пользователь может поблагодарить бота. Цель request_current_weather используется для представления высказываний, когда пользователь запрашивает текущую информацию о погоде, а цель inform_city – когда он упоминает название города.

```
public JsonObject processUserInput(JsonObject userInput){
    String userUtterance = null;
    JsonObject userAction = new JsonObject();

    // Предопределенный случай
    userAction.add("userIntent", new JsonPrimitive(""));
    if (userInput.has("userUtterance")){
        userUtterance = userInput.get("userUtterance").getAsString();
        userUtterance = userUtterance.replaceAll("%2C", ",");
    }
    if (userUtterance.matches("(hi|hello)( there)?")){
        userAction.add("userIntent", new JsonPrimitive("greet"));
    }
    else if (userUtterance.matches("(thanks)|(thank you)")){
        userAction.add("userIntent", new JsonPrimitive("thank"));
    }
    else if (userUtterance.matches("current weather")||
            userUtterance.matches("weather now")){
        userAction.add("userIntent", new
        JsonPrimitive("request_current_weather"));
    }
    else {
        // Контекстная обработка
```

```
        String currentTask = context.get("currentTask").getAsString();
        String botIntent = context.get("botIntent").getAsString();
        if (currentTask.equals("requestWeather") &&
                botIntent.equals("requestPlace")){
            userAction.add("userIntent", new
            JsonPrimitive("inform_city"));
           userAction.add("cityName", new JsonPrimitive(userUtterance));
        }
    }
    return userAction;
}
```

В предыдущем коде мы выделили цель пользователя из его высказывания. Некоторые цели также связаны с параметрами. Например, у объекта `inform_city` есть связанный параметр `cityName`, который представляет название города, для которого пользователь хочет получить прогноз погоды. Интересно, что цель `inform_city` также является целью, которую мы получаем из контекста. Потому что, когда его спрашивают о городе, пользователь просто упоминает название города. Таким образом, все, что вводится, будет указано как название города и должно быть подтверждено позже. Цель и связанные с ней параметры добавляются в качестве объекта `userAction` объекта JSON. Перейдем к обновлению контекста.

```
public void updateContext(JsonObject userAction){
    // Копирование userIntent
    context.add("userIntent", userAction.get("userIntent"));
    //
    String userIntent = context.get("userIntent").getAsString();
    if (userIntent.equals("greet")){
        context.add("currentTask", new JsonPrimitive("greetUser"));
    } else if (userIntent.equals("request_current_weather")){
        context.add("currentTask", new JsonPrimitive("requestWeather"));
        context.add("timeOfWeather", new JsonPrimitive("current"));
        context.add("placeOfWeather", new JsonPrimitive("unknown"));
        context.add("placeName", new JsonPrimitive("unknown"));
    } else if (userIntent.equals("inform_city")){
        String cityName = userAction.get("cityName").getAsString();
        JsonObject cityInfo = weather.getCityCode(cityName);
        if (!cityInfo.get("cityCode").isJsonNull()){
            context.add("placeOfWeather", cityInfo.get("cityCode"));
            context.add("placeName", cityInfo.get("cityName"));
        }
```

```
    } else if (userIntent.equals("thank")){
        context.add("currentTask", new JsonPrimitive("thankUser"));
    }
}
```

В предыдущем коде мы обновили контекст беседы с использованием ввода от пользователя. Теперь пользовательский ввод преобразуется в задачи и параметры. Идея контекста такая же, как и то, как люди поддерживают контекст диалога во время разговора с кем-то. Переменная `currentTask` представляет собой текущую задачу чат-бота. Такие цели, как `greet`, `thank` и `request_current_weather`, задают эту переменную. Каждая задача будет иметь ряд параметров, которые необходимо заполнить. Например, задача `requestWeather` имеет два слота: время и место. Они должны быть заполнены до того, как бот сможет получить отчет о погоде. Если пользователь запрашивает текущую погоду, время определяется в значении `current`, но место все еще неизвестно. Следующий шаг: определить цель бота!

Обновив контекст, необходимо определить цель чат-бота. Для некоторых задач, таких как приветствие и благодарность пользователю, цели бота довольно просты. Для других сложных задач цели определяются на основе слотов, которые необходимо заполнить, и ответа от задач на стороне сервера. Для бота у нас есть следующие цели:

- greetUser
- thankUser
- requestPlace
- informWeather

Цели `greetUser` и `thankUser` используются, чтобы приветствовать и благодарить пользователей. Цель `requestPlace` используется, когда бот должен знать место для отчета о погоде. Цель `informWeather` используется, когда бот успешно извлек отчет, полученный от серверной службы, которую мы создали в предыдущем разделе.

```
public void identifyBotIntent(){
    String currentTask = context.get("currentTask").getAsString();
    if (currentTask.equals("greetUser")){
        context.add("botIntent", new JsonPrimitive("greetUser"));
    } else if (currentTask.equals("thankUser")){
        context.add("botIntent", new JsonPrimitive("thankUser"));
    } else if (currentTask.equals("requestWeather")){
        if
```

```
        (context.get("placeOfWeather").getAsString().equals("unknown")){
            context.add("botIntent", new JsonPrimitive("requestPlace"));
        }
        else {
            Integer time = -1;
            if
            (context.get("timeOfWeather").getAsString().equals("current")){
                time = 0;
            }
            String weatherReport = null;
            try {
                weatherReport = weather.getWeatherReport(
                    context.get("placeOfWeather").getAsString(), time);
            }
            catch (ClientProtocolException e) {
                e.printStackTrace();
            } catch (IOException e) {
                e.printStackTrace();
            }
            if (weatherReport != null){
                context.add("weatherReport", new
                            JsonPrimitive(weatherReport));
                context.add("botIntent", new JsonPrimitive("informWeather"));
            }
        }
    } else {
        context.add("botIntent", null);
    }
}
```

Обратите внимание, как в предыдущем коде бот решает, что делать, если `currentTask` – это `requestWeather`. Он решает спросить о местоположении, если оно неизвестно. И когда известны время и место, он извлекает информацию и обновляет контекст. Далее цель бота должна быть переведена в высказывание.

```
public JsonObject getBotOutput(){
    JsonObject out = new JsonObject();
    String botIntent = context.get("botIntent").getAsString();
    String botUtterance = "";

    if (botIntent.equals("greetUser")){
        botUtterance = "Hi there! I am WeatherMan, your weather bot! "
            + "What would you like to know? Current weather or forecast?";
```

```
} else if (botIntent.equals("thankUser")){
botUtterance = "Thanks for talking to me! Have a great day!!";
} else if (botIntent.equals("requestPlace")){
botUtterance = "Ok. Which city?";
} else if (botIntent.equals("informWeather")){
String timeDescription =
getTimeDescription(context.get("timeOfWeather").getAsString());
String placeDescription = getPlaceDescription();
String weatherReport = context.get("weatherReport").getAsString();
botUtterance = "Ok. Weather " + timeDescription + " in " +
placeDescription + ". " + weatherReport;
}
out.add("botIntent", context.get("botIntent"));
out.add("botUtterance", new JsonPrimitive(botUtterance));
return out;
}

private String getPlaceDescription() {
return context.get("placeName").getAsString();
} private String getTimeDescription(String timeOfWeather) {
if (timeOfWeather.equals("current")){
return "now";
}
return null;
}
```

Теперь, когда цель и высказывание бота определены, выполним метод `main`! Запустите класс `Chatbot`, нажав сочетание клавиш **Ctrl+F11**. В консоли появится предложение начать диалог. Взгляните на следующий пример:

```
Пользователь: hi there.
Бот: Hi there! I am WeatherMan, your weather bot! What would you like to know?
Current weather or forecast?
Пользователь: current weather.
Бот: Ok. Which city?
Пользователь: London,GB.
Бот: Ok. Weather now in London,GB. The temperature is 291.72 degrees Farenheit.
broken clouds.
Пользователь: thanks.
Бот: Thanks for talking to me! Have a great day!!
Пользователь: QUIT.
```

Как вы могли заметить, здесь есть много незавершенных задач. Например, название города должно быть напечатано в том же

формате (`названиегорода, кодстраны`), что и в нашей базе данных погоды. Это не очень хорошо. Пользователь может сказать «в Лондоне» и должен получить ответ. Мы вернемся к этому в следующей главе, где рассмотрим, как обрабатывать высказывания, когда пользователи могут вызывать один и тот же объект разными запросами. Но сначала запустим этот код в Facebook Messenger.

Создание веб-службы чат-бота

Наш следующий шаг – реализовать чат-бота в виде веб-службы. Это нужно для того, чтобы платформы типа Facebook Messenger могли получить доступ к чат-боту без необходимости размещения на своих серверах. Чтобы сделать чат-бота доступным в виде веб-службы, код чат-бота должен быть упакован в виде веб-сервера и размещен на облачной платформе. Мы воспользуемся библиотекой Java Spark для переноса кода чат-бота в качестве веб-сервера в облачную платформу Heroku для его размещения.

1. Добавьте зависимость Spark в POM-файл.

```
<!-- https://mvnrepository.com/artifact/com.sparkjava/spark-core -->
<dependency>
    <groupId>com.sparkjava</groupId>
    <artifactId>spark-core</artifactId>
    <version>2.0.0</version>
</dependency>
```

2. Подключите класс Java под названием `WebServer.java`. Обязательно поместите его в пакет по умолчанию.

```
public class WebServer {
    public static void main(String[] args) {
        Spark.setPort(getHerokuAssignedPort());
        Spark.staticFileLocation("/public");
        final Chatbot bot = new Chatbot();

        get("/", (req, res) -> "Hello World! I am WeatherMan, the weather bot!!");

        // дескриптор для чат-бота WeatherMan
        post("/bot", new Route() {
            public Object handle(Request request, Response response) {
                String body = request.body();
                System.out.println("body: " + body);
```

```
            String splitChar = “&”;
            String keyValueSplitter = “=”;
            String[] params = body.split(splitChar);
            String userUtterance = “null”;
            for (int i=0; i < params.length; i++){
                String[] sv = params[i].split(keyValueSplitter);
                if (sv[0].equals(“userUtterance”)){
                    if (sv.length > 0){
                        userUtterance = sv[1];
                    } else {
                        userUtterance = “”;
                    }
                    userUtterance = userUtterance.replaceAll(“%20”,
                    “”);
                    userUtterance = userUtterance.replaceAll(“%3A”,
                    “:”);
                }
            }
            if (!userUtterance.equals(“null”)){
                System.out.println(“User says:” + userUtterance);
                JsonObject userInput = new JsonObject();
                userInput.add(“userUtterance”, new
                JsonPrimitive(userUtterance));
                String botResponse = bot.processFB(userInput);
                System.out.println(“Bot says:” + botResponse);
                if (botResponse != null) {
                    return botResponse;
                }
            } else {
                return null;
            }
            response.status(400);
            return new ResponseError(“Error! POST not handled.”);
        }
    }, json());

    after((req, res) -> { res.type(“application/json”); });

    exception(IllegalArgumentException.class, (e, req, res) -> {
        res.status(400);
        res.body(toJson(new ResponseError(e)));
    });
}

static int getHerokuAssignedPort() {
```

```
        ProcessBuilder processBuilder = new ProcessBuilder();
        if (processBuilder.environment().get("PORT") != null) {
            return
Integer.parseInt(processBuilder.environment().get("PORT"));
        }
        return 4567;
        //вернуть порт по умолчанию, если порт heroku не установлен (т. е.
localhost)
    }
}
```

3. Подключите вспомогательные классы веб-сервера. Класс `WebServer.java` импортирует классы из пакетов Spark и Gson. Ему также нужны два вспомогательных класса: `JSONUtil.java` и `ResponseError.java`. Вспомогательный класс `JSONUtil` помогает `WebServer` преобразовать формат ответа в JSON.

```
package weatherman.web.utils;

import com.google.gson.Gson;
import spark.ResponseTransformer;

public class JSONUtil {
    public static String toJson(Object object) {
        return new Gson().toJson(object);
    }
    public static ResponseTransformer json() {
        return JSONUtil::toJson;
    }
}
```

Класс `ResponseError` помогает классу `WebServer` сообщать об ошибках в определенном формате, которые могут возникнуть во время работы веб-службы.

```
package weatherman.web.utils;

public class ResponseError {
    private String errorType;
    public ResponseError(String message, String. args) {
        this.errorType = String.format(message, args);
    }
    public ResponseError(Exception e) {
```

```
        this.errorType = e.getMessage();
    }
    public String getMessage() {
        return this.errorType;
    }
}
```

4. Смените компилятор Java на версию 1.8. Файл pom.xml сообщает Heroku, какой компилятор должен использоваться при создании проекта в облаке. Поскольку мы используем лямбда-выражения, нам нужно установить компилятор Java 1.8. Для этого добавьте показанный ниже код (элемент `build`) в файл *pom.xml* сразу после элемента `dependencies`:

```
<build>
   <plugins>
      <plugin>
         <artifactId>maven-compiler-plugin</artifactId>
         <version>3.1</version>
         <configuration>
            <source>1.8</source>
            <target>1.8</target>
         </configuration>
         <executions>
            <execution>
               <id>default-compile</id>
               <phase>compile</phase>
               <goals>
                  <goal>compile</goal>
               </goals>
            </execution>
            <execution>
               <id>default-testCompile</id>
               <phase>test-compile</phase>
               <goals>
                  <goal>testCompile</goal>
               </goals>
            </execution>
         </executions>
      </plugin>
   </plugins>
</build>
```

Сохраните файл *pom.xml*. Щелкните правой кнопкой мыши по названию проекта, выберите пункт **Maven**, а пункт **Update Project** (Обновить проект) в контекстном меню. Нажмите кнопку **OK** в диалоговом окне. Так вы обновите настройки Maven.

5. Добавьте артефакт Heroku `Procfile`. `Procfile`. В этом файле мы сообщаем Heroku, что программа запускается после создания проекта. Подключите файл *Procfile* следующим образом:

```
web: java -cp target/classes:target/dependency/* WebServer
```

Предыдущая строка сообщает Heroku о запуске класса `WebServer` в пакете *по умолчанию* на веб-дино.

Совет. Имя файла должно быть ***Procfile***, а не *Procfile.txt* или *ProcFile*.

6. Передайте данные в облако. Теперь, когда мы создали нашу первую версию чат-бота в виде веб-сервера, разместим ее в облаке и сделаем доступной в виде веб-службы. В окне консоли проекта `Weatherman` выполните следующие команды:

```
> git init
> git add .
> git commit -m "chatbot for weather v1"
```

Выполнив эти команды, мы создали и сохранили файлы проекта в локальном репозитории GIT. Теперь создадим приложение Heroku для хранения нашего веб-приложения.

```
> heroku create weatherman-bot
```

Вы увидите результаты выполнения этой команды, как показано на следующем рисунке. Таким образом, мы создали приложение Heroku с именем `weatherman-bot` с URL-адресом `https://weatherman-bot.herokuapp.com` – рис. 3.8.

Наконец, мы поместим код в приложение Heroku, а также вызовем его как веб-службу. Для этого выполните следующую команду:

```
> git push heroku master
```

Вы увидите показанные ниже сообщения, что означает, что код был доставлен, успешно скомпилирован и запущен, – рис. 3.9.

```
C:\WINDOWS\system32\cmd.exe
C:\Users\Srini\Documents\_Book\workspace>cd weatherman

C:\Users\Srini\Documents\_Book\workspace\weatherman>git init
Initialized empty Git repository in C:/Users/Srini/Documents/_Book/workspace/weatherman/.git/

C:\Users\Srini\Documents\_Book\workspace\weatherman>git add .

C:\Users\Srini\Documents\_Book\workspace\weatherman>git commit -m "chatbot for weather version 1"
[master (root-commit) 4ef4011] chatbot for weather version 1
 20 files changed, 593 insertions(+)
 create mode 100644 .classpath
 create mode 100644 .project
 create mode 100644 .settings/org.eclipse.jdt.core.prefs
 create mode 100644 .settings/org.eclipse.m2e.core.prefs
 create mode 100644 ProcFile
 create mode 100644 pom.xml
 create mode 100644 src/main/java/weatherman/chatbot/Chatbot.java
 create mode 100644 src/main/java/weatherman/weather/Weather.java
 create mode 100644 src/main/java/weatherman/web/WebServer.java
 create mode 100644 src/main/java/weatherman/web/utils/JSONUtil.java
 create mode 100644 src/main/java/weatherman/web/utils/ResponseError.java
 create mode 100644 target/classes/META-INF/MANIFEST.MF
 create mode 100644 target/classes/META-INF/maven/com.sjanarthanam/weatherman/pom.properties
 create mode 100644 target/classes/META-INF/maven/com.sjanarthanam/weatherman/pom.xml
 create mode 100644 target/classes/weatherman/chatbot/Chatbot.class
 create mode 100644 target/classes/weatherman/weather/Weather.class
 create mode 100644 target/classes/weatherman/web/WebServer$1.class
 create mode 100644 target/classes/weatherman/web/WebServer.class
 create mode 100644 target/classes/weatherman/web/utils/JSONUtil.class
 create mode 100644 target/classes/weatherman/web/utils/ResponseError.class

C:\Users\Srini\Documents\_Book\workspace\weatherman>heroku create weatherman-bot
Creating weatherman-bot... done
https://weatherman-bot.herokuapp.com/ | https://git.heroku.com/weatherman-bot.git

C:\Users\Srini\Documents\_Book\workspace\weatherman>
```

Рис. 3.8

```
C:\WINDOWS\system32\cmd.exe
remote:       [INFO] Downloaded: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus/3.1/plexus-3.1.pom (19 KB at 2019.9 KB/sec)
remote:       [INFO] Downloading: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-digest/1.0/plexus-digest-1.0.pom
remote:       [INFO] Downloaded: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-digest/1.0/plexus-digest-1.0.pom (2 KB at 129.3 KB/
sec)
remote:       [INFO] Downloading: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-components/1.1.7/plexus-components-1.1.7.pom
remote:       [INFO] Downloaded: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-components/1.1.7/plexus-components-1.1.7.pom (5 KB
at 607.3 KB/sec)
remote:       [INFO] Downloading: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-utils/3.0.5/plexus-utils-3.0.5.jar
remote:       [INFO] Downloading: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-digest/1.0/plexus-digest-1.0.jar
remote:       [INFO] Downloaded: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-digest/1.0/plexus-digest-1.0.jar (12 KB at 526.4 KB
/sec)
remote:       [INFO] Downloaded: https://repo.maven.apache.org/maven2/org/codehaus/plexus/plexus-utils/3.0.5/plexus-utils-3.0.5.jar (226 KB at 9376.
9 KB/sec)
remote:       [INFO] Installing /tmp/build_c8544ea7c3a5ce51ff6859e4aefa109e/target/weatherman-0.0.1-SNAPSHOT.jar to /app/tmp/cache/.m2/repository/co
m/sjanarthanam/weatherman/0.0.1-SNAPSHOT/weatherman-0.0.1-SNAPSHOT.jar
remote:       [INFO] Installing /tmp/build_c8544ea7c3a5ce51ff6859e4aefa109e/pom.xml to /app/tmp/cache/.m2/repository/com/sjanarthanam/weatherman/0.0
.1-SNAPSHOT/weatherman-0.0.1-SNAPSHOT.pom
remote:       [INFO] ------------------------------------------------------------------------
remote:       [INFO] BUILD SUCCESS
remote:       [INFO] ------------------------------------------------------------------------
remote:       [INFO] Total time: 10.149 s
remote:       [INFO] Finished at: 2017-06-19T14:38:13+00:00
remote:       [INFO] Final Memory: 22M/160M
remote:       [INFO] ------------------------------------------------------------------------
remote: -----> Discovering process types
remote:        Procfile declares types -> (none)
remote:
remote: -----> Compressing...
remote:        Done: 51.1M
remote: -----> Launching...
remote:        Released v3
remote:        https://weatherman-bot.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/weatherman-bot.git
 * [new branch]      master -> master
```

Рис. 3.9

Теперь, когда мы успешно создали чат-бота в качестве веб-службы, нам нужно настроить этого бота на общение с пользователями в Facebook Messenger.

Публикация в Facebook Messenger

Чтобы опубликовать пользовательского чат-бота в Facebook, вам необходимо выполнить приведенные далее действия.

Создание страницы Facebook

Вам нужна страница Facebook, на которой может быть размещен чат-бот. Чтобы создать страницу Facebook, вам понадобится учетная запись Facebook. Если у вас ее нет, то создайте ее на сайте **www.facebook.com**.

Когда вы авторизуетесь на сайте Facebook, то сможете создавать любое количество страниц. Чтобы создать страницу, выполните следующие действия.

1. Слева находится меню с тремя вкладками: **Shortcuts** (Быстрые ссылки), **Explore** (Интересное) и **Create** (Создать). В разделе **Create** (Создать) щелкните мышью по ссылке **Page** (Страница).

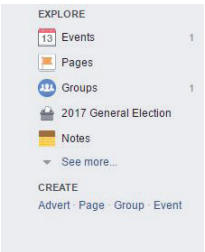


Рис. 3.10

2. Страница Facebook может быть создана как общедоступный профиль для представления компаний, брендов, публичных личностей, мероприятий и т. д. Вы увидите показанное ниже меню – рис. 3.11:
   Выберите пункт **Artist, Band or Public Figure** (Исполнитель, группа или публичная личность). Поскольку мы создаем чат-бота, выберите пункт **Fictional Character** (Вымышленный персонаж), укажите его название и щелкните мышью по ссылке **Get Started** (Начать) – рис. 3.12.
3. Страница готова. Настроим ее – рис. 3.13.
   Сделайте закладку для вашей страницы. Добавьте изображение профиля, щелкнув мышью по ссылке **Add a Picture** (Добавить изображение), и изображение баннера, щелкнув мышью по ссылке **Add a Cover** (Добавить обложку) (всегда используйте изображения, на которые у вас есть авторские права).

Совет. Не беспокойтесь, если вы случайно закрыли свой браузер и не добавили в избранное свою страницу. Вы можете просмотреть все свои страницы по адресу **www.facebook.com/bookmarks/pages**.

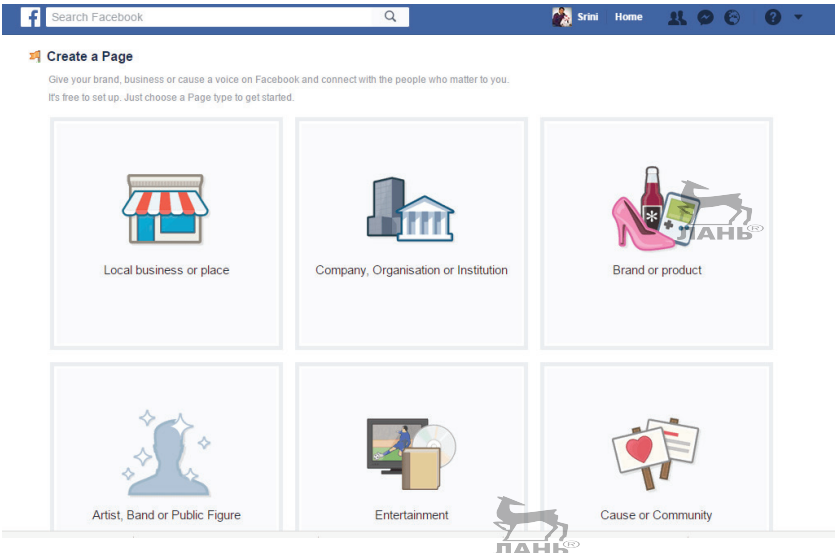


Рис. 3.11

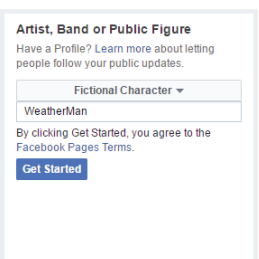


Рис. 3.12

Теперь вы можете сделать свой первый пост на странице.

Создание имени пользователя

Создав страницу, выполним еще две настройки, прежде чем двигаться дальше: добавим описание страницы и создадим имя пользователя.

1. Чтобы создать имя пользователя, щелкните мышью по ссылке **Create Page @ username** (Создать страницу @ имя пользователя) под изображением профиля – рис. 3.14.

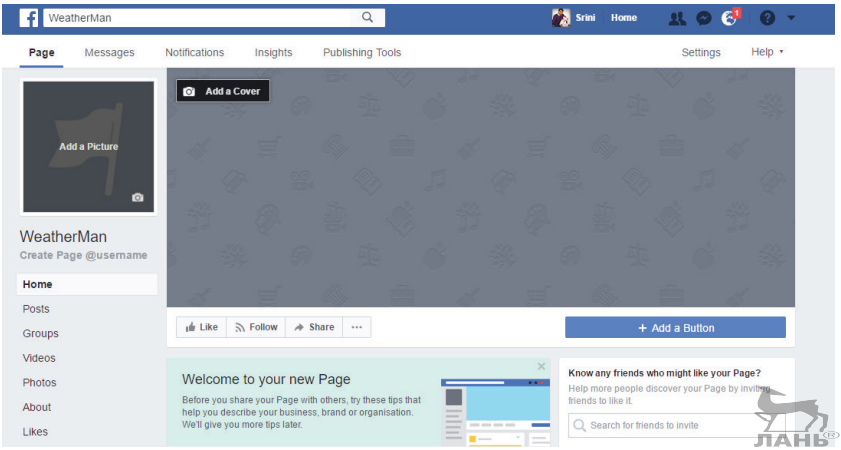


Рис. 3.13

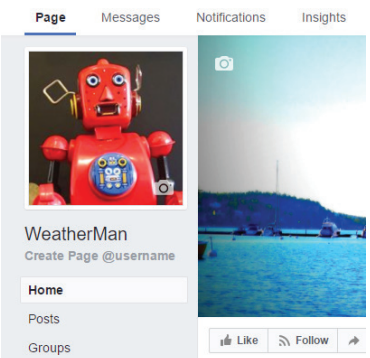


Рис. 3.14

2. Вам будет предложено ввести имя пользователя. Это имя однозначно идентифицирует страницу и упрощает запоминание URL-адреса. Возможно, вам придется попробовать несколько имен, поскольку то, которое вы ввели, уже может быть занято – рис. 3.15.
3. Наконец, когда все будет настроено, вы будете уведомлены о новой странице и URL-адресе Messenger – рис. 3.16.

Создание приложения Facebook

Вам нужно создать приложение Facebook, которое может подключиться к странице Facebook, которую мы только что создали.

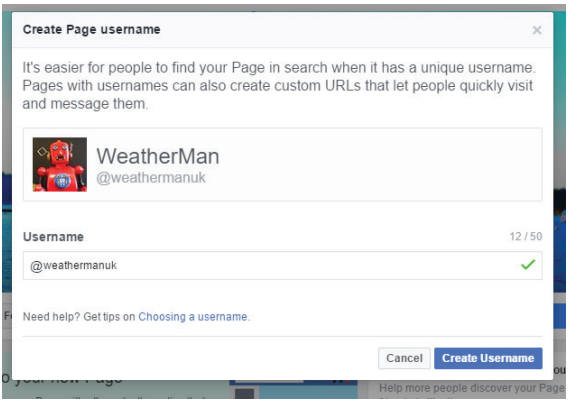


Рис. 3.15

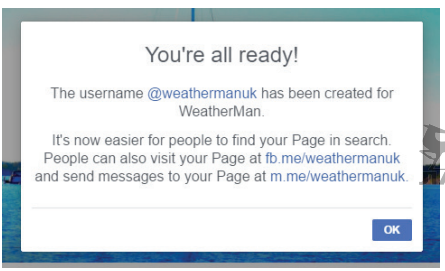


Рис. 3.16

Приложение Facebook можно создать на портале разработчиков Facebook. Выполните следующие шаги.

1. Перейдите в браузере на сайт **developers.facebook.com**.
2. Щелкните мышью по ссылке **MyApps** (МоиПриложения) в правом верхнем углу (рядом с изображением вашего профиля).
3. Щелкните мышью по ссылке **Add A New App** (Добавить новое приложение).
4. Заполните отображаемое имя приложения и контактный адрес электронной почты и щелкните мышью по ссылке **Create App ID** (Создать идентификатор приложения) – рис. 3.17.
5. Создав оболочку, нам нужно добавить к нему приложение. В нашем случае, поскольку мы создаем чат-бота, нам нужно добавить в оболочку приложение Messenger – рис. 3.18.
6. Наведите указатель мыши на значок **Messenger** и щелкните мышью по ссылке **Set Up** (Настроить). Messenger будет

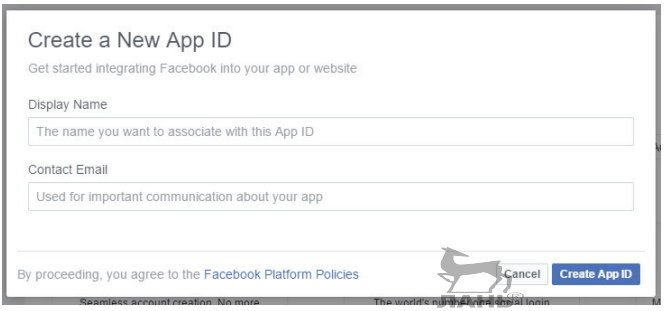


Рис. 3.17

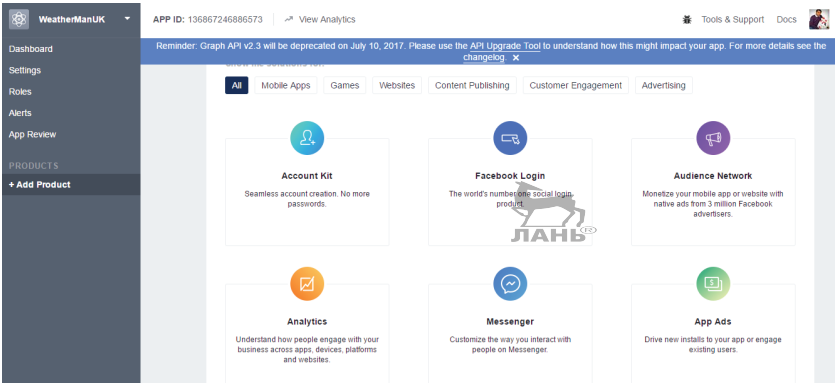


Рис. 3.18

добавлен в приложение, и откроется страница настроек. Прокрутите страницу вниз до раздела **Token Generation** (Генерация токена). Так мы связываем приложение со страницей, которую мы создали.

7. Щелкните мышью по ссылке **Select A Page** (Выбрать страницу) и выберите страницу, к которой нужно привязать приложение. Откроется диалоговое окно проверки подлинности, в котором вам нужно разрешить приложению доступ к странице. Щелкните мышью по ссылке **Continue as** X (Продолжить как X), где X – ваше имя. Не беспокойтесь о предупреждениях на данный момент – рис. 3.19.
8. На следующей странице щелкните мышью по ссылке **OK**, чтобы завершить процесс. Диалоговое окно автоматически закроется – рис. 3.20.

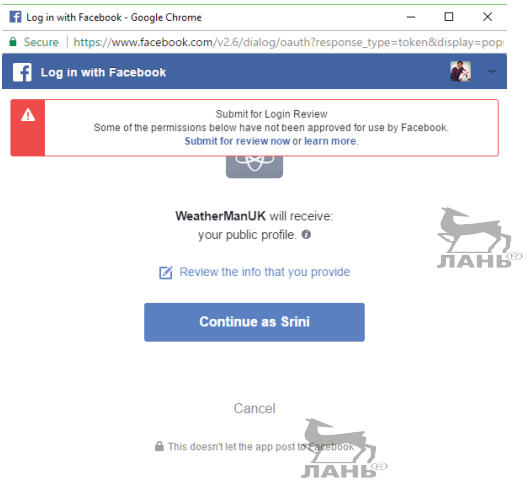


Рис. 3.19

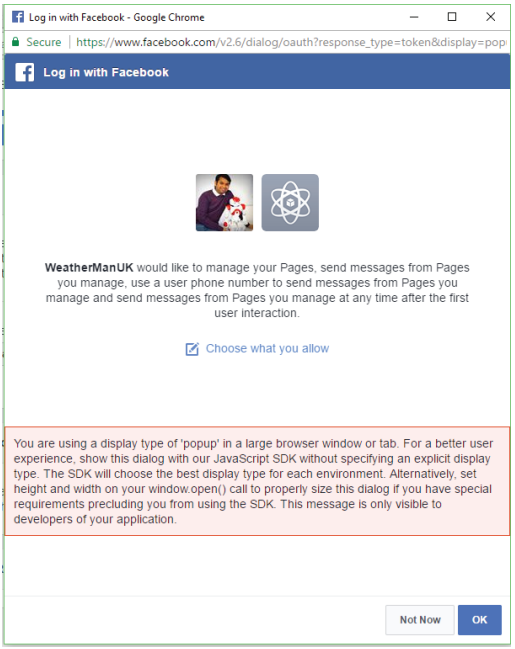


Рис. 3.20

9. Теперь вы увидите, что на странице назначен токен, называемый **Page Access Token** (Токен доступа к странице). Скопируйте его и сохраните.
10. Для завершения процесса нам нужно настроить вебхук, но мы сделаем это чуть позже.

Создание интерфейса для веб-приложения Facebook

На этом этапе мы создадим интерфейс, который позволит нашему чат-боту общаться с Facebook-приложением. В предыдущем разделе мы создали приложение Facebook. Это приложение должно будет взаимодействовать с интерфейсом, который мы создадим в данном разделе. Информация об этом модуле должна войти в настройки вебхука, которые мы оставили незавершенными в предыдущем разделе. Мы создадим новый проект для этого интерфейса и назовем его `weatherman-fbmi`. В консоли создайте отдельный каталог (*weatherman-fbmi*), в котором будет храниться данный проект. В этом каталоге мы создадим следующие три файла:

- *index.js*
- *package.json*
- *Procfile*

index.js – это сценарий Node.js, который действует как веб-сервер для отправки и получения сообщений в нашем приложении Facebook.

```
'use strict';
const express = require('express')
const bodyParser = require('body-parser')
const request = require('request')

const app = express()

// установите токен доступа к странице Weatherman в сети FB
const token = 'ТокенДоступаКстраницеFB';

const botServerUrl = ,https://weatherman-bot.herokuapp.com/bot';

app.set(,port', (process.env.PORT || 5000))

// Выполнение application/x-www-form-urlencoded
app.use(bodyParser.urlencoded({extended: false}))
```

```
// Выполнение application/json
app.use(bodyParser.json())

app.use(express.static('public'))

// Индексный маршрут
app.get('/', function (req, res) {
    res.send('Hello world, I am Weatherman!.')
})

// Поднимаем сервер
app.listen(app.get('port'), function() {
    console.log('running on port', app.get('port'))
})
```

Вам нужно заменить значение `ТокенДоступаКстраницеFB` маркером доступа к странице, который вы получили в шаге 9 предыдущего раздела. Возможно, вам также придется заменить значение параметра `botServerURL` на свое собственное.

Предыдущего кода достаточно, чтобы развернуть сервер на платформе Heroku. Но нам нужен сервер, чтобы создать мост между нашим Facebook-приложением и чатом. Теперь мы внедрим необходимые маршруты на предыдущий веб-сервер. Добавьте в файл *index.js* показанный ниже код:

```
// Проверка на Facebook
app.get('/webhook/', function (req, res) {
    if (req.query['hub.verify_token'] === 'iam-weatherman-bot') {
        res.send(req.query['hub.challenge'])
    }
    res.send('Error, wrong token')
})

// Вебхук FBM
app.post('/webhook/', function (req, res) {
    console.log(JSON.stringify(req.body));
    let messaging_events = req.body.entry[0].messaging
    for (let i = 0; i < messaging_events.length; i++) {
        let event = req.body.entry[0].messaging[i]
        let sender = event.sender.id
        let recipient = event.recipient.id
        let time = req.body.entry[0].time
```

```
        // Вызываем чат-бота
        if (event.message && event.message.text) {
            let text = event.message.text
            //передача боту
            request({
                url: botServerUrl,
                method: 'POST',
                form: {
                    'userUtterance':text
                }
            }, function (error, response, body) {
                // Ответ от бота
                if (!error && response.statusCode == 200) {
                    // Вывод ответа бота
                    body = body.substring(1,body.length-1);
                    body = body.replace(/\\/g, '')
                    let botOut = JSON.parse(body)
                    if (botOut.botUtterance != null){
                        sendTextMessage(sender, botOut.botUtterance)
                    }
                } else {
                    sendTextMessage(sender, 'Error!')
                }
            }); // запрос завершения
        }
    }
    res.sendStatus(200)
})

function sendTextMessage(sender, text) {
    if (text != 'null'){
        let messageData = { 'text':text }
        request({
            url: 'https://graph.facebook.com/v2.6/me/messages',
            qs: {access_token:token},
            method: 'POST',
            json: {
                recipient: {id:sender},
                message: messageData,
            }
        }, function(error, response, body) {
            if (error) {
                console.log('Error sending messages: ', error)
```

```
            } else if (response.body.error) {
                console.log('Error: ', response.body.error)
            }
        })
    }
}
```

В предыдущем коде создается POST-маршрут, называемый вебхук, который будет вызываться приложением Facebook с высказываниями пользователя. Он будет отправлен чат-боту через его URL-адрес (botServerURL) в качестве еще одного запроса POST. Ответ бота отправляется обратно в Messenger методом sendTextMessage(). Существуют другие форматы сообщений, которые могут использоваться для отправки более полных ответов от бота. Мы рассмотрим их позднее.

Мы создадим файл *package.json*, чтобы сообщить Heroku о пакетах, которые необходимы файлу *index.js*. Создайте файл *package.json* в текстовом редакторе со следующим JSON-кодом:

```
{
 "name": "weatherman-bot-server",
 "version": "1.0.0",
 "description": "Weatherman bot fbmi server",
 "main": "index.js",
 "scripts": {
  "test": "echo \"Error: no test specified\" && exit 1"
 },
 "author": "Srini Janarthanam",
 "license": "ISC",
 "dependencies": {
 "body-parser": "^1.15.2",
 "express": "^4.14.0",
 "request": "^2.72.0"
 }
}
```

В качестве последнего шага нам нужно создать файл *Procfile*, чтобы сообщить Heroku о запуске файла *index.js*. Создайте файл *Procfile* в текстовом редакторе с помощью следующей команды:

```
web: node index.js
```

Выполните следующие шаги, чтобы доставить программу интерфейса в облако. В окне консоли проекта интерфейса Weatherman

выполните следующие команды для создания приложения Heroku и выгрузите приложение интерфейса в облако:

```
> git init
> git add .
> git commit -m “fbmi for weatherman v1”
> heroku create weatherman-bot-fbmi
> git push heroku master
```

Вы получите сообщение с URL-адресом приложения. В моем случае это **https://weatherman-bot-fbmi.herokuapp.com**.

Связывание приложения Facebook и приложения интерфейса

Наконец, мы должны создать вебхук, который мы бросили на шаге 10 при разработке приложения Facebook. Вы, возможно, заметили, что мы создали два маршрута для вебхука. Способ `GET` используется для проверки приложения, а способ `POST` – для обмена сообщениями с чат-ботом в обе стороны – рис. 3.21.

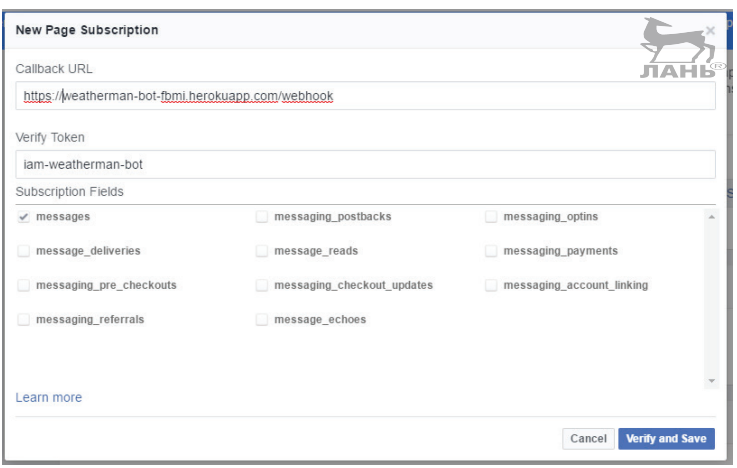


Рис. 3.21

Подпишитесь на страницу WeatherMan Facebook, чтобы получать сообщения, отправленные пользователями Messenger в веб-службе рис. 3.22.

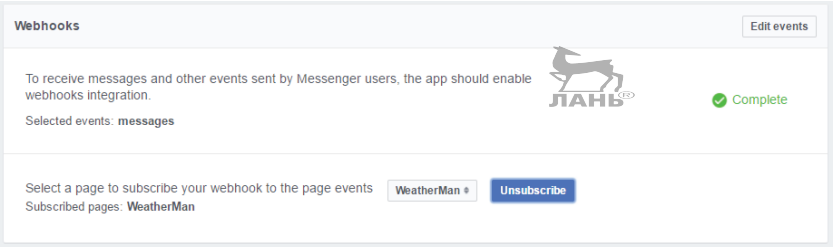


Рис. 3.22

Теперь мы готовы протестировать чат-бота!

Тестирование чат-бота

Теперь, когда чат-бот запущен и работает, мы можем протестировать его на соответствующей странице Facebook. Имя пользователя для страницы было создано в разделе «Создание имени пользователя». Перейдите по адресу **m.me/weathermanuk** в веб-браузере. Откроется интерфейс Facebook Messenger, где вы сможете общаться с нашим ботом WeatherMan. Скажите ему привет – рис. 3.23.

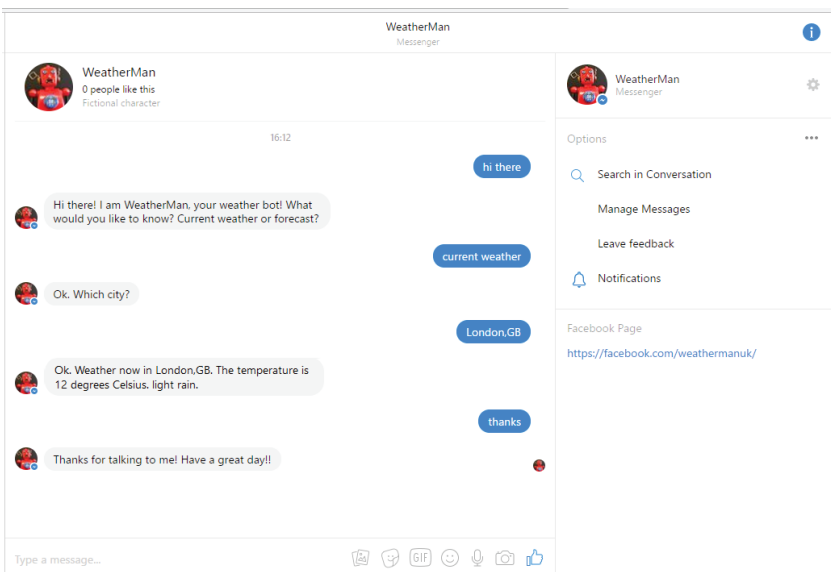


Рис. 3.23

Добавление других тестировщиков

Вы могли заметить, что только у вас есть доступ к чат-боту. Любой, кто перейдет по тому же URL-адресу и напишет слово «привет!», не получит ответа от чат-бота. Это связано с тем, что чат-бот еще не опубликован. Вы можете добавить других тестировщиков, чтобы помочь вам в проверке чата. Для этого выполните следующие действия.

1. Перейдите в своем браузере по адресу **developers.facebook.com**.
2. Выберите свое приложение.
3. На панели управления выберите пункт **Roles** (Роли).
4. На вкладке **Testers** (Тестеры) вы можете добавить пользователей Facebook, указав их имена.

Добавленные пользователи будут иметь доступ к чат-боту. Но помните, ваш бот до сих пор не опубликован – рис. 3.24.

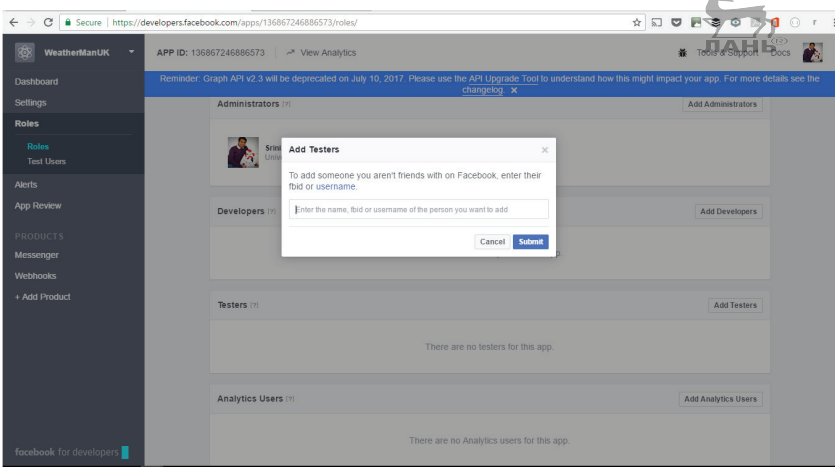


Рис. 3.24

Публикация для общего доступа

Последним шагом является публикация чат-бота. Этот последний шаг сделает чат-бота доступным для всех по URL-адресу Messenger (**m.me/weathermanuk**).

1. На панели управления щелкните мышью по ссылке **Settings** (Настройки), а затем выберите пункт **Basic** (Основные).
2. Загрузите значок для представления чата. Это должно быть изображение размером 1024×1024 пикселей.

3. Выберите категорию **Apps for Messenger** (Приложения для Messenger).
4. Добавьте URL-адрес в политику конфиденциальности. Эта политика должна сообщать пользователям, что происходит с собранными пользовательскими данными. Существует несколько служб, которые могут помочь вам создать политики конфиденциальности Facebook.
   - IUBENDA: **www.iubenda.com/blog/privacy-policy-facebook-app/**;
   - TermsFeed: **termsfeed.com/blog/privacy-policy-url-facebook-app/**.
5. На панели управления нажмите кнопку **App Review** (Обзор приложения).
6. Сделайте свое приложение общедоступным, выбрав вариант **Yes** (Да).
7. На панели управления выберите пункт **Messenger** в разделе **Products** (Продукты).
8. Справа найдите вкладку **App Review For Messenger** (Просмотр приложений для Messenger).
9. Добавьте пункт **page_messaging** к представлению.
10. Перейдите на вкладку **Current Submission** (Текущее представление).
11. Щелкните мышью по ссылке **Edit Notes** (Редактировать заметки) на странице **page_messaging**.
12. Выберите страницу Facebook, на которую подписан чат-бот.
13. Выберите вариант **Your Messenger experience includes automated replies** (Опыт работы с Messenger включает автоматические ответы) и укажите примеры, что пользователи могут спросить и что должен ответить бот.
14. Добавьте любые другие примечания, если необходимо. Это поможет рецензенту разобраться в чат-боте и узнать, оправдывает ли он ожидания.
15. Нажмите кнопку **Save** (Сохранить). Вернитесь к настройкам Messenger. Вы увидите кнопку **Submit for review** (Отправить для просмотра). Нажмите ее для отправки данных. Если кнопка не активна, вы увидите список элементов, которые нужно настроить, прежде чем вы сможете отправить данные.

Вы сразу же получите сообщение, которое сможете просмотреть на странице оповещений. Нажмите кнопку **Alerts** (Оповещения) на панели управления. Появится сообщение о том, что приложение

проверяется. Проверка обычно занимает пару дней. После того как приложение будет рассмотрено и одобрено, ваш чат-бот готов к публичному использованию.

Поздравляю! Вы уже продвинулись так далеко!

Дополнительные форматы сообщений Facebook

В этом разделе мы рассмотрим другие форматы сообщений Facebook, помимо текста. В предыдущем примере мы запрограммировали чат-бота для отправки текстовых ответов пользователю. Тем не менее Facebook поддерживает больше форматов, которые могут быть использованы для обеспечения более комфортного обмена сообщениями с пользователем. Вы можете настроить формат, изменив содержимое элемента `message` в запросе `POST` в методе `sendTextMessage()` в файле *index.js*.

Типы содержимого

Помимо текста, пользователям могут быть отправлены другие типы контента, такие как изображения, аудио- и видеоролики и документы.

Вложения изображений

Пользователю могут быть отправлены изображения в форматах JPEG, PNG и GIF с помощью сообщения следующего формата:

```
"message":{
        "attachment":{
            "type":"image",
            "payload":{
                "url":"https://myimagelibrary.com/ijustwannadrawsomething.jpg"
        }
    }
}
```

Звуковые вложения

Аудиоклипы могут быть отправлены пользователю с помощью следующего элемента `message`:

```
"message":{
    "attachment":{
```

```
        "type":"audio",
        "payload":{
            "url":"https://mymp3library.com/ijustwannasing.mp3"
        }
    }
}
```

Вложения видео

Видеофайлы можно отправить через Messenger, указав URL-адрес с сообщением следующего содержимого:

```
"message":{
    "attachment":{
        "type":"video",
        "payload":{
            "url":"https://myvideolibrary.com/ijustwannadance.mp4"
        }
    }
}
```

Документы

Документы могут быть отправлены пользователям с помощью сообщения следующего формата, с указанием URL-адреса файла.

```
"message":{
    "attachment":{
        "type":"file",
        "payload":{
            "url":"https://myfileslibrary.com/ijustwannawrite.pdf"
        }
    }
}
```

Шаблоны

Помимо отправки простого текста и других типов контента, Facebook предоставляет способ отправки карточек. Это GUI-элементы, которые визуально привлекательны. Существуют также варианты добавления в карты кнопок, которые служат для ответа пользователя.

Шаблон кнопки

Шаблоны кнопок могут быть созданы с помощью сообщения показанного ниже формата. Следующее сообщение отобразит карту с двумя кнопками. Подписи на кнопках будут **Buy now** и **More info**. Щелчок по кнопке **Buy now** отправит полезную нагрузку (PAYLOAD_BUY) в качестве ответного сообщения обратно боту. При нажатии кнопки **More info** пользователь перейдет на веб-страницу, указанную во встроенном окне веб-просмотра.

```
"message":{
    "attachment":{
        "type":"template",
        "payload":{
            "template_type":"button",
            "text":"What would you like to do now?",
            "buttons":[
                {
                    "type":"postback",
                    "title":"Weather this weekend",
                    "payload":"PAYLOAD_WEEKEND_LONDON"
                },
                {
                    "type":"Goto Website",
                    "url":"https://myweather.com/london",
                    «title»:»More Info»
                }
            ]
        }
    }
}
```

Предыдущее сообщение будет отображаться так, как показано на следующем рисунке – 3.25.

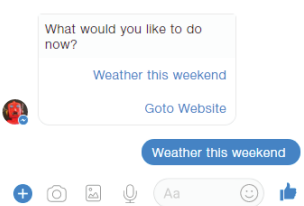


Рис. 3.25

Шаблон списка

Шаблоны списков можно использовать для отображения вертикального списка элементов. Используя показанный ниже формат сообщения, вы можете составить список, представленный визуально привлекательным образом.

```
“message”: {
    “attachment”: {
        “type”: “template”,
        “payload”: {
            “template_type”: “list”,
            “top_element_style”: “compact”,
            “elements”: [
                {
                    “title”: “London,GB”,
                    “image_url”: “https://myweather.com/london.png”,
                    “subtitle”: “Weather in London,GB”,
                    “default_action”: {
                        “type”: “web_url”,
                        “url”: “https://myweather.com/london”,
                        “messenger_extensions”: true,
                        “webview_height_ratio”: “tall”,
                        “fallback_url”: “https://myweather.com/”
                    },
                    “buttons”: [
                        {
                            “title”: “Info”,
                            “type”: “web_url”,
                            “url”: “https://myweather.com/london/weather”,
                            “messenger_extensions”: true,
                            “webview_height_ratio”: “tall”,
                            “fallback_url”: “https://myweather.com/”
                        }
                ]
            },
            {
                “title”: “Paris,FR”,
                “image_url”: “https://myweather.com/paris.png”,
                “subtitle”: “Weather in Paris,FR”,
                “default_action”: {
                    “type”: “web_url”,
                    “url”: “https://myweather.com/paris”,
                    “messenger_extensions”: true,
                    “webview_height_ratio”: “tall”,
```

```
                    "fallback_url": "https://myweather.com/"
                },
                "buttons": [
                    {
                        "title": "Info",
                        "type": "web_url",
                        "url": "https://myweather.com/paris/weather",
                        "messenger_extensions": true,
                        "webview_height_ratio": "tall",
                        "fallback_url": "https://myweather.com/"
                    }
                ]
            }
        ],
        "buttons": [
            {
                "title": "View More",
                "type": "postback",
                «payload»: «payload»
            }
        ]
    }
}
```

Предыдущий код также служит примером того, как вы можете использовать шаблон списка в чате погоды. Вы можете составить список известных городов или позволить пользователю создать собственный список с городами. Этот список будет содержать ссылки для получения информации о погоде в списке городов. В предыдущем коде это делается путем добавления информации, ориентированной на город, в список массива `elements` JSON-кода. Каждый элемент имеет два URL-адреса. Один из них – для того момента, когда пользователь касается самого элемента, а другой – для того, когда пользователь нажимает кнопку `call-to-action`.

В предыдущем коде кнопки используются для того, чтобы отправить пользователя на URL-адрес, а не отправлять сообщения боту. Однако могут использоваться оба варианта. Вам просто нужно изменить тип кнопки. И вот как это выглядит в Messenger – рис. 3.26.

Универсальный шаблон

Универсальный шаблон является одним из первых шаблонов, введенных Facebook для создания визуально привлекательных карт.

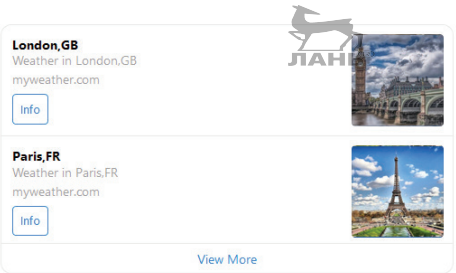


Рис. 3.26

Он также называется *каруселью*. Карусель позволяет отображать список элементов в виде горизонтального списка карт, где пользователь может прокручивать их и отвечать с помощью кнопок. На каждой карте может быть размещено не более трех кнопок. Чтобы создать универсальный шаблон, используйте показанный ниже формат сообщения:

```
«message»:{
    “attachment”:{
        “type”:”template”,
        “payload”:{
            “template_type”:”generic”,
            “elements”:[
                {
                    “title”:”London, GB”,
                    “image_url”:”https://myweather.com/london.png”,
                    “subtitle”:”What’s the weather in London, GB?”,
                    “default_action”: {
                        “type”: “web_url”,
                        “url”: “https://myweather.com/london”,
                        “messenger_extensions”: true,
                        “webview_height_ratio”: “tall”,
                        “fallback_url”: “https://myweather.com/”
                    },
                    “buttons”:[
                        {
                            “type”:”web_url”,
                            “url”:”https://myweather.com/london/weather”,
                            “title”:”Goto Website”
                        },{
```

```
                    "type":"postback",
                    "title":"Weather now",
                    "payload":"PAYLOAD_CURRENT_WEATHER_LONDON"
                }
            ]
        }
    ]
  }
 }
}
```

В предыдущем примере используется только одна карта. Добавьте в список больше элементов, чтобы создать дополнительные карты, возможно, по одной для каждого города. Обратите внимание, что карта имеет две кнопки. Одна из них представляет собой веб-представление, в котором нажатие кнопки **Goto Website** выведет вас на сайт с информацией о погоде в Лондоне во встроенном веб-браузере. И следующая – кнопка обратной передачи, которая отправит чат-боту в ответ сообщение PAYLOAD_CURRENT_WEATHER_LONDON.

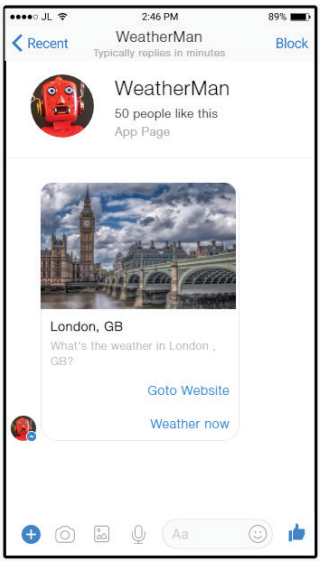


Рис. 3.27

Используя комбинацию этих и других элементов, вы можете создавать волшебные впечатления от общения для пользователя в

Facebook Messenger. API Facebook Messenger быстро развивается, добавляются новые элементы, чтобы помочь разработчикам создавать потрясающие диалоги с чат-ботом на платформе, поэтому следите за развитием событий.

Заключение

В этой главе мы рассмотрели, как создать чат-бота с использованием внешнего источника информации и интегрировать его в Facebook Messenger. Изучили, как создать бота и опубликовать на платформе Messenger, и эксклюзивные функции, такие как кнопки и шаблоны, которые помогут получить богатый и привлекательный опыт общения пользователя в Messenger.

В следующей главе мы перейдем от взаимодействия с помощью кнопок и рассмотрим, как обрабатывать пользовательские высказывания, чтобы сделать чат-бота более гибким для пользователей. Мы изучим инструментарий для распознавания естественного языка, называемый Dialogflow, и создадим бота-персонажа, который будет имитировать общение с Альбертом Эйнштейном.

Ссылки

- Документация по Facebook Messenger: **developers.facebook.com/docs/messenger-platform**.

Глава 4. Создание бота-персонажа

В предыдущей главе мы построили чат-бота, чтобы поговорить о погоде в мессенджере Facebook Messenger. Хотя мы и смогли пообщаться с ним, беседа сводилась главным образом к нажатию кнопок, доступных на платформе развертывания. Чтобы беседа с чат-ботом стала естественной, мы должны разрешить пользователям общаться с чат-ботом с использованием естественного языка. Это означает, что чат-бот должен уметь обрабатывать высказывания пользователей, распознавать их и подбирать наиболее подходящий ответ для пользователя. В этой главе мы рассмотрим, как можно обрабатывать ввод пользователя на естественном языке с помощью инструмента Google Dialogflow, используемого для создания чата, «понимающего» естественный язык. Мы развернем чат-бота на двух платформах: в виде веб-чата и в Facebook Messenger. Изучим концепции естественного языка, такие как цели, сущности и контексты, и обсудим, как разрабатывать и развертывать эффективные диалоговые инструменты с использованием этих идей. С пониманием этих концепций мы разработаем бота-персонажа по образу любимого физика всего мира – Альберта Эйнштейна.

В ходе этой главы вы:

- освоите основы Dialogflow;
- поймете концепции: цели, сущности и контексты;
- создадите агента Dialogflow;
- интегрируете агента Dialogflow в веб-чат и Facebook Messenger;
- расширите возможности агента с помощью вебхуков.

Знакомство с Dialogflow

Dialogflow (ранее API.AI) – платформа компании Google для создания диалоговых агентов. Это веб-платформа, к которой можно получить доступ из любого веб-браузера. Инструмент со временем развился из разработок для платформы Android,

начавшихся в ответ на технологию Apple Siri. Приложение называлось *SpeakToIt* и давало возможность общения, схожего с технологией Siri, на любом Android-устройстве. Технологии ИИ и естественного языка, которые использовались в приложении SpeakToIt, была открыты для разработчиков в виде инструмента API.AI в 2015 году.

API.AI позволял разработчикам создавать условия для ведения эффективного диалога, предоставляя им инструменты для выполнения двух видов задач: распознавать и генерировать высказывания на естественном языке и управлять диалогом. Хотя в отрасли существует множество инструментов, конкурирующих с Dialogflow (ранее API.AI), мы выбрали его, потому что он был одним из первых, достаточно зрелый и постоянно развивающийся, и его легко использовать и интегрировать в другие службы.

Настройка Dialogflow

Сначала создадим учетную запись разработчика на сайте Dialogflow.com.

1. Перейдите на сайт **dialogflow.com**.
2. Щелкните мышью по ссылке **GO TO CONSOLE** (Перейти в консоль) в правом верхнем углу.
3. Авторизуйтесь. Для этого потребуется использовать свою учетную запись Google.

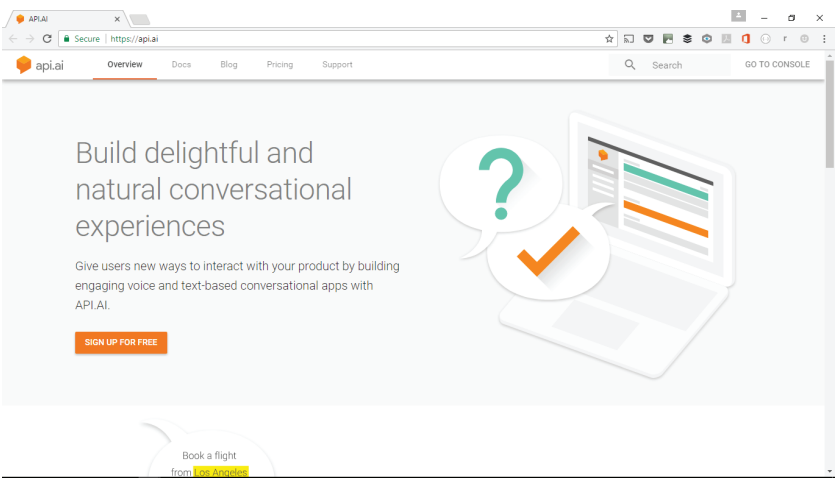


Рис. 4.1

Создание базового агента

Создадим нашего первого агента в Dialogflow. Начнем с базового, а затем добавим более сложные функции.

1. Чтобы создать нового агента, в раскрывающемся списке в левой части главной страницы (**dialogflow.com**) выберите пункт **Create new agent** (Создать нового агента).
2. Заполните форму в правой части страницы. Укажите имя и описание агента. Выберите часовой пояс и щелкните мышью по кнопке **CREATE** (Создать).
3. Откроется страница со списком целей. На ней уже есть две цели: **Default Fallback Intent** (Цель уточнения по умолчанию) и **Default Welcome Intent** (Цель приветствия по умолчанию).
4. Добавим собственную цель. Цель – это тот смысл, который пользователь или бот хочет передать с помощью высказываний или нажатия кнопок. Цель – символическое представление высказывания. Цели важны, так как существует много способов спросить то же самое. Процесс идентификации целей заключается в том, чтобы однозначно сопоставить многие способы с целью. Например, пользователь может спросить прогноз погоды в своем городе, используя следующие высказывания:

```
«расскажи про погоду в Лондоне»
«какая сейчас погода в Лондоне»
«погода в Лондоне»
«на улице солнечно?»
```

   В предыдущих высказываниях пользователь запрашивает сведения о погоде в Лондоне. В некоторых из этих высказываний он также упоминает время (то есть сейчас). В других время явно не упоминается. Первый шаг нашего алгоритма состоит в том, чтобы сопоставить эти многочисленные высказывания с одной целью: `request_weather_report`.

   Название цели (**Intent name**) соответствует цели пользователя. Поэтому присваивайте название с точки зрения пользователя. Добавим цель `user_greet`, которая соответствует акту приветствия пользователя чат-бота. Чтобы добавить цель, щелкните мышью по ссылке **Create intent** (Создать цель).
5. Вы увидите показанную ниже страницу, на которой сможете создать новую цель (рис. 4.2.):

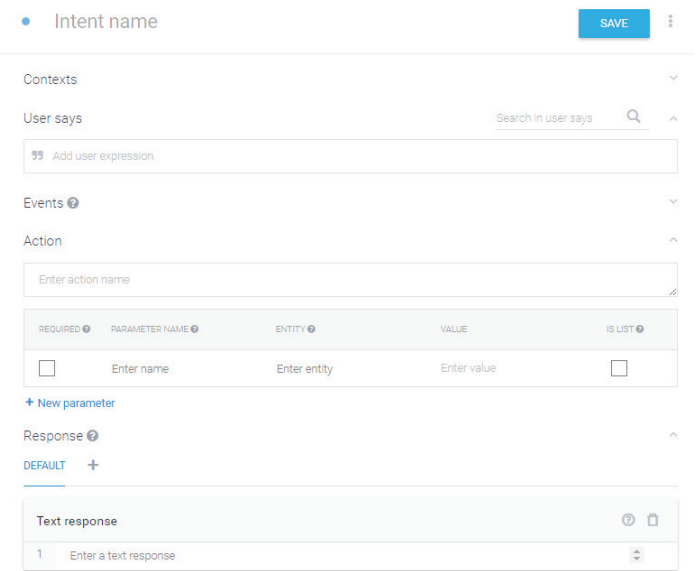


Рис. 4.2

Присвойте цели название (например, `user_greet`).

6. Добавьте пример пользовательских высказываний в текстовое поле **User says** (Речь пользователя). Это примеры высказываний, которые помогут агенту определить цели пользователя. Добавим несколько приветственных высказываний, которые пользователь может сказать нашему чат-боту:

```
hello
hello there
Hi there Albert
hello doctor
good day doctor
```

7. На данный момент проигнорируйте вкладку **Events** (События) и перейдите на вкладку **Action** (Действие). Добавьте имя, чтобы определить здесь цель системы (например, `bot_greet`, чтобы передать приветствие чат-бота пользователю).
8. На вкладке **Response** (Ответ) добавьте ответ бота пользователю. Это фактическое высказывание, которое бот отправит пользователю. Добавим показанное ниже высказывание в поле **Text**

response (Текст ответа). Вы можете добавить больше ответов, чтобы агент мог рандомно выбрать один, чтобы разговор был более разнообразным.

```
Hi there. I am Albert. Nice to meet you!
```

Вы также можете добавить до 10 дополнительных ответов, щелкнув мышью по ссылке **ADD MESSAGE CONTENT** (Добавить содержание сообщения).

9. Щелкните мышью по кнопке **SAVE** (Сохранить) в правом верхнем углу, чтобы сохранить цель. Вы создали свою первую цель для агента.
10. Проверьте его, используя эмулятор в правой части страницы. В поле **Try it now** (Попробовать сейчас) введите приветствие (`hello`) и нажмите клавишу **Enter** (рис. 4.3.).

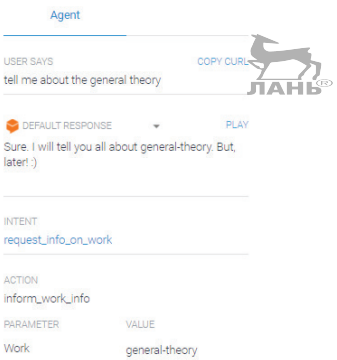


Рис. 4.3

Вы увидите, как чат-бот распознал введенное вами высказывание и ответил соответствующим образом.

11. Продолжайте и добавьте еще несколько целей, повторив шаги с 5 по 10. Чтобы создать новую цель, щелкните мышью по значку **+** в строке пункта **Intents** (Цели) в меню слева – рис. 4.4.

Подумайте, какую информацию пользователи будут сообщать чат-боту, и составьте список. Он и станет пользовательскими целями. Ниже приведен примерный список целей:

- `request_name`
- `request_birth_info`

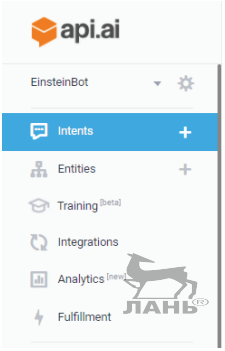


Рис. 4.4

- `request_parents_names`
- `request_first_job_experience`
- `request_info_on_hobbies`
- `request_info_patent_job`
- `request_info_lecturer_job_bern`

Конечно, этот список может быть бесконечным. Так что продолжайте и получайте удовольствие.

После того как вы указали достаточное количество фактов в указанном формате, вы можете протестировать чат-бота в эмуляторе, как описано в шаге 10.

Развертывание чат-бота

Теперь, когда у нас есть чат-бот, опубликуем его на платформе, где пользователи смогут его применять. Dialogflow позволяет размещать чат-бота (то есть агента) на многих платформах. Щелкните мышью по ссылке **Integrations** (Интеграции), чтобы просмотреть все доступные платформы, – рис. 4.5.

В этом разделе мы рассмотрим две платформы интеграции: веб-сайт и Facebook. К другим вернемся в следующих разделах – рис. 4.6.

Интеграция в веб-сайт

Эта возможность позволяет разместить созданного чат-бота на веб-сайте. Пользователь может взаимодействовать с чат-ботом на веб-сайте так же, как и с онлайн-чатом.

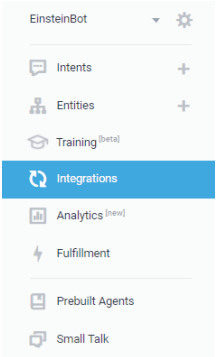


Рис. 4.5

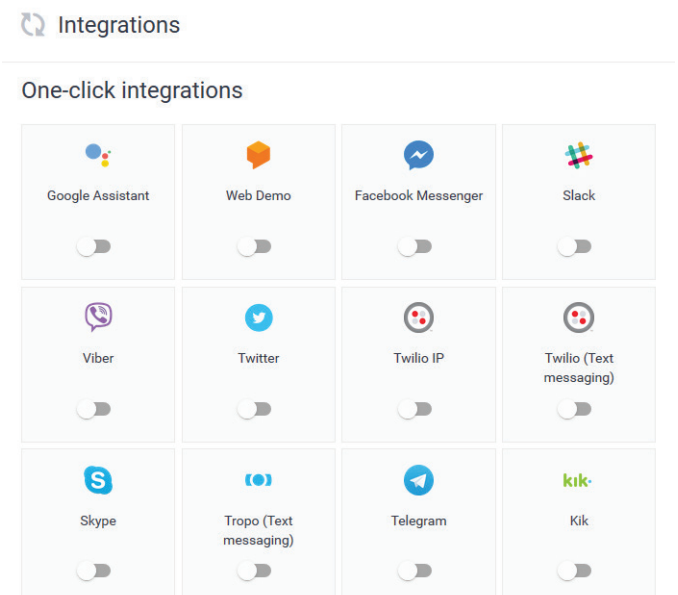


Рис. 4.6

1. На странице **Integrations** (Интеграции) найдите платформу **Web Demo** (Веб-демонстрация) и установите переключатель в активное положение.
2. Щелкните мышью по ссылке **Web Demo** (Веб-демонстрация), чтобы открыть окно с показанными ниже настройками, – рис. 4.7:

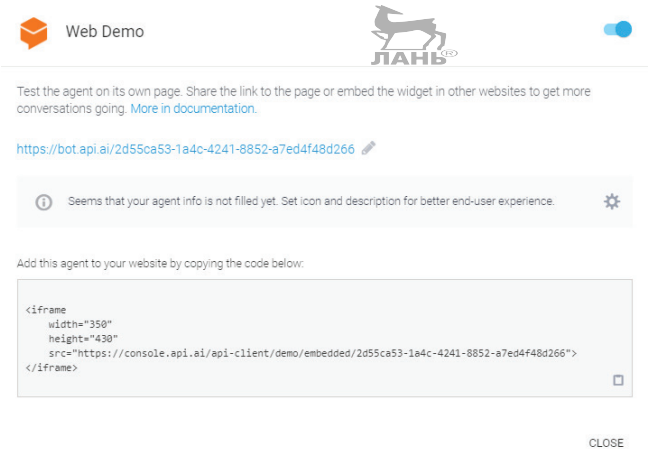


Рис. 4.7

3. Щелкните мышью по URL-адресу **bot.dialogflow.com**, чтобы открыть образец веб-страницы, где вы увидите чат-бота в виде виджета, встроенного в страницу. Попробуйте поговорить с ним – рис. 4.8.
   Вы можете поделиться ботом с друзьями по электронной почте или в социальных сетях, нажав кнопку **Email and Share** (Электронная почта и общий доступ).
4. Виджет чата также может быть встроен в любой веб-сайт с помощью элемента `iframe`, код которого указан в том же окне настроек. Скопируйте и вставьте его в код HTML-страницы, откройте ее в веб-браузере.

```
<iframe
    width="350"
    height="430"
    src="https://console.api.ai/api-client/demo/embedded/
        2d55ca53-1a4c-4241-8852-a7ed4f48d266">
</iframe>
```

Интеграция в Facebook Messenger

Чтобы опубликовать чат-бота Dialogflow в Facebook Messenger, понадобится страница Facebook. Также понадобится приложение

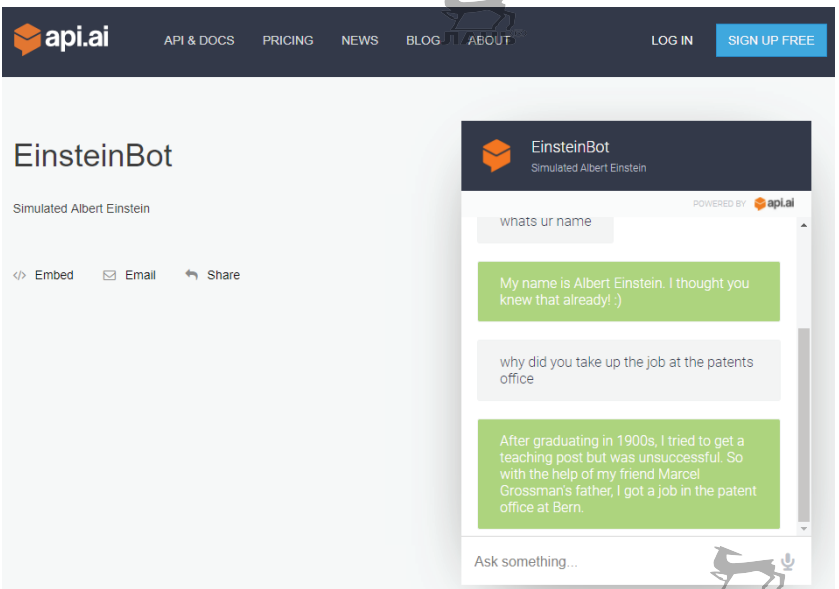


Рис. 4.8

Facebook Messenger, подписанное на эту страницу. Шаги по созданию Facebook-страницы и приложения Facebook Messenger подробно описаны в главе 3. Теперь обсудим дальнейшие шаги.

1. Создав приложение Facebook Messenger, получите его токен доступа к странице (**Page Access Token**). Вы можете найти его на вкладке **Settings** (Настройки) приложения Messenger.

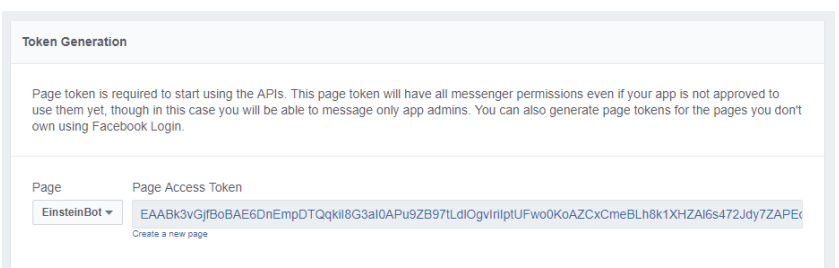


Рис. 4.9

2. На той же вкладке щелкните мышью по ссылке **Set up Webhooks** (Настроить вебхуки). Откроется диалоговое окно **New**

Page Subscription (Подписка на новую страницу). Оставьте его открытым в отдельной вкладке браузера.

3. На другой вкладке браузера, на странице **Integrations** (Интеграции) сайта Dialogflow щелкните мышью по ссылке **Facebook Messenger**.

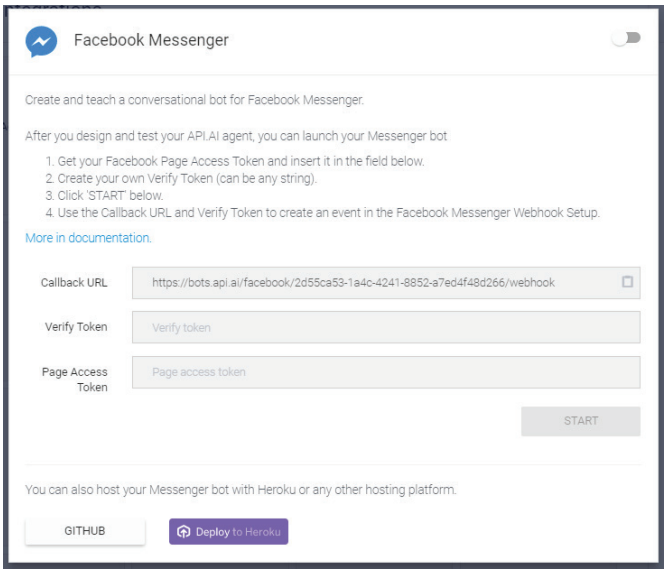


Рис. 4.10

4. Скопируйте URL-адрес в текстовое поле **Callback URL** (URL-адрес обратного вызова). Это URL-адрес агента Dialogflow для вызова из приложения Messenger. Вставьте адрес в поле **Callback URL** (URL-адрес обратного вызова) в окне **New Page Subscription** (Подписка на новую страницу) в приложении Facebook Messenger.
5. Введите токен проверки. Он может быть любым, мы наберем `iam-einstein-bot`.
6. Оформите подписки, установив флажки **messages** и **messaging_postbacks** в разделе **Subscription Fields** (Поля подписки). Подождите! Не нажимайте пока кнопку **Verify and Save** (Проверить и сохранить) – рис. 4.11.
7. На вкладке **dialogflow.com** браузера открыты настройки интеграции. Установите переключатель в верхнем правом углу в неактивное положение.

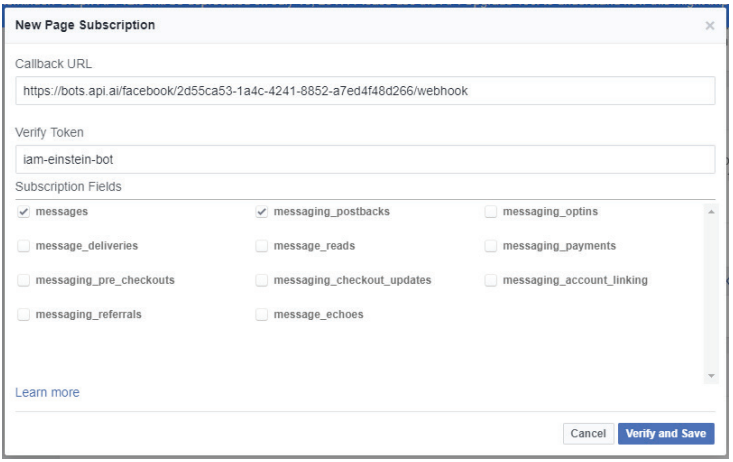


Рис. 4.11

8. Теперь вы можете изменить настройки. Введите токен проверки в поле **Verify Token** (Токен проверки). Он должен совпадать с введенным в настройках приложения Facebook Messenger на шаге 5.
9. Вставьте токен доступа к странице в поле **Page Access Token** (Токен доступа к странице) и щелкните мышью по кнопке **START** (Пуск).
10. Теперь вернитесь в приложение Facebook Messenger и щелкните мышью по кнопке **Verify and Save** (Проверить и сохранить). Произойдет подключение приложения к агенту (чат-боту).
11. Теперь на странице настроек Messenger Facebook в разделе **Webhooks** (Вебхуки) выберите нужную страницу Facebook, на которую приложение должно подписаться, и щелкните мышью по кнопке **Subscribe** (Подписаться).

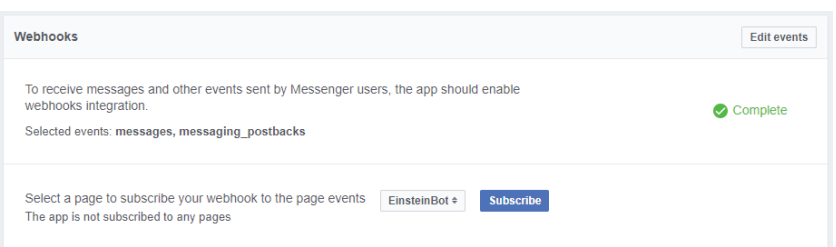


Рис. 4.12

Теперь вы можете открыть страницу Facebook, нажать кнопку **Send Message** (Отправить сообщение) и пообщаться с чат-ботом.

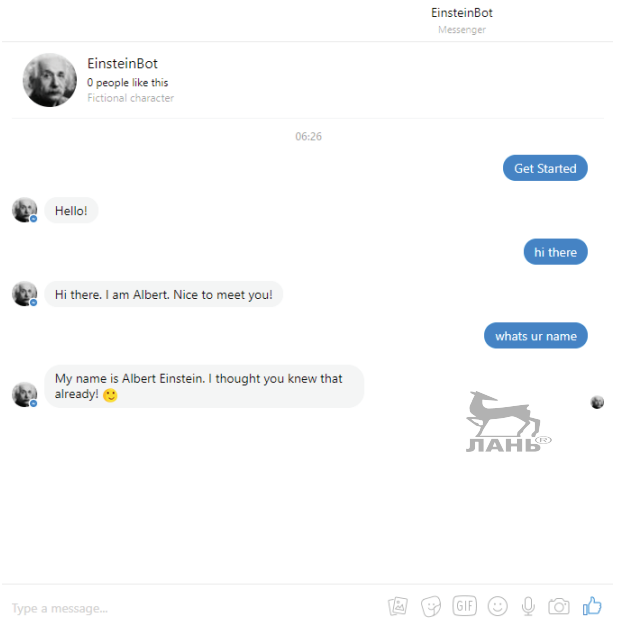


Рис. 4.13

Прекрасно! Вы успешно создали чат-бота на сайте Dialogflow.com и развернули его на двух платформах: веб-сайте и Facebook Messenger. В дополнение к этим платформам Dialogflow позволяет разместить вашего агента на нескольких популярных платформах обмена сообщениями, таких как Slack, Skype, Cisco Spark, Viber, Kik, Telegram и даже Twitter.

Контекст

Пока наш чат-бот отвечает на вопросы односложно. Пользователь задает вопрос, и чат-бот отвечает. А как же быть, если есть вопросы, требующие развернутых ответов? Что делать, если чат-бот должен запросить ответ на вопрос, прежде чем сам сможет ответить на него? Такие задачи требуют реализации диалогового контекста.

Рассмотрим следующий диалог:

```
Пользователь: Почему ты устроился на работу в патентное бюро?
Бот: После окончания школы я попытался устроиться в школу, но безуспешно.
Поэтому с помощью отца моего друга Марселя Гроссмана я получил работу в
патентном бюро в Берне.
Пользователь: Когда ты устроился на работу?
```

Второй вопрос пользователя касается первого вопроса и ответа от чат-бота. Это не самостоятельный вопрос. Слово `работа` могло означать любую работу, которую Эйнштейн мог иметь при жизни, но в соответствии с контекстом ясно, о чем говорит пользователь. Поэтому чат-бот должен учитывать это, чтобы распознать, что на самом деле имеет в виду пользователь.

На вкладке **Contexts** (Контексты) сайта Dialogflow.com при создании целей мы можем позволить агентам вести запись контекста диалогов. Каждая цель должна иметь контексты ввода и вывода. Первый определяет контекст, в котором учитывается цель. Контекст вывода определяет сопоставление цели и ее успешное выполнение в ответном высказывании.

Чтобы успешно поддерживать диалог, агенту необходимо записать контекстную информацию в контексте вывода для цели `request_info_patent_job`, на который отвечает вопрос о работе патентного бюро Эйнштейна. Затем это послужит исходным контекстом для последующих вопросов, таких как «Когда вы получили работу?» или «Как оплачивалась работа?» и многое другое.

Вот как вы можете это сделать:

1. Откройте цель `request_info_patent_job`. Перейдите на вкладку **Contexts** (Контексты). Вы увидите развернутое содержимое вкладки, как показано на рисунке ниже.

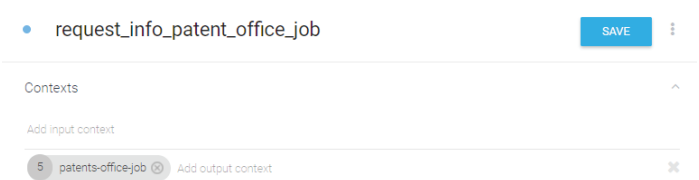


Рис. 4.14

2. Введите имя контекстного элемента, который агент должен сохранить в поле контекста вывода. Поскольку агенту нужно знать, что вопрос пользователя касается работы в патентном

бюро, введите название контекстного элемента (то есть `patents-office-job`) в качестве контекста вывода.

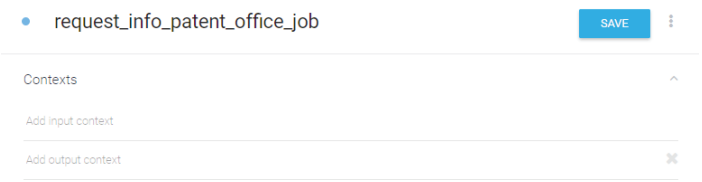


Рис. 4.15

Для каждого контекстного элемента можно ограничить продолжительность его существования. По умолчанию установлено значение 5. При каждом входящем пользовательском высказывании значение уменьшается на 1. Когда оно достигает нуля, элемент автоматически удаляется из контекста диалога.

3. Мы указали выходной контекст для цели. Поэтому если агент распознает эту цель в высказывании пользователя, он будет отвечать соответствующим образом, а также установит выходной контекст. Попробуем в эмуляторе:

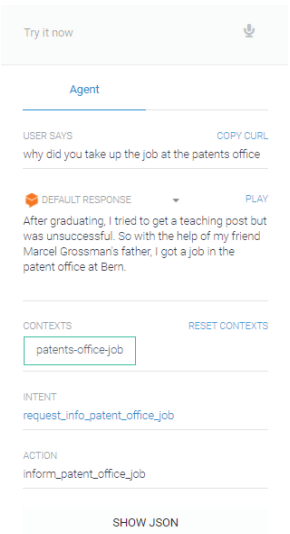


Рис. 4.16

На предыдущем рисунке показано, что теперь контекст изменен с пустого значения на `patents-office-job`. Каждое новое высказывание пользователя теперь будет обработано в этом контексте.

4. Чтобы использовать контекст, контекстные элементы должны быть указаны как входные в последующих целях пользователя. Протестируем, создав новый запрос пользователя `request_info_poffice_joining`. Эта цель будет распознана, если пользователь спросит, когда вы начали свою работу в патентном бюро или что-то подобное. Но даже если пользователь спрашивает о дате начала работы и не упоминает о патентном бюро, нужно уметь распознать это в процессе диалога о работе. Чтобы сделать это, упомянем, о чем идет речь во входном контексте.

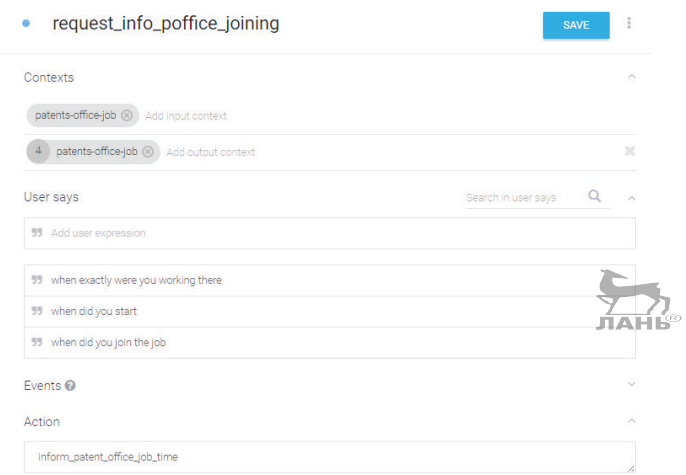


Рис. 4.17

На предыдущем рисунке видно, как упоминается элемент `patents-office-job` в поле входных контекстов. Таким образом, высказывание, например, «когда вы начали» в контексте `patents-office-job` будет распознано как `request_info_poffice_joining`. То же высказывание в других контекстах не будет обработано как цель `request_info_poffice_joining` – рис. 4.18.

На предыдущем рисунке показано, как одно и то же высказывание в разных точках диалога дает разные результаты, потому что агент теперь обрабатывает входные данные от пользователя в соответствии с контекстом.

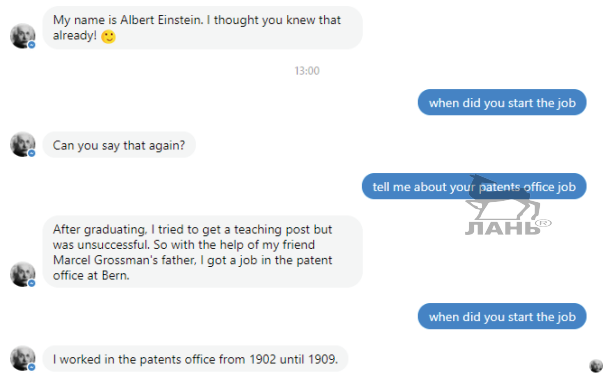


Рис. 4.18

Продолжительность существования элемента в контексте вывода может быть увеличена или уменьшена. Если контекстный элемент больше не нужен, его можно удалить, установив продолжительность существования равной нулю. Это может быть полезным, если пользователь благодарит бота после завершения задачи. В таком случае чат-бот может сбросить диалоговый контекст.

При необходимости можно добавлять дополнительные контекстные элементы как в поля ввода, так и вывода. Это может быть полезно при конструировании более сложных диалогов. Хотя использование контекстных элементов в процессе разговора и хорошая идея, я рекомендую использовать глобальные (`global`) контекстные элементы и сохранять и использовать их в виде глобальных параметров. Такие элементы будут доступны для всех целей на входе и, кроме того, переданы в выходной контекст.

Сущности

Сущности добавляют еще один уровень сложности в диалоги на естественном языке, благодаря которому чат-боты становятся более мощными и гибкими. Любой объект, представляющий интерес для беседы чат-бота с пользователем, может рассматриваться в виде **сущности**. Например, в информации о погоде в чат-боте города и малые города могут рассматриваться как сущности. Другими представляющими интерес сущностями могут быть любые данные, интересные пользователю: температура, вероятность дождя, скорость ветра и т. д. В отношении температуры единицы измерения в градусах

Фаренгейта и градусах Цельсия также могут рассматриваться как сущности.

Сущности могут быть представлены как совокупности имен и значений параметров. Например, в случае с погодой вы можете добавить запрос о погоде в зависимости от города следующим образом:

```
intent = request_weather
city_name = london_gb
```

Предыдущий формат эквивалентен тому, что пользователь спрашивает: «Какая погода в Лондоне?» Поэтому, вместо того чтобы использовать одну цель для каждого города (например, `request_weather_london`), мы можем лучшим образом организовать пользовательские запросы с точки зрения целей и сущностей. Это также упрощает распознавание целей пользователями.

Взглянем на несколько других примеров:

```
intent = change_metric
metric = celsius
```

В предыдущем примере представлены высказывания, такие как «Сколько градусов Цельсия?» и «Сообщите в градусах по шкале Цельсия, пожалуйста».

В случае бота Эйнштейна мы можем представить его основные работы, такие как специальная теория относительности, общая теория относительности и броуновское движение как сущности. Такие сущности могут использоваться в соответствующих целях, таких как `request_info_on_work` и `request_publications`. Возьмем, к примеру, следующую спецификацию целей:

```
intent = request_info_on_work
work = special_theory_of_relativity
```

Предыдущая цель может означать, что пользователь говорит: «Расскажите мне о своей работе над специальной теорией относительности» или что-то подобное.

Чтобы использовать сущности, нужно перечислить их в агенте. Для этого выполните следующие действия.

1. Щелкните мышью по ссылке **Entities** (Сущности) в главном меню. Она находится ниже пункта **Intents** (Цели), который использовался для создания пользовательских целей.

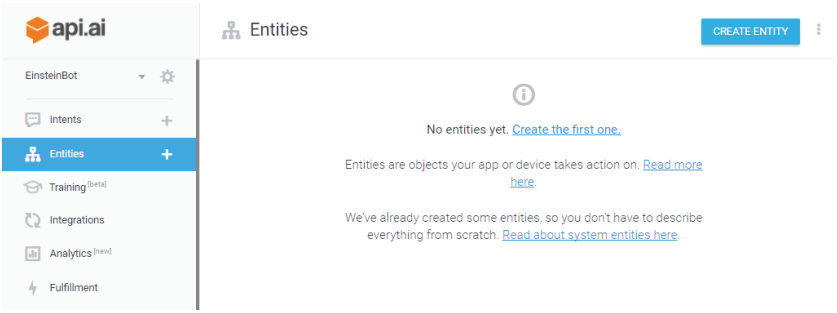


Рис. 4.19

2. Щелкните мышью по ссылке **Create entity** (Создать сущность).
3. Задайте название сущности. Для погодного бота подходящими сущностями могут быть **London**, **Paris** или **New York**, и их можно сгруппировать под названием **City**. В случае бота Эйнштейна назовем их словом **Work**, поскольку сущности фактически относятся к его работе.
4. Ниже находится таблица с двумя столбцами. Слева нужно указать контрольное значение сущности, а справа – ее синонимы. Чтобы было понятно, вернемся к городам. Каждый город имеет контрольное значение (например, `moscow-rus`), чтобы в системе он определялся однозначным образом. Каждое контрольное значение может быть выражено в терминах, словах или фразах на естественном языке. В этом случае `moscow-rus` можно назвать «город на семи холмах», «город Москва», «златоглавая» и т. д. Они должны быть перечислены как синонимы. Для нашего бота добавим три работы Эйнштейна: специальную теорию относительности, общую теорию относительности и теорию броуновского движения. Для каждой из этих работ добавим также синонимы – рис. 4.20.
5. Создав сущности, применим их, создав цель, которая будет их использовать. Создадим цель с именем `request_info_on_work` – рис. 4.21.
   Обратите внимание, как на предыдущем рисунке выделяются слова **special theory ofrelativity**. Это делается автоматически с помощью редактора Dialogflow на основе списка сущностей,

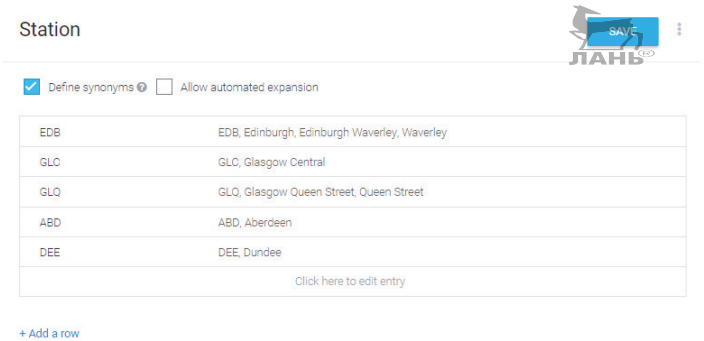


Рис. 4.20

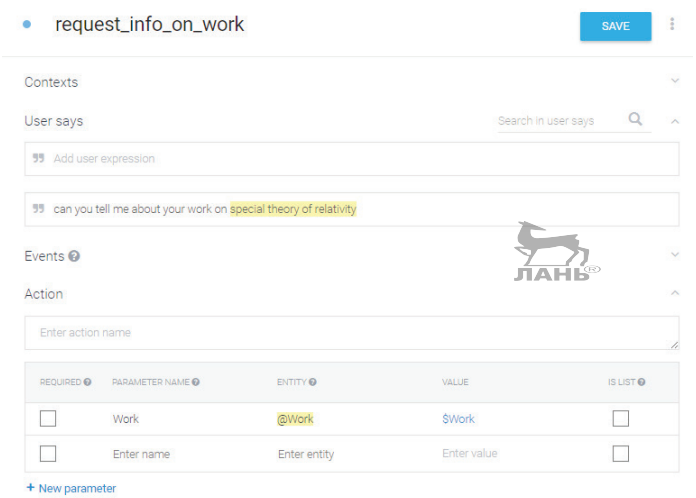


Рис. 4.21

который мы создали. Он идентифицирует сущности в примерах высказывания, которые мы предоставляем для каждой цели. Затем он помещает эти идентифицированные сущности в список параметров – рис. 4.22.

6. Протестируйте бота в эмуляторе и понаблюдайте за выводом агента. Как показано на предыдущем рисунке, вы увидите, что агент принял название и создал параметр с соответствующим именем (то есть `Work`) и значением (то есть `general-theory`). В поле

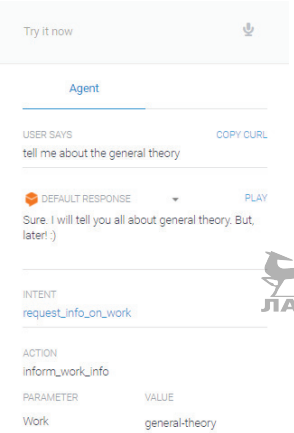


Рис. 4.22

значения вы найдете контрольное значение сущности независимо от того, какой синоним использовался в высказывании.

Теперь, извлекая параметры и значения, выясним, как их использовать в ответном высказывании. Параметр может быть указан в ответных высказываниях с использованием нотации `$`. В этом случае контрольное значение параметра `work` может быть получено с помощью кода `$work`. Примените ответное высказывание `Sure. I will tell you all about $Work. But later! :)` и посмотрите, что произойдет, – рис. 4.23.

7. Поскольку контрольные значения содержат дефис, реально они не соответствуют высказываниям. Поэтому мы можем

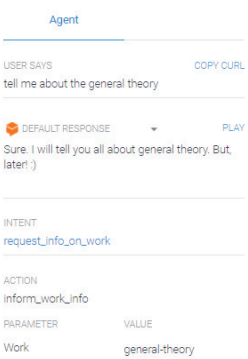


Рис. 4.23

использовать синоним, применяемый пользователем. Для этого измените код `$Work` на `$Work.original`. Сохраните цель и протестируйте бота в эмуляторе – рис. 4.24.

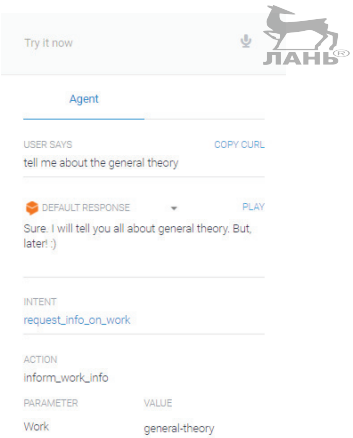


Рис. 4.24

8. Попробуйте аналогичное, но частично определенное высказывание, например `tell me about your work`. Оно отражает цель, но не имеет требуемого параметра. Как агент справится с этим высказыванием? Проверив, вы увидите, что цель была распознана, но ответ пуст. Это связано с тем, что агент не может найти значение `$Work.original`.
   Чтобы убедиться, что агент получает значение параметра `Work`, вы должны сделать параметр обязательным. Для этого установите флажок **REQUIRED** (Требуется). Щелкните мышью по ссылке **Define prompts** (Определить подсказки), чтобы предоставить агенту подсказки, которые он будет выводить, если пользователь забыл указать значение параметра в своем высказывании, – рис. 4.25.
   Щелкните мышью по ссылке **SAVE** (Сохранить), после того как вы определили сущности, пометили их требующимися и определили подсказки, – рис. 4.26.
9. Протестируйте бота в эмуляторе еще раз с высказыванием `tell me about your work`. Вы увидите, что агент просит вас подсказать ответ – назвать работу. После этого агент ответит соответствующим образом, – рис. 4.27.
10. Сущности могут храниться в контексте и использоваться во всех высказываниях. Чтобы сделать это, добавим контекстный

Рис. 4.25

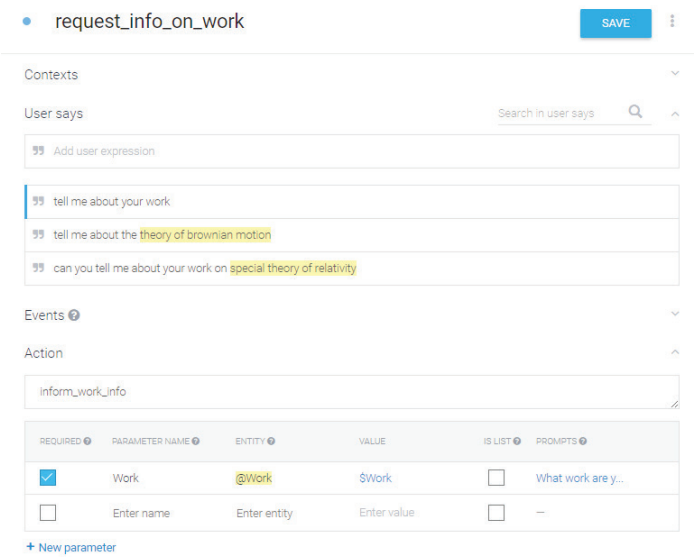


Рис. 4.26

элемент `talk_about_work` в выходной контекст цели `request_info_on_work`. Мы также добавим еще один `request_info_on_work_now` с помощью значения `talk_about_work` в качестве входного контекста – рис. 4.28.

Сущности передаются от одной цели другой через контекстные элементы. Мы можем получить доступ к сущностям из предыдущих сеансов диалога, используя символ **#**. Например, обратите внимание, как мы заполняем значение параметра `Work`, используя значение

Try it now

Agent

USER SAYS COPY CURL
tell me about your work

DEFAULT RESPONSE PLAY
What work are you referring to?

CONTEXTS RESET CONTEXTS
request_info_on_work_dialog_params_work
request_info_on_work_dialog_context
25cc6173-236c-491f-9b4f-66316776170d_id_dialog_context

INTENT
request_info_on_work

ACTION
inform_work_info

PARAMETER VALUE
Work

Try it now

Agent

USER SAYS COPY CURL
general theory

DEFAULT RESPONSE PLAY
Sure. I will tell you all about general theory. But, later! :)

INTENT
request_info_on_work

ACTION
inform_work_info

PARAMETER VALUE
Work general-theory

Рис. 4.27

request_info_on_work_now SAVE

talk-about-work Add input context

5 talk-about-work Add output context

User says Search in user says

Add user expression

i wanna know now

not later

tell me now

no tell me now

Events

Action

inform_work_info_now

REQUIRED	PARAMETER NAME	ENTITY	VALUE	IS LIST
☐	Work	Enter entity	#talk-about-work.Work	☐
☐	Enter name	Enter entity	Enter value	☐

Рис. 4.28

`#talk-about-work`. Затем это можно использовать в ответном выражении: `$Work` (например, `Alright. I will tell you about $Work`). – рис. 4.29.

Как использовать сущности в процессе выполнения расширенной обработки на стороне сервера, мы рассмотрим в конце главы.

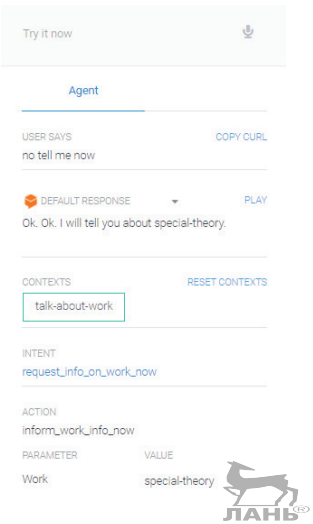


Рис. 4.29

Системные сущности

В дополнение к сущностям, которые мы создаем (то есть *сущности разработчика*), существуют также предопределенные *системные сущности*. Они предназначены для объектов, общих для большинства доменов и диалоговых задач. Такие понятия, как даты, личные имена, названия городов, названия стран, адреса электронной почты и номера телефонов, могут быть распознаны в пользовательских высказываниях с использованием системных сущностей – рис. 4.30.

На предыдущем рисунке показано, как вы можете получить к ним доступ, используя квалификатор `@sys` при предоставлении выборочных высказываний. Например, вы хотите получить адрес электронной почты пользователя. Для этого вы можете настроить цель пользователя в контексте диалога, например `inform_user_email`. Этот экземпляр пользователя должен иметь примерные высказывания, такие как «bluewhale@bmail.com» или «это bluewhale@bmail.com», где пользователь предоставляет чат-боту свой адрес электронной почты. Вы заметите, что адрес электронной почты

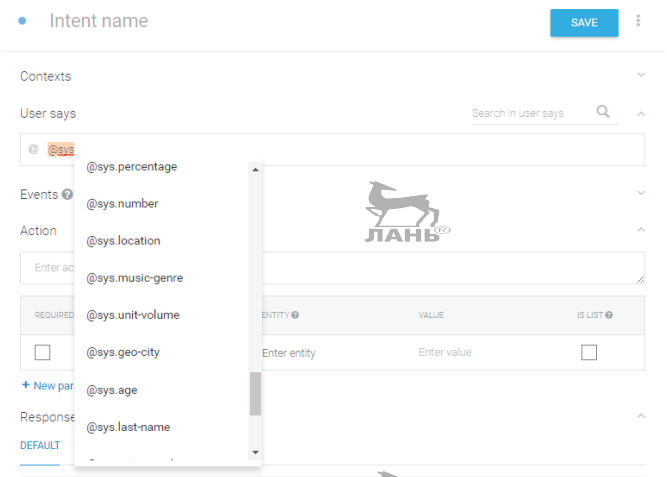


Рис. 4.30

распознается как `@sys.email`. Если он не распознается автоматически, вы можете указать, что это адрес электронной почты, выбрав его и указав `@sys.email`. Так агент поймет, что все высказывания с действительными адресами электронной почты должны быть распознаны как цель `inform_user_email`.

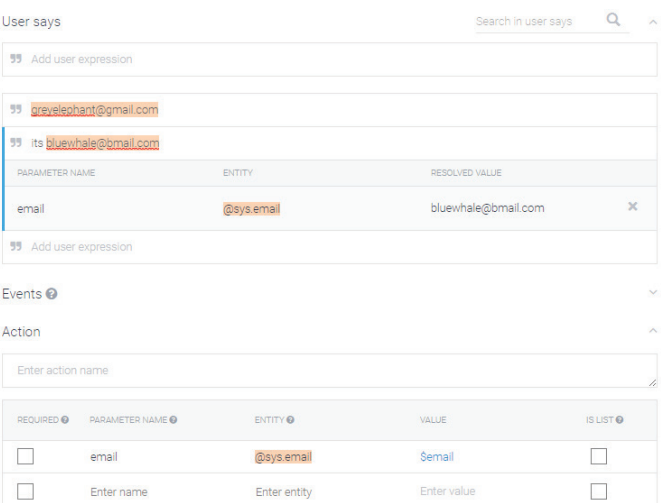


Рис. 4.31

После распознавания сущность будет передана как параметр, который будет добавлен к цели. Системные сущности, такие как `@sys.geo-city` и `@sys.geo-country`, могут использоваться для распознавания имен городов и названий стран в пользовательских высказываниях без указания всех имен в качестве целей разработчика.

Чтобы распознавать составные элементы в выражениях на естественном языке, такие как упоминание величин (вес, длина и т. д.), мы можем использовать составные системные сущности. `@sys.unit-weight`: вес единицы может использоваться для распознавания высказываний, таких как *10 кг*, *15 кило* и *20 килограммов*. Для дальнейшего действия они будут разобраны в следующем параметре:

```
{«amount»:10,»unit»:»kg»}
```

Сущности не могут быть непосредственно указаны в выборочных высказываниях. Например, `its@sys.email` не может быть указан как пример высказывания. Вместо этого он должен быть представлен как шаблон. Чтобы предоставить примеры шаблонов для целей, измените двойные кавычки на знак `@`.

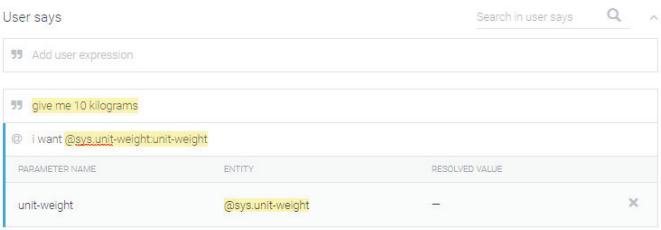


Рис. 4.32

Обратите внимание, что на предыдущем рисунке есть примерное высказывание (т. е. **give me 10 kilograms**) и образец шаблона **i want @sys.unit-weight:unit-weight**. Изучите системные сущности, чтобы узнать, как вы можете использовать их в своем чат-боте.

Совет. Более подробную информацию о системных сущностях можно найти на странице **dialogflow.com/docs/reference/system-entities**.

Развернутые ответы

В дополнение к текстовым ответам вы также можете предоставить развернутые типы сообщений, такие как карты, карусели и такие,

что мы использовали в главе 2 при создании погодного бота для платформы Facebook Messenger. Тип ответа, который мы можем добавить к агенту, зависит от его интеграции. Каждая интеграция предоставляет другой набор типов ответов.

Поскольку мы интегрировали в Facebook Messenger бота Эйнштейна, мы можем добавить контент сообщений, специфичный для платформы. Для этого выполните следующие действия.

1. Выберите любую цель пользователя, для которой вы хотите добавить ответы расширенных форматов.
2. Прокрутите страницу вниз до вкладки **Response** (Ответ).

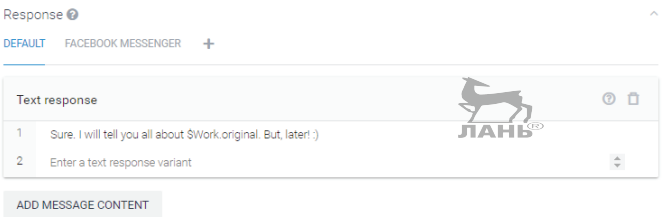


Рис. 4.33

3. Наряду с **Default** (По умолчанию) будет видна вкладка настроенной интеграции (например, **Facebook Messenger**). Если вы не видите такую вкладку, вы можете добавить интеграцию, щелкнув мышью по значку +.
4. Для каждой интеграции вы увидите следующее:

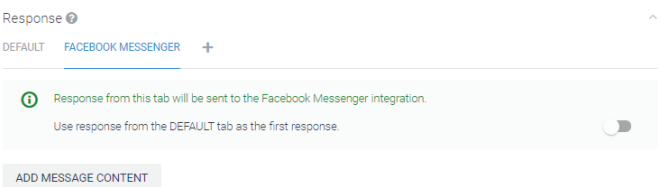


Рис. 4.34

5. Если вы хотите, чтобы ответ по умолчанию отображался первым на платформе интеграции, установите этот переключатель в положение.
6. Щелкните мышью по кнопке **Add message content** (Добавить содержимое сообщения) и выберите тип ответа. В случае интеграции

в Facebook Messenger вы найдете типы ответов, такие как текстовое сообщение, изображение, карта, быстрые ответы и пользовательская полезная нагрузка.

7. Введите информацию о формате ответа.

Ниже приведен пример ответа в виде карты для интеграции в Facebook Messenger.

И вот как это выглядит в Facebook Messenger:

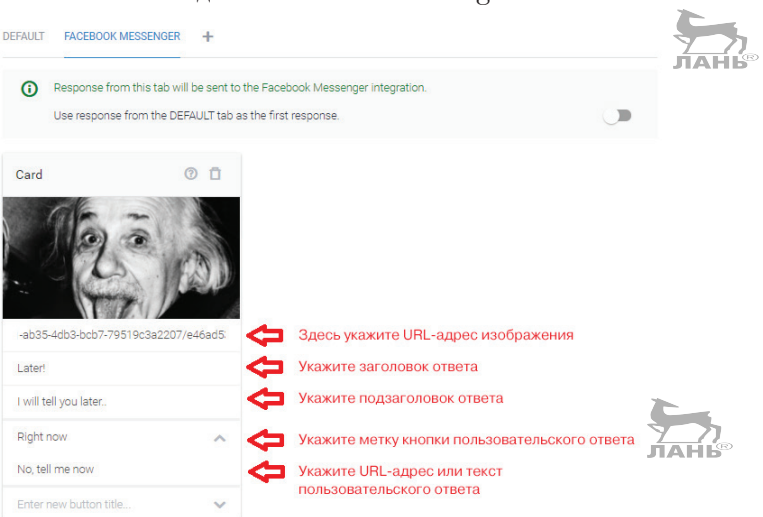


Рис. 4.35

Нажав кнопку **Right now**, вы отправите агенту сообщение **No, tell me now** в виде пользовательского ответа – рис. 4.36.

Теперь, когда у вас есть представление о том, как сделать ваши ответы более насыщенными, продолжайте экспериментировать с различными форматами, доступными для интеграции.

Импорт и экспорт агентов

Все агенты Dialogflow могут быть экспортированы и сохранены. Эта возможность также позволяет переместить агента из одной учетной записи Dialogflow в другую.

Экспорт, восстановление и импорт агентов

Вот как вы можете экспортировать агента:

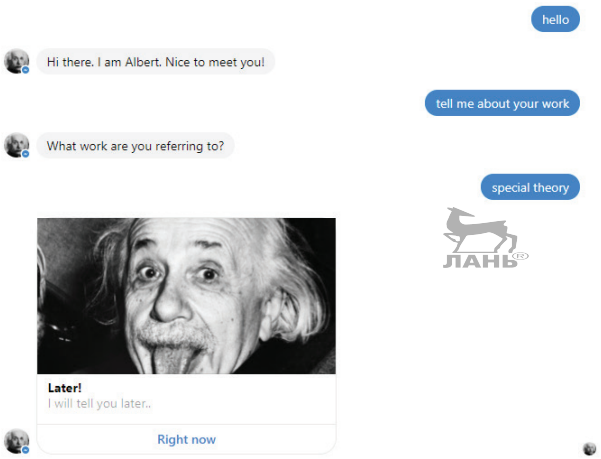


Рис. 4.36

1. Щелкните мышью по значку **Settings** (Настройки) рядом с именем агента в меню слева.
2. Перейдите на вкладку **Export and Import** (Экспорт и импорт).

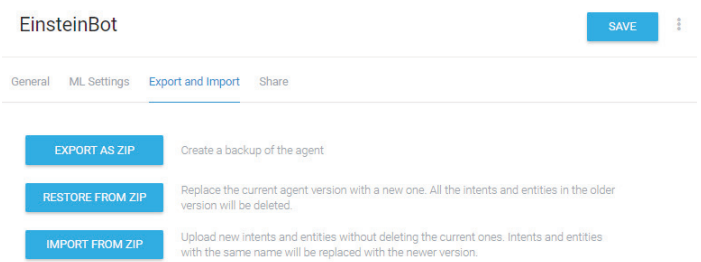


Рис. 4.37

3. Щелкните мышью по кнопке **EXPORT AS ZIP** (Экспортировать в виде ZIP-архива).

Загруженный ZIP-файл будет организован с точки зрения настроек агента, сущностей и целей в формате файла JSON. Ниже приведен JSON-код настроек агента для нашего бота Эйнштейна.

```
{
  "description": "Simulated Albert Einstein",
```

```
  "language": "en",
  "googleAssistant": {
    "googleAssistantCompatible": false,
    "project": "einsteinbot-dce04",
    "welcomeIntentSignInRequired": false,
    "startIntents": [],
    "systemIntents": [],
    "endIntentIds": [],
    "oAuthLinking": {
      "required": false,
      "grantType": "AUTH_CODE_GRANT"
    },
    "voiceType": "MALE_1",
    "capabilities": [],
    "protocolVersion": "V2"
  },
  "defaultTimezone": "Africa/Casablanca",
  "webhook": {
    "available": false,
    "useForDomains": false
  },
  "isPrivate": true,
  "customClassifierMode": "use.after",
  «mlMinConfidence»: 0.3
}
```

Вы также можете восстановить агента из файла резервной копии, нажав кнопку **Restore from Zip** (Восстановить из ZIP-архива). Все цели и сущности будут удалены и восстановлены из ZIP-файла.

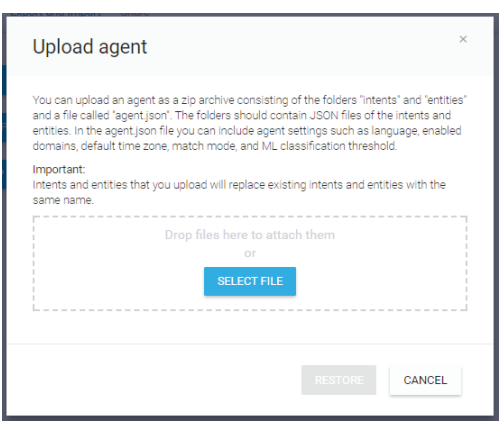


Рис. 4.38

В случае если вы хотите расширить своего агента из ZIP-файла, это можно сделать с помощью кнопки **Import from Zip** (Импортировать из ZIP-архива). Данная возможность позволяет добавлять дополнительные цели и сущности к существующему агенту. Все сущности и цели в агенте с теми же именами, что и в файле, будут заменены на те, которые находятся в файле ZIP.

Экспорт и импорт целей и сущностей

Цели и сущности могут экспортироваться независимо друг от друга и по очереди, без необходимости полностью экспортировать агента. Это можно сделать, щелкнув мышью по значку **cloud_download** напротив каждой цели и сущности.

Рис. 4.39

Вот пример цели бота Эйнштейна:

```
{
  "userSays": [
    {
      "id": "aa492d7b-6119-4d4c-a626-03aea6868882",
      "data": [
        {
          "text": "what was your job at the patent office"
        }
      ],
      "isTemplate": false,
      "count": 0
    },
    {
      "id": "f8a43750-c215-4750-aac8-1a0f977611d8",
      "data": [
        {
          "text": "why did you take up the job at the patent office"
        }
      ],
      "isTemplate": false,
      "count": 0
    },
    {
```

```
      "id": "2b48f3e6-7949-4cf6-b25c-ebb4691e4812",
      "data": [
        {
          "text": "what did you do at the patent office"
        }
      ],
      "isTemplate": false,
      "count": 0
    }
  ],
    "id": "38b83408-16ae-4b05-b841-187b7205e116",
    "name": "request_info_patent_office_job",
    "auto": true,
    "contexts": [],
    "responses": [
      {
        "resetContexts": false,
        "action": "inform_patent_office_job",
        "affectedContexts": [
        {
          "name": "patents-office-job",
          "parameters": {},
          "lifespan": 5
        }
      ],
      "parameters": [],
      "messages": [
      {
      "type": 0,
      "speech": "After graduating, I tried to get a teaching post
      but was unsuccessful. So with the help of my friend Marcel
      Grossman\u0027s father, I got a job in the patent office at Bern."
      }
    ]
  }
  ],
  "priority": 500000,
  "webhookUsed": false,
  "webhookForSlotFilling": false,
  «fallbackIntent»: false,
  «events»: []
}
```


Цели также могут быть загружены с использованием спецификаций в формате JSON, показанных ранее. Для этого создайте

JSON-файл, щелкните мышью по ссылке в виде трех точек на вкладке **Intents** (Цели) и выберите пункт **Upload Intent** (Загрузить цель).

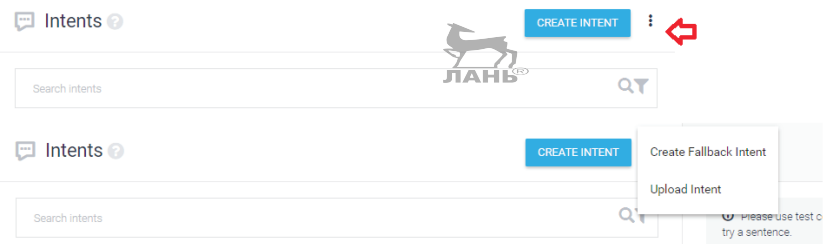


Рис. 4.40

При появлении запроса предоставьте файл цели в формате JSON, как показано ниже.

Рис. 4.41

Ниже показан файл сущности из бота Эйнштейна (из сущности `Work`). Имя сущности – `Work`, и каждая сущность вводится в виде записи с контрольными значениями и синонимами.

```
{
  "id": "9ed80101-6ecc-4006-bf7f-7c49bcbc03ef",
  "name": "Work",
  "isOverridable": true,
  "entries": [
    {
    "value": "special-theory",
    "synonyms": [
      "special theory of relativity",
      "special theory"
    ]
    },
    {
    "value": "general-theory",
    "synonyms": [
```

```
      "general theory",
      "general theory of relativity"
    ]
    },
    {
    "value": "brownian-movement-theory",
    "synonyms": [
      "brownian motion",
      "theory of brownian motion",
      "theory of brownian movement",
      "brownian movement",
      "brownian movement theory"
    ]
    }
  ],
  "isEnum": false,
  «automatedExpansion»: false
}
```

Новые сущности могут быть добавлены к агенту, если их создать в предыдущем формате и загрузить. Такой же подход можно использовать для редактирования существующих сущностей.

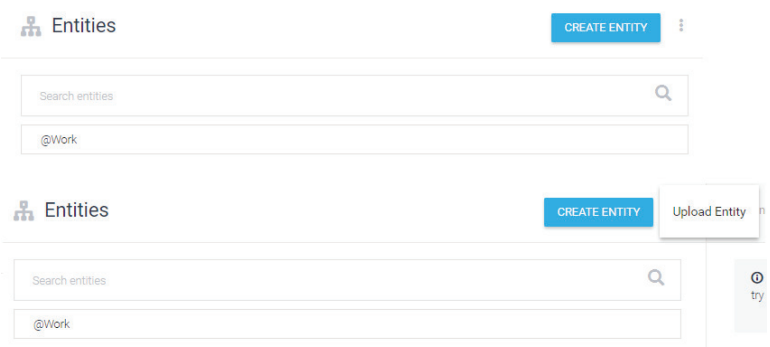


Рис. 4.42

Осуществления

Хотя вы можете создавать полноценных агентов на сайте Dialogflow, вам все равно нужно использовать внешние источники для серверных задач, которые должен выполнять чат-бот. Задачи на стороне

сервера – это задачи, которые чат-бот будет выполнять за кадром во время диалога с пользователем. Они могут быть различны: запись диалога в CRM, идентификация команды, обращение к базе данных за информацией, запрашиваемой пользователем. Все это называется **исполнением**.

Примером выполнения в чат-боте планирования полетов является запрос к базе данных для проверки и получения доступных рейсов на определенную дату, определения аэропортов вылета и прилета. Эта задача будет выполнена веб-приложением, которое может вызываться чат-ботом, когда у него есть необходимая информация для выполнения запроса. Другой пример – это когда потребуется сделать заказ.

Мы создадим задачу для бота Эйнштейна, чтобы поэкспериментировать. Создадим модуль вычислений на основе всемирно известного уравнения Эйнштейна $E = mc^2$.

Это уравнение вычисляет количество энергии, которое будет генерироваться, если масса системы полностью исчезнет. Чтобы вычислить это, нам нужно узнать массу системы m от пользователя. Мы передадим значение в веб-приложение, где вычисляется количество энергии, вырабатываемой для указанной массы m. Взглянем на пример.

```
Пользователь: Привет. Можешь сказать, сколько энергии генерирует атом?
Бот: Да, конечно. Сколько он весит?
Пользователь: Предположим, что 2 грамма.
Бот: Хорошо. Система весом 2 грамма преобразуется в 179 751 036 МДж энергии.
```

Чтобы добавить к коду бота вебхук, сделайте следующее:

1. Создайте веб-приложение, которое может принимать параметры, выполнять серверную задачу и отвечать высказываниями.
2. Разместите его в облаке.
3. Сконфигурируйте вебхук для агента.
4. Включите вебхуки в целях, где они требуются.

Чтобы создать и разместить веб-приложение, выполните следующие действия.

1. В консоли создайте каталог для хранения этого проекта. В этом каталоге мы создадим следующие три файла.
   - index.js
   - package.json
   - Procfile

index.js – это сценарий Node.js, которая действует как веб-приложение для получения параметров из чат-бота, выполняет бэкенд-операцию и отправляет обратно высказывание:

```
const express = require('express')
const bodyParser = require('body-parser')
const request = require('request')

const app = express()

app.set('port', (process.env.PORT || 5000))

// Обрабатываем application/x-www-form-urlencoded
app.use(bodyParser.urlencoded({extended: false}))

// Обрабатываем application/json
app.use(bodyParser.json())

app.use(express.static('public'))

// Индексирование пути
app.get('/', function (req, res) {
    res.send('Hello world, I am EinsteinBot!.')
})

app.post('/emc2/', function (req, res) {
    console.log(JSON.stringify(req.body));
    var weight = req.body.result.parameters.weight;
    var m = weight.amount;
    var weight_unit = weight.unit;
    // преобразование веса в кг
    if (weight_unit == 'g'){
        m = m/1000.0;
    }
    var c2 = 9 * 10^16; //in m^2/s^2
    var e = m * c2;
    res.setHeader('Content-Type', 'application/json');
    var botSpeech = "Energy that the system can create is " + e
    + " Joules.";
    out = {speech: botSpeech,
            displayText: botSpeech,
            data: null};
    var outString = JSON.stringify(out);
    console.log('Out:' + outString);
    res.send(outString);
```

```
})

// обращение к серверу
app.listen(app.get('port'), function() {
    console.log('running on port', app.get('port'))
})
```

emc2 – это дескриптор POST, который обрабатывает запрос от чат-бота, чтобы преобразовать массу в энергию.

2. Создадим файл *package.json*, чтобы сообщить Heroku о пакетах, которые необходимы файлу *index.js*. Создайте файл *package.json* в текстовом редакторе со следующим кодом на языке JSON:

```
{
  „name“: „einstein-bot-webhook“,
  „version“: „1.0.0“,
  “description”: “Einstein Bot Webhook”,
  “main”: “index.js”,
  “scripts”: {
    “test”: “echo \”Error: no test specified\” && exit 1”
  },
  “author”: “Srini Janarthanam”,
  “license”: “ISC”,
  “dependencies”: {
  “body-parser”: “^1.15.2”,
  “express”: “^4.14.0”,
  “request”: “^2.72.0”
  }
}
```

3. В качестве последнего шага нам нужно создать файл *Procfile*, чтобы сообщить Heroku о запуске файла *index.js*. Создайте файл *Procfile* в текстовом редакторе с помощью следующей команды:

```
web: node index.js
```

Выполните следующие шаги, чтобы поместить программу интерфейса в облако. В окне консоли проекта интерфейса Einstein выполните показанные ниже команды для создания приложения Heroku и разместите приложение интерфейса в облаке:

```
> git init
> git add .
```

```
> git commit -m "webhook for einstein bot v1"
> heroku create einstein-bot
> git push heroku master
```

Вы получите сообщение с URL-адресом приложения в нем. В моем случае это **https://einstein-bot.herokuapp.com**.

Чтобы настроить вебхук для агента, выполните следующие действия.

1. Щелкните мышью по ссылке **Fulfillment** (Осуществление) в меню слева.
2. Включите вебхук. Вы увидите показанную ниже форму:

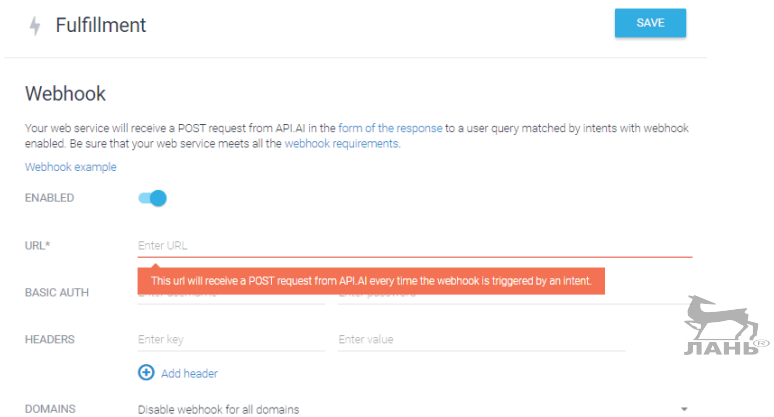


Рис. 4.43

3. Укажите URL-адрес веб-приложения и щелкните мышью по кнопке **Save** (Сохранить). В нашем примере это **https://einstein-bot.herokuapp.com/emc2/**. На данный момент мы не будем заботиться об аутентификации и других заголовках.

Чтобы включить вебхуки в рамках целей, прокрутите страницу вниз до конца. На вкладке **Fulfillment** (Осуществление) щелкните мышью по ссылке **Use webhook** (Использовать вебхук). Расширяя веб-приложение, разработанное ранее для преобразования массы в энергию с использованием известного уравнения Эйнштейна, посмотрим, как оно может быть встроено в агента. Чтобы сделать это, сначала создадим цель `request_emc`, которая распознает высказывания, требующие преобразования (например, «Можете ли

вы преобразовать массу 5 граммов в энергию?»). Для этой цели требуется параметр (т. е. `mass`) и указывается в виде системной сущности (`@sys.unit-weight`).

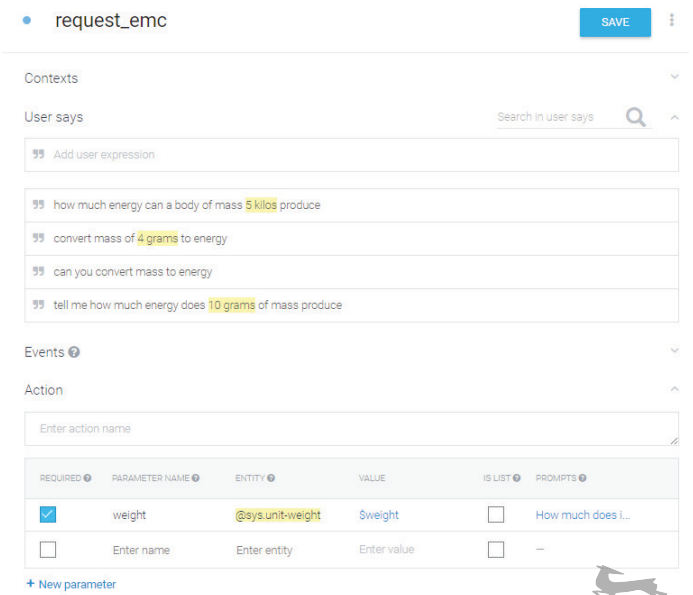


Рис. 4.44

Ответ по умолчанию должен быть подходящим на случай, если вебхук не работает. Включите вебхук, чтобы веб-приложение вызывалось для обработки и предоставляло агенту ответное высказывание, – рис. 4.45.

На следующем рисунке показано, что вебхук действительно работает в процессе общения на платформе веб-чата – рис. 4.46.

Заключение

Отличная работа! В этой главе мы разработали и создали продвинутого диалогового чат-бота, эмулирующего Эйнштейна, с использованием ресурса Dialogflow.com. Очевидно, что чат-бот, которого мы создали, очень простой, и для создания того, который сможет эмулировать человека на достаточно приличном уровне, потребуется огромное количество времени и усилий. Однако это упражнение

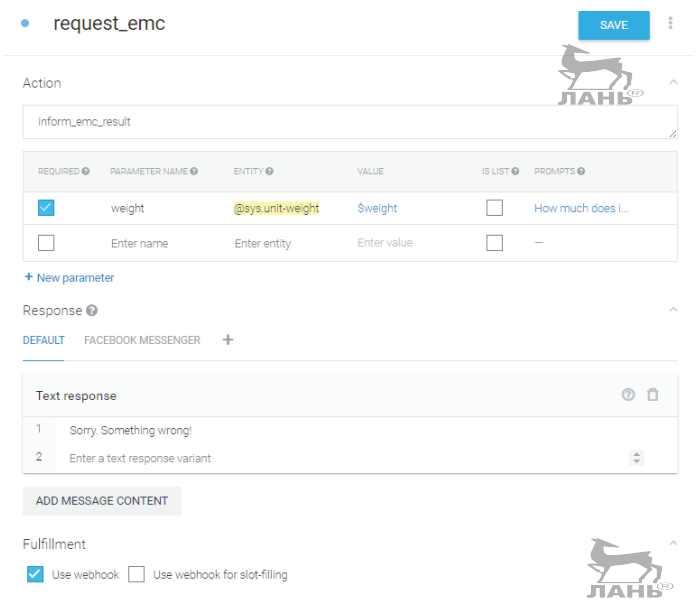


Рис. 4.45

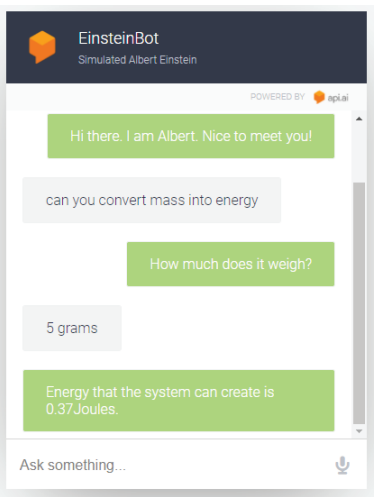


Рис. 4.46

было полезно для демонстрации концепций, лежащих в основе распознавания естественного языка, в диалоговых элементах, таких как цели, сущности и контексты. На практических примерах я продемонстрировал, как выполнять интеграцию на платформы развертывания, такие как Facebook Messenger и веб-чат. Я показал, как можно управлять и интегрировать серверные задачи в диалог.

Я бы посоветовал вам попробовать готовых агентов, которые можно импортировать в любой агент, чтобы улучшить его навыки общения. Например, к вашему агенту может быть добавлен небольшой модуль для обсуждения, чтобы сделать его более персонифицированным, не создавая и не кодируя всех целей и ответов, необходимых для небольших диалогов. Существуют интересные инструменты других разработчиков. Их можно исследовать точно так же. Структура большинства этих инструментов очень схожа. Ниже представлен список инструментов, которые вы могли бы изучить:

- MS LUIS;
- IBM Watson;
- Facebook WIT.AI;
- Amazon Lex.

В следующей главе мы разработаем и создадим SMS-чат-бота для планирования поездок.

Ссылки

- Документация по Dialogflow: **dialogflow.com/docs/getting-started/basics**.
- Документация по Facebook Messenger: **developers.facebook.com/docs/messenger-platform**.

Глава 5. Общественный транспорт

До сих пор мы создавали чат-ботов и разворачивали их на платформах обмена сообщениями, таких как Facebook Messenger. Платформа Facebook Messenger в настоящее время наиболее популярна. Тем не менее уже долгое время существует другая служба обмена сообщениями, которая по-прежнему пользуется популярностью у клиентов. Это служба коротких сообщений, сокращенно **SMS**, и она прекрасно подходит для чат-ботов. Компании во всем мире используют эту услугу для отправки тысяч уведомлений клиентам: маркетинг, продажи, транзакции, доставка и т. д. И самое главное, что для работы не требуется передача каких-либо данных через Интернет.

В этой главе мы разработаем и построим чат-бота, помогающего пользователям спланировать путешествие на поезде. Мы будем использовать Transport API для получения информации о поездах, их времени прибытия и отбытия на станциях и т. д., а также для обслуживания клиентов, чтобы чат-бот мог взаимодействовать с ними на естественном языке. Затем мы рассмотрим, как чат-бот может быть связан с платформой SMS. Вы научитесь пользоваться услугами провайдера коммуникационного API под названием Twilio. Вы также узнаете, как создавать и интегрировать в чат-боты агентов Dialogflow для распознавания пользовательских высказываний и управления ходом беседы.

В ходе этой главы вы:

- разработаете диалоговые задачи на основе данных;
- создадите серверные модули задач с использованием Transport API;
- научитесь создавать SMS-ботов с помощью Twilio;
- интегрируете агента Dialogflow для распознавания пользовательских высказываний.

Обзор Transport API

Для начала взглянем на источник данных, который мы собираемся использовать в этой главе. Transport API – это служба передачи

данных для всех служб общественного транспорта в Великобритании. Если вы находитесь в другой стране, вы сможете найти что-то похожее у себя. Transport API – это платформа транспортных данных, предоставляющая информацию о прибытиях и отправлениях поездов в реальном времени, расписаниях, планировании поездок, тарифах, показателях эффективности и пассажирах. Данные передаются с использованием веб-служб RESTful.

Создание учетной записи разработчика

1. Чтобы начать работу, создайте учетную запись разработчика на странице **developer.transportapi.com** и получите ключ приложения и идентификатор приложения.

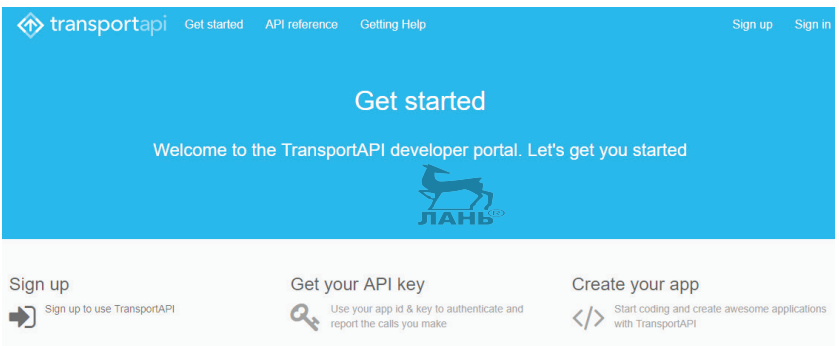


Рис. 5.1

2. Выполните показанный ниже запрос. Замените значения `ВАШ_ИДЕНТИФИКАТОР_ПРИЛОЖЕНИЯ` и `ВАШ_КЛЮЧ_ПРИЛОЖЕНИЯ` своими идентификатором и ключом приложения. Выполните показанный ниже запрос `GET` из веб-браузера. По этому запросу мы попытаемся получить информацию о железнодорожной станции Эустон в Лондоне.

```
http://transportapi.com/v3/uk/places.json?query=euston&type=train_station&app_
id=ВАШ_ИДЕНТИФИКАТОР_ПРИЛОЖЕНИЯ&app_key=ВАШ_КЛЮЧ_ПРИЛОЖЕНИЯ
```

Предыдущий запрос возвращает ответ в формате JSON с информацией, содержащей полное название станции, ее координаты широты и долготы и код станции.

```
{
  "request_time": "2017-08-05T07:06:08+01:00",
  "source": "Network Rail",
  "acknowledgements": "Contains information of Network Rail Infrastructure Limited. License
  http://www.networkrail.co.uk/data-feeds/terms-and-conditions/",
  "member": [
    {
        "type": "train_station",
        "name": "Edinburgh Gateway",
        "latitude": 55.940938,
        "longitude": -3.320251,
        "accuracy": 100,
        "station_code": "EGY",
        "tiploc_code": "EDINGWY"
    },
    {
        "type": "train_station",
        "name": "Edinburgh",
        "latitude": 55.952391,
        "longitude": -3.188228,
        "accuracy": 100,
        "station_code": "EDB",
        "tiploc_code": "EDINBUR"
    },
    {
        "type": "train_station",
        "name": "Edinburgh Park",
        "latitude": 55.927549,
        "longitude": -3.307664,
        "accuracy": 100,
        "station_code": "EDP",
        "tiploc_code": "EDINPRK"
    }
  ]
}
```


Рис. 5.2

Изучение набора данных

Transport API предоставляет данные о поездах, автобусах, подземке (метро) и многих других видах транспорта. Чтобы разработать нашего чат-бота, начнем с данных о поездах. В этом разделе рассмотрим различные типы данных, применимые к поездам. Для всех запросов базовым URL-адресом является **http://transportapi.com/v3/uk/**.

Остановки рядом с вами

Этот запрос предоставляет список железнодорожных станций вблизи данного места. Позиция поиска должна быть представлена в виде координат широта/долгота:

```
http://transportapi.com/v3/uk/train/stations/near.json?lat=55.9485&lon=-3.2021&app_id=ВАШ_ИДЕНТИФИКАТОР_ПРИЛОЖЕНИЯ&app_key=ВАШ_КЛЮЧ_ПРИЛОЖЕНИЯ
```

Ответ

Ответ, который мы получим, – это список станций вблизи точки с указанными координатами. Для каждой станции мы получаем ее название, местоположение и код.

```
{
    "minlon": -3.4521,
    "minlat": 55.6985,
    "maxlon": -2.9521,
    "maxlat": 56.1985,
    "searchlon": -3.2021,
    "searchlat": 55.9485,
    "page": 1,
    "rpp": 25,
    "total": 54,
    "request_time": "2017-08-05T06:21:24+01:00",
    "stations": [ … ] // 25 items
}
```

```
"stations": [
    {
        "station_code": "EDB",
        "atcocode": null,
        "tiploc_code": "EDINBUR",
        "name": "Edinburgh Waverley",
        "mode": "train",
        "longitude": -3.188228,
        "latitude": 55.952391,
        "distance": 966
    },
    { … }, // 8 items
    { … }, // 8 items
    { … }, // 8 items
```


Рис. 5.3

Поезда в этом районе

Железнодорожные вокзалы также можно искать в определенном квадрате. Для этого нужно указать верхнюю левую и нижнюю правую координаты квадрата. Особенно пригодится, если вам нужен список всех станций вблизи какого-то города. Ограничим поиск станциями вблизи Эдинбурга:

---

http://transportapi.com/v3/uk/train/stations/bbox.json?minlon=-3.4521&minlat=55.6985&maxlon=-2.9521&maxlat=56.1985&app_id=**ВАШ_ИДЕНТИФИКАТОР_ПРИЛОЖЕНИЯ**&app_key=**ВАШ_КЛЮЧ_ПРИЛОЖЕНИЯ**

Ответ

Ответ, который мы получим, выглядит следующим образом – рис. 5.4.

Отправления в реальном времени

Действующее расписание поездов, прибывающих и отбывающих с данной станции, может быть получено с использованием показанного ниже запроса. Попробуем решить задачку для вокзала Эдинбург-Уэверли, чей код – EDB.

---

http://transportapi.com/v3/uk/train/station/EDB/live.json?&app_id=**ВАШ_ИДЕНТИФИКАТОР_ПРИЛОЖЕНИЯ**&app_key=**ВАШ_КЛЮЧ_ПРИЛОЖЕНИЯ**

Ответ

Вот что мы получим – рис. 5.5.

Расписание на конкретной станции

Расписания поездов, прибывающих и отходящих от указанной станции в заданную дату и время, могут быть получены с использованием

ЛАНЬ


```
{
    "minlon": -3.4521,
    "minlat": 55.6985,
    "maxlon": -2.9521,
    "maxlat": 56.1985,
    "searchlon": -3.2021,
    "searchlat": 55.9485,
    "page": 1,
    "rpp": 25,
    "total": 54,
    "request_time": "2017-08-05T06:58:20+01:00",
    "stations": [
        {
            "station_code": "EDB",
            "atcocode": null,
            "tiploc_code": "EDINBUR",
            "name": "Edinburgh Waverley",
            "mode": "train",
            "longitude": -3.188228,
            "latitude": 55.952391,
            "distance": 966
        },
        {
            "station_code": "HYM",
            "atcocode": null,
            "tiploc_code": "HAYMRKT",
            "name": "Haymarket",
            "mode": "train",
            "longitude": -3.218449,
            "latitude": 55.945806,
            "distance": 1061
        },
```


ЛАНЬ

Рис. 5.4

Raw Parsed


```
{
    "date": "2017-08-05",
    "time_of_day": "06:00",
    "request_time": "2017-08-05T06:49:44+01:00",
    "station_name": "Edinburgh Waverley",
    "station_code": "EDB",
    "departures": {
        "all": [
            {
                "mode": "train",
                "service": "13560015",
                "train_uid": "G83644",
                "platform": "10",
                "operator": "SR",
                "operator_name": "Scotrail",
                "aimed_departure_time": "06:07",
                "aimed_arrival_time": null,
                "aimed_pass_time": null,
                "origin_name": "Edinburgh Waverley",
                "source": "ATOC",
                "destination_name": "Milngavie",
                "category": "OO",
                "service_timetable": {
                    "id": "http://transportapi.com/v3/uk/train/service/train_uid:G83644/2017-08-
                    05/timetable.json?app_id=60ce46ea&app_key=251dc61414961e8ebfe110329ffa367d"
                }
            },
            {
                "mode": "train",
                "service": "22180008",
                "train_uid": "P31962",
                "platform": "7",
                "operator": "XC",
                "operator_name": "CrossCountry",
```


Рис. 5.5

показанного ниже запроса. Найдем все поезда, прибывающие и покидающие станцию Эдинбург-Уэверли (EDB) 5 августа 2017 года в 6 утра.

```
http://transportapi.com/v3/uk/train/station/EDB/2017-08-05/06:00/timetable.
json?app_id=ВАШ_ИДЕНТИФИКАТОР_ПРИЛОЖЕНИЯ&app_key=ВАШ_КЛЮЧ_ПРИЛОЖЕНИЯ
```

Ответ

И получим ответ:


```
{
    "date": "2017-08-05",
    "time_of_day": "06:00",
    "request_time": "2017-08-05T06:49:44+01:00",
    "station_name": "Edinburgh Waverley",
    "station_code": "EDB",
    "departures": {
        "all": [
            {
                "mode": "train",
                "service": "13560015",
                "train_uid": "G83644",
                "platform": "10",
                "operator": "SR",
                "operator_name": "Scotrail",
                "aimed_departure_time": "06:07",
                "aimed_arrival_time": null,
                "aimed_pass_time": null,
                "origin_name": "Edinburgh Waverley",
                "source": "ATOC",
                "destination_name": "Milngavie",
                "category": "OO",
                "service_timetable": {
                    "id": "http://transportapi.com/v3/uk/train/service/train_uid:G83644/2017-08-
                    05/timetable.json?app_id=60ce46ea&app_key=251dc61414961e8ebfe110329ffa367d"
                }
            },
            {
                "mode": "train",
                "service": "22180008",
                "train_uid": "P31962",
                "platform": "7",
                "operator": "XC",
                "operator_name": "CrossCountry",
```


Raw Parsed

Рис. 5.6

Расписание конкретного поезда

Также можно получить расписание для конкретного поезда. Эти данные позволяют увидеть время прибытия и отбытия определенного поезда на выбранных станциях, в определенную дату и время. Выполним запрос с поездом 23587103, 5 августа 2017 года в 6 утра.

```
http://transportapi.com/v3/uk/train/service/23587103/2017-08-05/06:00/timetable.
json?app_id=ВАШ_ИДЕНТИФИКАТОР_ПРИЛОЖЕНИЯ&app_key=ВАШ_КЛЮЧ_ПРИЛОЖЕНИЯ
```

Ответ

Ответ, который мы получим, выглядит следующим образом:


```
{
    "service": "23587103",
    "train_uid": "G82394",
    "headcode": "",
    "toc": {
        "atoc_code": "SR"
    },
    "train_status": "P",
    "origin_name": "Markinch",
    "destination_name": "Edinburgh Waverley",
    "stop_of_interest": null,
    "date": "2017-08-05",
    "time_of_day": "06:00",
    "mode": "train",
    "request_time": "2017-08-05T06:46:22+01:00",
    "category": "OO",
    "operator": "SR",
    "operator_name": "Scotrail",
    "stops": [
        {
            "station_code": "MNC",
            "tiploc_code": "MKIN",
            "station_name": "Markinch",
            "stop_type": "LO",
            "platform": "1",
            "aimed_departure_date": "2017-08-05",
            "aimed_departure_time": "06:19",
            "aimed_arrival_date": null,
            "aimed_arrival_time": null,
            "aimed_pass_date": null,
            "aimed_pass_time": null
        },
        {
            "station_code": "GLT",
            "tiploc_code": "GLNRTHS",
            "station_name": "Glenrothes With Thornton",
```


Рис. 5.7

Структура диалога

Теперь, когда у вас есть представление о данных, которые нам доступны, продумаем диалоговые задачи для нашего чата.

- ближайшая станция;
- следующий поезд;
- время прибытия;
- более поздние поезда со станции;
- маршрут от точки А до точки Б.

Начнем с простого: определение ближайшей станции.

Ближайшая станция

Чтобы добраться до ближайшей станции по запросу пользователя, нам нужно узнать его/ее местоположение. Информация может быть предоставлена в виде почтового индекса или координат широта/

долгота. Платформы, такие как Facebook Messenger, позволяют пользователям делиться местоположением в виде геометки с координатами широта/долгота. Однако, поскольку мы собираемся разместить чат-бота на платформе SMS, мы будем использовать почтовый индекс. Диалог для этой задачи может осуществляться следующим образом:

```
Пользователь: Какая ближайшая станция?
Бот: Вы можете указать индекс?
Пользователь: EH12 9LD.
Бот: Отлично. Ближайшая станция - Юг Джайл.

Пользователь: Какая станция ближе всего к месту с индексом EH12 9QR?
Бот: Ближайшая станция - Юг Джайл.
```

Следующий поезд

Чтобы получить информацию о следующем поезде, диалог может происходить таким образом:

```
Пользователь: Когда ближайший поезд до Глазго?
Бот: С какой станции?
Пользователь: Парк Эдинбург.
Бот: Следующий поезд до Глазго-сентрал в 10:00.

Пользователь: Ближайший поезд.
Бот: Откуда?
Пользователь: Парк Эдинбург.
Бот: Ищу ближайший поезд из Эдинбургского парка. Куда?
Пользователь: Глазго.
Бот: Следующий поезд до Глазго-сентрал в 10:00.
```

Время прибытия

Иногда пользователи могут спрашивать время прибытия поезда в контексте конкретной станции. Чтобы получить информацию о времени прибытия, диалог может происходить следующим образом:

```
Пользователь: Когда поезд прибывает на Глазго-сентрал?
Бот: Поезд прибудет на Глазго-сентрал в 11:00.
```

В отношении путешествий возможно множество других диалогов. Все предыдущие задачи были реактивными, когда

бот отвечает на запросы пользователей. И напротив, могут быть созданы упреждающие задачи, когда бот сам инициирует разговор. Например, бот может отправлять время поездок и информацию по задержке или отмене поездов пользователю в установленные сроки в течение дня:

```
Бот: Поезд до вокзала Эдинбург-Уэверли со временем прибытия в 15:00 задерживается
до 15:30.
Пользователь: Есть ли поезд до Хеймаркета  пораньше?
.
```

Диалоговые задачи, такие как список более поздних поездов и планирование вашего маршрута, более сложны, чем предыдущие задачи. По мере чтения главы вы увидите, как создать чат-бота, который может обрабатывать некоторые из упомянутых задач.

Создание простого SMS-бота

Теперь мы создадим SMS-интерфейс для чат-бота. Для упрощения сделаем это в два этапа. Во-первых, построим бота, способного просто отправлять текстовые SMS-сообщения на номер сотового телефона. Это может быть сообщение о статусе поезда, прибывающего на станцию, или о ближайшем поезде до определенного пункта назначения с данной станции. Во-вторых, мы создадим двухстороннего чат-бота, который сможет получать сообщения от пользователей и отвечать на них соответствующим образом. Для этого мы воспользуемся сервисом под названием Twilio. Twilio – это платформа для разработчиков, позволяющая разработчикам добавлять свои программные средства для обмена сообщениями, голоса и видео. Мы рассмотрим, как создавать ботов уведомлений и чат-ботов, используя инфраструктуру обмена сообщениями Twilio.

Начало работы

Чтобы начать работу с Twilio, выполните следующие действия.

1. Перейдите на сайт **www.twilio.com**, щелкните мышью по ссылке **Sign Up** (Зарегистрироваться) и зарегистрируйте бесплатную пробную учетную запись.
2. Зарегистрировавшись, перейдите на страницу консоли на странице **www.twilio.com/console**.

3. Скопируйте идентификатор безопасности (**ACCOUNT SID**) и ключ учетной записи (**Account Key**). Мы будем использовать их в наших проектах.

Настройка панели управления

Создадим бота, который отправляет уведомления на номер мобильного телефона пользователя. Twilio предлагает четыре основных инструмента: программируемый чат (**Programmable Chat**), программируемые SMS (**Programmable SMS**), программируемый голос (**Programmable Voice**) и программируемое видео (**Programmable Video**). Чтобы создать отправителя уведомлений, нам понадобится инструмент **Programmable SMS**. Выполните следующие действия.

1. На странице консоли выберите пункт **Programmable SMS** (Программируемые SMS).
2. На странице **Programmable SMS** (Программируемые SMS) щелкните мышью по кнопке **Get Started** (Начать).

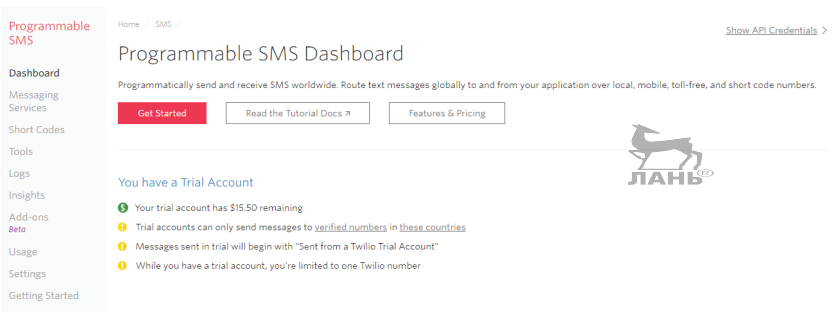


Рис. 5.8

3. Чтобы отправлять SMS-сообщения, вам понадобится номер телефона. Щелкните мышью по ссылке **Get a number** (Получить номер), чтобы приобрести его, – рис. 5.9.
4. Вам будет предложен телефонный номер. Вы можете принять его, нажав кнопку **Choose this Number** (Выбрать этот номер), или выбрать другой. Убедитесь, что выбранный номер поддерживает передачу SMS-сообщений, – рис. 5.10.
5. Вы увидите подтверждение, что телефонный номер приобретен, – рис. 5.11.

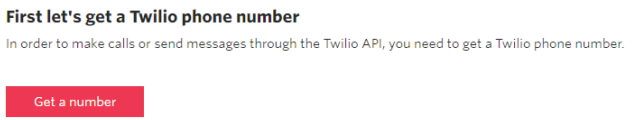


Рис. 5.9

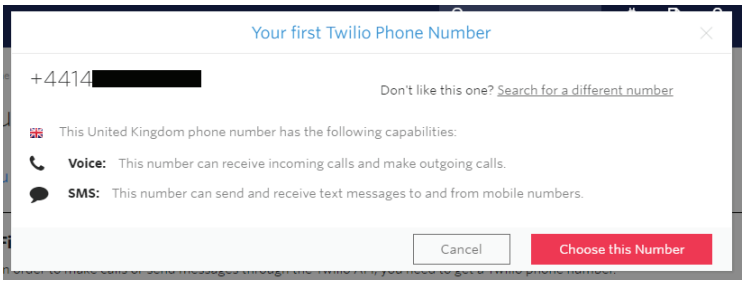


Рис. 5.10

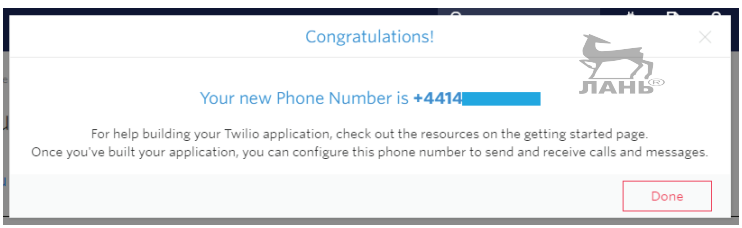


Рис. 5.11

Щелкните мышью по кнопке **Done** (Готово).

6. Появится окно **Send a Message** (Отправить сообщение). Отправьте тестовое сообщение на зарегистрированный номер мобильного телефона (номер, который вы использовали для проверки учетной записи) – рис. 5.12.
7. Проверьте свой мобильный телефон, чтобы узнать, получили ли вы тестовое сообщение. Щелкните мышью по ссылке **Yes** (Да), чтобы сообщить Twilio, что вы получили сообщение.

Отправка простых сообщений

Теперь, когда мы создали учетную запись и получили номер телефона, создадим бота для отправки уведомлений. Для этого выполните следующие действия.

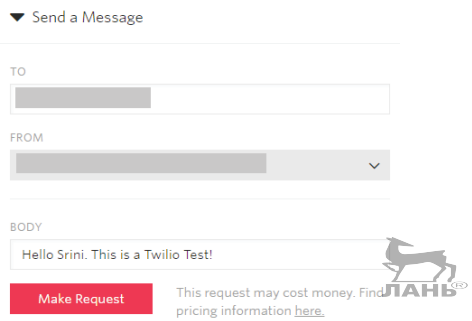


Рис. 5.12

1. Откройте окно консоли и создайте каталог с именем **SMSBot**.
2. В каталоге *SMSBot* создайте новый проект **Node.js**, выполнив команду `npm init`, как показано ниже, – рис. 5.13.
   Проверьте каталог, чтобы просмотреть файлы, которые были созданы. Вы увидите файл с именем package.json с метаинформацией о проекте.

```
C:\Users\Srini\Documents\workspace\SMSBot>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (SMSBot) smsbot
version: (1.0.0)
description: A simple SMS sending bot
entry point: (index.js)
test command:
git repository:
keywords:
author: Srini Janarthanam
license: (ISC)
About to write to C:\Users\Srini\Documents\workspace\SMSBot\package.json:

{
  "name": "smsbot",
  "version": "1.0.0",
  "description": "A simple SMS sending bot",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "Srini Janarthanam",
  "license": "ISC"
}
```

Рис. 5.13

3. Мы должны установить библиотеку Twilio Node.js для создания SMS-бота. В консоли выполните команду `npm install twilio --save` – рис. 5.14.

```
C:\Users\Srini\Dropbox\_Book\workspace\twitterbot>npm install twitter --save
npm WARN package.json twitterbot@1.0.0 No repository field.
npm WARN package.json twitterbot@1.0.0 No README data
twitter@1.7.1 node_modules\twitter
├── deep-extend@0.5.0
└── request@2.81.0 (aws-sign2@0.6.0, oauth-sign@0.8.2, forever-agent@0.6.1, tunnel-agent@0.6.
0, is-typedarray@1.0.0, caseless@0.12.0, safe-buffer@5.1.1, stringstream@0.0.5, isstream@0.1.
2, aws4@1.6.0, json-stringify-safe@5.0.1, extend@3.0.1, performance-now@0.2.0, uuid@3.1.0, qs
@6.4.0, combined-stream@1.0.5, mime-types@2.1.16, tough-cookie@2.3.2, form-data@2.1.4, hawk@3
.1.3, http-signature@1.1.1, har-validator@4.2.1)
```

Рис. 5.14

4. Создайте новый JS-файл с именем *index.js*. Добавьте в файл следующий код:

```
//Index.js - SMSBot

// Укажите идентификатор безопасности
var accountSid = 'ваш_account_sid';

// Укажите токен авторизации
var authToken = 'ваш_auth_token';

var twilio = require('twilio');
var client = new twilio(accountSid, authToken);

// Создание сообщения с номерами отправителя и получателя
client.messages.create({
    body: 'Srini says hello',
    to: '+447888999999',
    from: '+447888999990'
})
.then((message) => console.log(message.sid));
```

В этом коде в строке `to` указывается допустимый номер из числа пробных учетных записей. Вы не сможете отправлять сообщения на другие номера, пока не обновите свой аккаунт на сайте до версии Pro. Значение в строке `from` – телефонный номер, полученный вами ранее на сайте Twilio.

5. Сохраните файл и запустите его, выполнив команду `node index.js`. Программа должна отправить текстовое сообщение на ваш подтвержденный номер телефона.

Уведомления о поездах

Представьте себе ситуацию, что пользователь хочет узнать ежедневное расписание на пути от вокзала Эдинбург-Уэверли до вокзала Квин-стрит в Глазго. Он хочет разобраться, какие поезда в какое время отправляются каждое утро. Было бы круто, если бы мы могли уведомлять его сообщением со списком подходящих поездов каждое утро.

Используя модуль из предыдущего раздела, построим бота, который будет отправлять список поездов, курсирующих между определенными станциями. Добавим модуль, чтобы получить список поездов от данной станции к месту назначения.

1. Установите библиотеку запросов с помощью команды `npm install request --save`.
2. Создайте функцию отправки уведомлений посредством SMS:

```
function sendSMS(msg, userPhoneNumber){
    var twilio = require('twilio');
    var client = new twilio(accountSid, authToken);
    // Создание сообщения с номерами отправителя и получателя
    client.messages.create({
        body: msg,
        to: userPhoneNumber,
        from: '+4414XXXXXXXX' //YOUR_NUMBER
    })
    .then((message) => console.log(message.sid));
}
```

3. Создайте функцию, запрашивающую список всех поездов, отправляющихся с указанной станции:

```
function getTrains(sourceStation, sourceStationCode,
                                destinationStation,
                                userPhoneNumber){
    var request = require('request');
    var url = 'http://transportapi.com/v3/uk/train/station/' +
                sourceStationCode + '/live.json?
                app_id=ВАШ_ИДЕНТИФИКАТОР_ПРИЛОЖЕНИЯ&app_key=ВАШ_КЛЮЧ_ПРИЛОЖЕНИЯ';
    request(url, function (error, response, body) {
        if (response){
            var json = JSON.parse(body);
            if (json.departures){
```

```
                //console.log('Departures:',
                //JSON.stringify(json.departures));
                var dep =
                getTrainsToDestination(destinationStation,
                json.departures.all);
                var summary = summarize(destinationStation,
                                        sourceStation, dep);
                console.log('Summary: ' + summary);
                sendSMS(summary, userPhoneNumber);
            } else {
                console.log('No Departures found!');
            }
        } else {
            console.log('error:', error); // Вывод ошибки при возникновении
        }
    });
}
```

Создайте функцию для выборки всех поездов, следующих до станции назначения:

```
function getTrainsToDestination(destination, allDepartures){
    var d = [];
    if (allDepartures){
        for (var i=0; i < allDepartures.length; i++){
            var service = allDepartures[i];
            if (service.destination_name == destination){
                d.push(service)
            }
        }
    }
    return d;
}
```

4. Создайте функцию для вызова предыдущих функций, чтобы отправить пользователю уведомление обо всех поездах на определенную станцию назначения со станции, выбранной пользователем:

```
function summarize(destinationStation, sourceStation, departures){
    var out = '';
    if (departures.length > 0){
        out = 'Here are the departures this morning to ' +
              destinationStation
                                + ".\n";
```

```
        for (var i=0; i< departures.length; i++){
            var service = departures[i];
            var serviceSummary = service.operator_name
                                 + " at " +
            service.expected_departure_time;
            out += serviceSummary + "\n"
        }
    } else {
        out = 'There are no trains to ' + destinationStation +
                                                  ' from ' +
                                          sourceStation;
    }
    return out;
}
```

И наконец, установите переменные и вызовите главный модуль.

```
//Index.js
// Укажите идентификатор безопасности

var accountSid = 'ваш_account_sid';
// Укажите токен авторизации

var authToken = 'ваш_auth_token';
var destinationStation = 'Glasgow Queen Street';
var userPhoneNumber = '+447888999999';
var sourceStationCode = 'EDB';
var sourceStation = 'Edinburgh Waverley';

getTrains(sourceStation, sourceStationCode, destinationStation, userPhoneNumber);
```

5. Выполните в консоли команду `node index.js`, чтобы проверить, работает ли код. Вы должны увидеть сводку поездов, следующих до станции назначения, а также получить ее в виде SMS-сообщения.

```
C:\Users\Srini\Documents\workspace\SMSBot>node index.js
Summary: Here are the departures this morning to Glasgow Queen Street.
Scotrail at 06:45
Scotrail at 07:00
Scotrail at 07:15
Scotrail at 07:30

SM79cb0e9ed7e5499f99ec35b3c2f77681

C:\Users\Srini\Documents\workspace\SMSBot>
```

Рис. 5.15

6. Проверьте свой мобильный телефон, чтобы узнать, поступило ли сообщение.

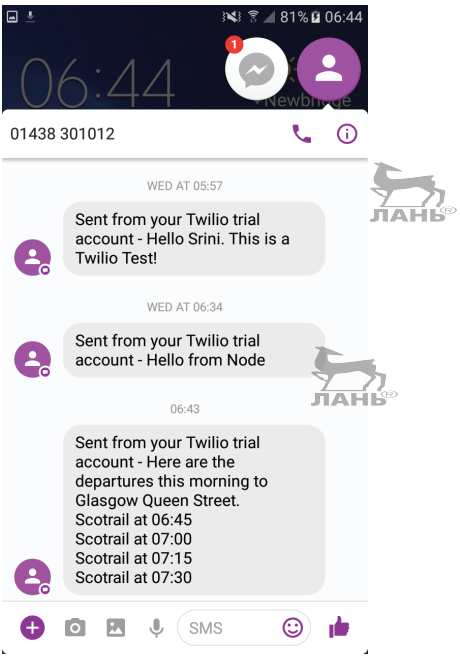


Рис. 5.16

7. Поздравляем! SMS-бот готов.

Планирование задач

Теперь, когда у нас есть бот, который рассылает SMS-уведомления, настроим его так, чтобы он автоматически запускался ежедневно или ежечасно. Эта функция будет полезна для создания проактивных ботов, которые инициируют диалог с пользователями в определенное время дня. Для этого выполните следующие действия.

1. Создайте каталог *bin*.
2. Переместите в каталог *bin* файл *index.js*. Переименуйте его в файл *sendTrainNotification.js*.
3. Добавьте первую строку кода:

```
#!/usr/bin/env node
```

4. Вернитесь в каталог проекта. Теперь мы намерены разместить это приложение в облаке Heroku.
5. Создайте репозиторий Git, добавьте файлы и передайте их:

```
git init
git add .
git commit -m “initial commit”
```

6. Создайте приложение Heroku:

```
heroku create sms-notification-bot
```

7. Разместите приложение в Heroku:

```
git push heroku master
```

Теперь у нас есть приложение, размещенное по адресу **sms-notification-bot.herokuapp.com**.

8. Запустите приложение локально.

```
heroku run sendTrainNotification.js
```

Веб-приложение должно запуститься и отправить SMS с информацией о поезде на телефон пользователя.

```
C:\Users\Srini\Dropbox\_Book\workspace\sms-notification-bot>heroku run sendTrainNotification.js
Running sendTrainNotification.js on sms-notification-bot... up, run.6693 (Free)
Running - SMS Train Notification
Summary: Here are the departures this morning to Glasgow Queen Street.
Scotrail at 16:45
Scotrail at 17:00
Scotrail at 17:15
```

Рис. 5.17

9. Теперь нам нужно запланировать задачу. Для этого нам нужно настроить ее на странице ресурсов Heroku приложения. Перейдите на страницу **dashboard.heroku.com/apps/smsnotification-bot/resources** в своем браузере – рис. 5.18.
10. Введите слово *Scheduler* в поле поиска надстроек и выберите пункт **Heroku Scheduler** (Планировщик Heroku).
11. После добавления запустите **Heroku Scheduler** (Планировщик Heroku). Вы увидите страницу **scheduler.heroku.com/dashboard**.
12. Щелкните мышью по ссылке **Добавить новое задание** (Add new job).

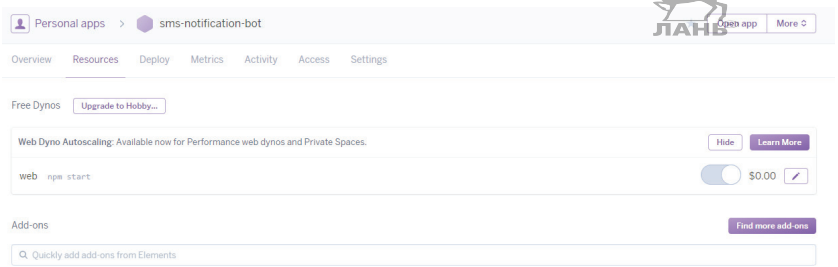


Рис. 5.18

13. В текстовом поле **$** введите название запускаемой задачи (то есть **sendTrainNotification.js**). Выберите время выполнения (ежедневно (**Daily**), ежечасно (**Hourly**) или каждые 10 минут (**Every 10 minutes**)) и нажмите кнопку **SAVE** (Сохранить).

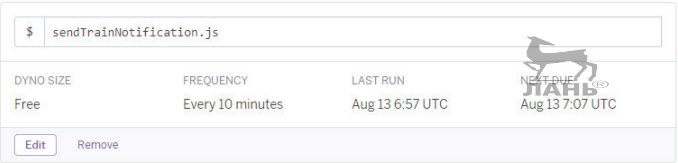


Рис. 5.19

14. Проверьте журналы регистрации событий на странице консоли Heroku. Вы увидите, что задача будет выполняться с регулярными интервалами, и пользователю отправляются SMS-сообщения.

```
2017-08-13T07:07:25.867628+00:00 heroku[scheduler.4555]: Starting process with command `sendTrainNotification.js`
2017-08-13T07:07:26.517425+00:00 heroku[scheduler.4555]: State changed from starting to up
2017-08-13T07:07:28.022517+00:00 app[scheduler.4555]: Running - SMS Train Notification
2017-08-13T07:07:29.133248+00:00 app[scheduler.4555]: Summary: Here are the departures this morning to Glasgow Queen Street.
2017-08-13T07:07:29.133261+00:00 app[scheduler.4555]: Scotrail at 08:30
2017-08-13T07:07:29.133261+00:00 app[scheduler.4555]: Scotrail at 09:00
2017-08-13T07:07:29.133262+00:00 app[scheduler.4555]: Scotrail at 09:30
2017-08-13T07:07:29.133264+00:00 app[scheduler.4555]:
2017-08-13T07:07:29.133263+00:00 app[scheduler.4555]: Scotrail at 10:00
2017-08-13T07:07:29.804633+00:00 app[scheduler.4555]: SM7f6591a12f8f43eab488c7be5d5a9fbf
2017-08-13T07:07:29.892212+00:00 heroku[scheduler.4555]: State changed from up to complete
2017-08-13T07:07:29.880146+00:00 heroku[scheduler.4555]: Process exited with status 0
```

Рис. 5.20

Поздравляю! Вы создали проактивного SMS-бота.

Создание двухстороннего чат-бота

На данный момент мы создали бота, который умеет отправлять SMS-уведомления пользователям через определенные промежутки

времени. Хотя он проактивен, он способен к общению только в одну сторону. Пользователь не может отправить запрос боту, чтобы изменить характер или содержимое отправляемого сообщения. Проработаем этот вопрос.

Чтобы создать чат-бота, который может общаться в обоих направлениях, нам нужно сделать две вещи: создать чат-бота в веб-приложении и изменить настройки конфигурации в Twilio. Для этого выполните следующие действия.

1. Создайте файл *index.js* в корневой директории проекта.
2. Установите библиотеки `express` и `body-parser`. Эти библиотеки будут использоваться для создания веб-приложения.

```
npm install body-parser --save
npm install express -save
```

3. Напишите код веб-приложения в файле *index.js*:

```
// Двунаправленный SMS-бот

const express = require('express')
const bodyParser = require('body-parser')
const twilio = require('twilio')

const app = express()
app.set('port', (process.env.PORT || 5000))

// Обработка application/x-www-form-urlencoded
app.use(bodyParser.urlencoded({extended: false}))

// Обработка application/json
app.use(bodyParser.json())

// обращение к серверу
app.listen(app.get('port'), function() {
    console.log('running on port', app.get('port'))
})

// Индексируем маршрут
app.get('/', function (req, res) {
    res.send('Hello world, I am SMS bot.')
})

// Вебхук Twilio
```

```
app.post('/sms/', function (req, res) {
    var botSays = 'You said: ' + req.body.Body;
    var twiml = new twilio.TwimlResponse();
    twiml.message(botSays);
    res.writeHead(200, {'Content-Type': 'text/xml'});
    res.end(twiml.toString());
})
```

С помощью этого кода создается веб-приложение, которое отслеживает входящие сообщения от пользователей и отвечает на них. На данный момент в ответе повторяется то, что сказал пользователь.

4. Разместите приложение в облаке:

```
git add .
git commit -m webapp
git push heroku master
```

Теперь у нас есть веб-приложение в облаке по адресу **https://sms-notification-bot.herokua pp.com/sms/**, которое можно вызвать при поступлении входящего SMS-сообщения. Это приложение будет генерировать соответствующий ответ чат-бота на входящее сообщение.

5. Перейдите на страницу **Programmable SMS Dashboard** сайта Twilio по адресу **www.twilio.com/console/sms/dashboard**.
6. Выберите в меню пункт **Messaging Services** (Службы обмена сообщениями) и щелкните мышью по кнопке **Create new Messaging Service** (Создать новую службу обмена сообщениями).

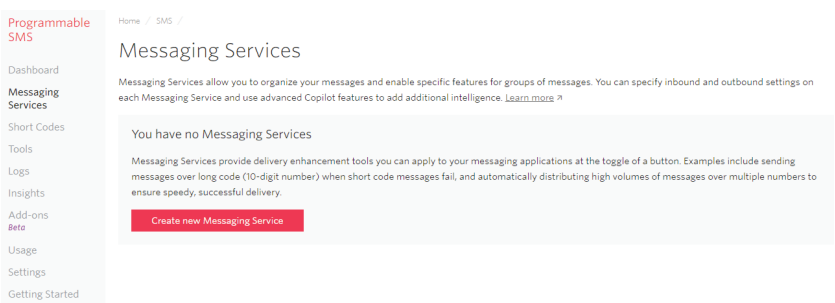


Рис. 5.21

7. Присвойте службе название и выберите пункт **Chat Bot/Interactive 2-Way** в качестве способа использования.

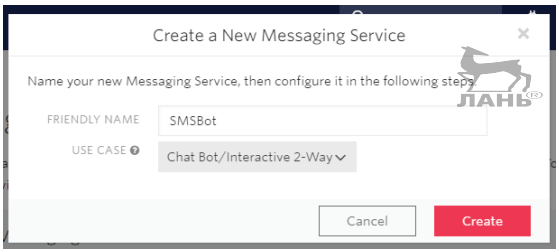


Рис. 5.22

8. Откроется страница конфигурации с новым назначенным идентификатором службы.

Рис. 5.23

9. В разделе **Inbound Settings** (Параметры входящего трафика) укажите URL-адрес созданного веб-приложения (т. е. **https://sms-notification-bot.herokuapp.com/sms/**) в поле **REQUEST URL** (URL-адрес запроса).

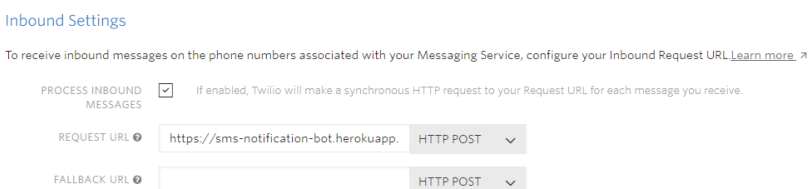


Рис. 5.24

10. Вернитесь на страницу консоли SMS по адресу **www.twilio.com/console/sms/services**. Обратите внимание, что ваша новая

служба обмена сообщениями указана вместе с URL-адресом входящего запроса.

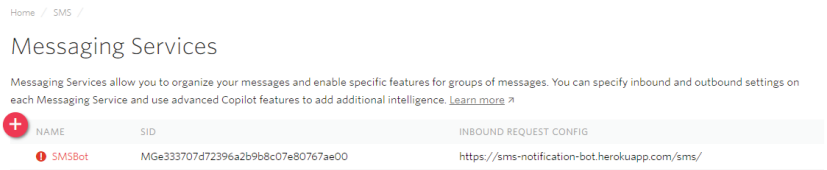


Рис. 5.25

11. Выберите службу и присвойте ей номер.

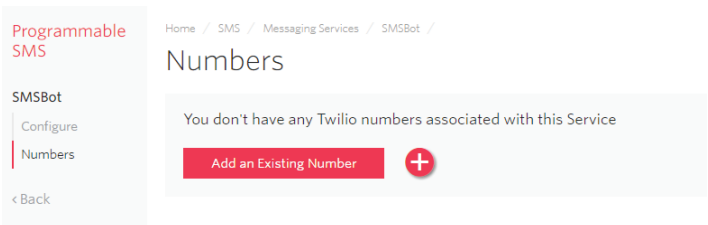


Рис. 5.26

12. Вы можете либо добавить новый номер, и в этом случае вам нужно приобрести его, либо выбрать уже имеющийся. У нас уже настроена одна отправка уведомлений, и этот номер мы можем использовать повторно. Нажмите кнопку **Add an Existing Number** (Добавить существующий номер).

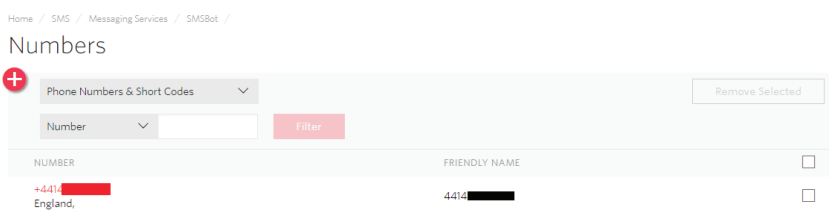


Рис. 5.27

13. Выберите номер, установив флажок справа, и нажмите кнопку **Add Selected** (Добавить выбранное) – рис. 5.28.
14. После добавления номер будет указан на странице **Numbers** (Номера) следующим образом:

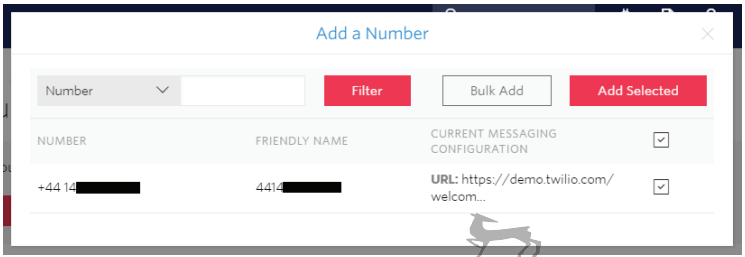


Рис. 5.28

В дополнительных настройках мы можем добавить несколько номеров для обслуживания разных географических регионов и настроить их так, как будто чат-бот отвечает с местного номера.

15. На последнем этапе попробуйте отправить SMS-сообщение на номер и получить ответ. Отправьте сообщение с помощью любого SMS-приложения на свой телефон и посмотрите ответ.

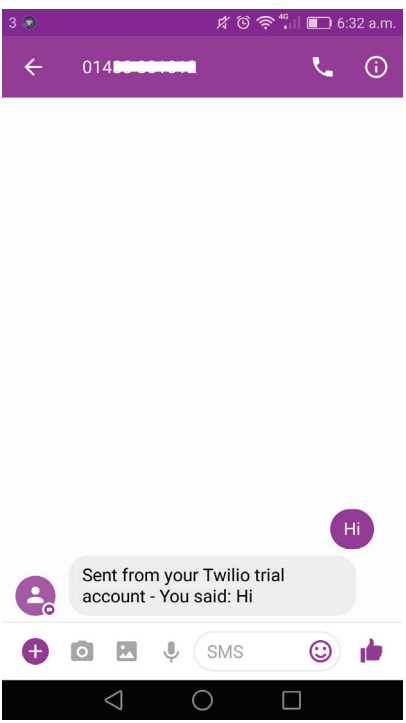


Рис. 5.29

Отлично! Теперь у нас есть двусторонний интерактивный чат-бот.

Обработка произношения пользователей

Теперь, когда у нас есть базовый двухсторонний чат-бот, рассмотрим, как мы можем обрабатывать пользовательские высказывания, распознавать информационные запросы и эффективно обрабатывать их. Для этого мы воспользуемся набором инструментов Dialogflow, который был представлен в главе 3. Пользовательские высказывания будут обработаны и преобразованы в пользовательские цели и параметры Dialogflow. Затем они будут использоваться для извлечения и предоставления соответствующей информации пользователю.

Создание агента Dialogflow

Сначала создадим агента Dialogflow, позволяющего распознавать пользовательские высказывания в диалоговых задачах, которые мы перечислили ранее. Поскольку мы уже подробно обсудили создание агентов Dialogflow в главе 3, следующие шаги будут сокращенными. Тем не менее мы будем взаимодействовать с агентом по-другому, по сравнению с тем, как это было сделано в главе 3.

1. Перейдите на сайт **dialogflow.com** в своем веб-браузере. Авторизуйтесь в системе со своими учетными данными Google и щелкните мышью по ссылке **GO TO CONSOLE** (Перейти в консоль).
2. Щелкните мышью по ссылке **Create new agent** (Создать нового агента). Вы найдете ее в нижней части списка, в котором перечислены все агенты, созданные вами ранее.
3. Присвойте ему название, описание и выберите часовой пояс. Щелкните мышью по кнопке **Save** (Сохранить).
4. Щелкните мышью по ссылке **Entities** (Сущности) в меню. Создайте новые сущности для станций с кодами станций в качестве значений и имен и альтернатив в качестве синонимов – рис. 5.30.
5. Щелкните мышью по ссылке **CREATE INTENT** (Создать цель). Для начала создайте следующие две цели:
   - `request_live_departures`: для запроса следующих нескольких отправляемых поездов с данной станции – рис. 5.31.
     Установите параметр **Station** обязательным и определите подсказки в случае отсутствия параметра в высказываниях пользователя – рис. 5.32;

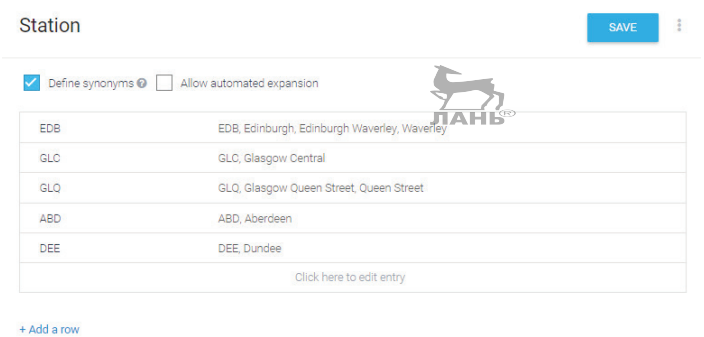


Рис. 5.30

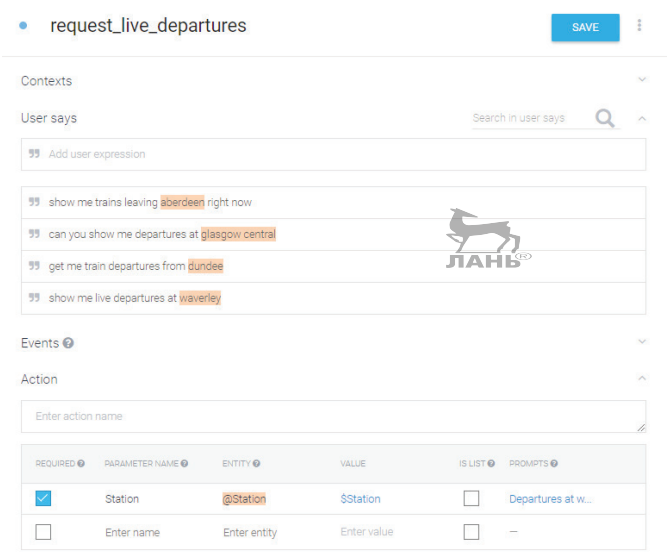


Рис. 5.31

- `request_next_train`: для запроса следующего поезда с данной станции до указанной другой станции – рис. 5.33.
  Задайте параметры **fromStation** и **toStation** обязательными – рис. 5.34.
  Определите подсказки в случае отсутствия параметров в высказываниях пользователя. Ниже показаны подсказки для параметра **fromStation** – рис. 5.35.

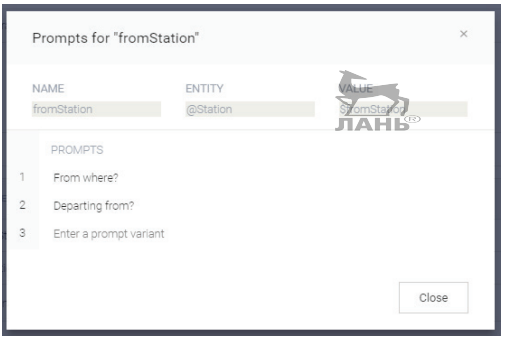


Рис. 5.32

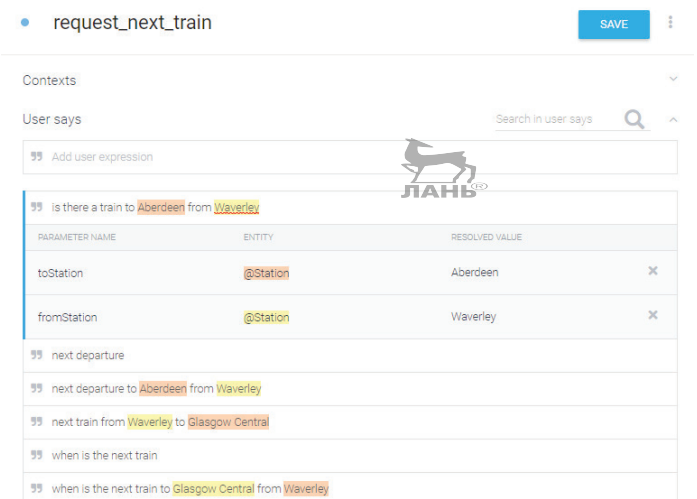


Рис. 5.33

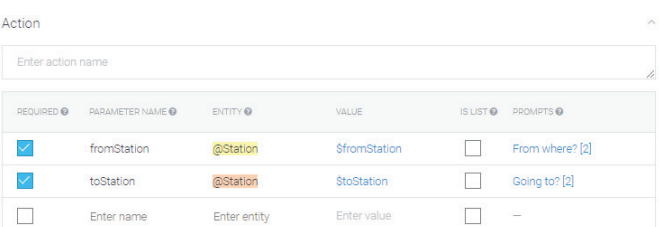


Рис. 5.34

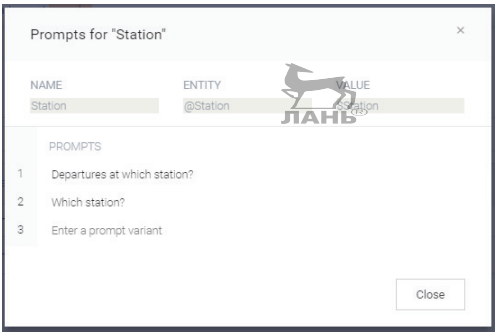


Рис. 5.35

Создайте интерфейс Node.js для агента Dialogflow

Существует несколько способов подключения к агенту Dialogflow. Мы обсудили некоторые из них в главе 3. В этом разделе мы рассмотрим еще один подход.

1. Создайте в корневом каталоге проекта сценарий Node.js с именем *apiai.js*.
2. Установите библиотеку Dialogflow Node.js, выполнив следующую команду в консоли:

```
npm install apiai -save
```

3. Вернитесь к настройкам агента (щелкните мышью по значку настроек рядом с названием агента) на странице веб-консоли.

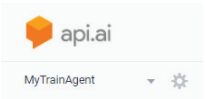


Рис. 5.36

4. На вкладке **General** (Общие) скопируйте значение из поля **Developer access token** (Токен доступа разработчика) – рис. 5.37.
5. Вернитесь к сценарию *apiai.js* и создайте запрос для доступа к агенту. Укажите свой токен доступа разработчика в коде.

```
//Dialogflow
var apiai = require('apiai');
var apiai1 = apiai("YOUR_DEVELOPER_ACCESS_TOKEN");
```

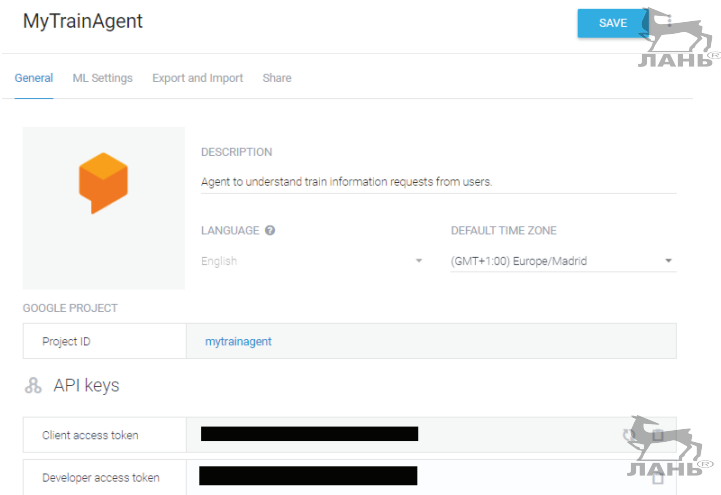


Рис. 5.37

```
var userUtterance = 'live departures from waverley';
var requestAPIAI = apiai1.textRequest(userUtterance, {
 sessionId: '12345'
});

requestAPIAI.on('response', function(response) {
 console.log(response);
});

requestAPIAI.on('error', function(error) {
 console.log(error);
});

requestAPIAI.end();
```

6. Запустите программу в консоли, выполнив команду `node apiai.js`, – рис. 5.38.
7. Взгляните на ответ, отображаемый в консоли. Вы увидите классифицированную цель `intentName` и связанные с ней параметры. Используя их, мы можем получить соответствующую информацию от Transport API. И обратите внимание, что речи о диалоге нет, поскольку мы не определяли его при построении целей.

```
C:\Users\Srini\Documents\workspace\SMSBot>node apiai.js
{ id: 'c8ad170d-df2d-4384-b78d-727cfb394259',
  timestamp: '2017-08-15T04:50:21.052Z',
  lang: 'en',
  result:
   { source: 'agent',
     resolvedQuery: 'waverley',
     action: '',
     actionIncomplete: false,
     parameters: { Station: 'EDB' },
     contexts: [],
     metadata:
      { intentId: 'c35552da-b1b3-41ac-afb4-177533126c08',
        webhookUsed: 'false',
        webhookForSlotFillingUsed: 'false',
        intentName: 'request_live_departures' },
     fulfillment: { speech: '', messages: [Object] },
     score: 1 },
  status: { code: 200, errorType: 'success' },
  sessionId: '12345' }
```

Рис. 5.38

8. Попробуем еще несколько примеров: измените пользовательское высказывание на `show me departures` и запустите код.

```
C:\Users\Srini\Documents\workspace\SMSBot>node apiai.js
{ id: 'c8ad170d-df2d-4384-b78d-727cfb394259',
  timestamp: '2017-08-15T04:50:21.052Z',
  lang: 'en',
  result:
   { source: 'agent',
     resolvedQuery: 'waverley',
     action: '',
     actionIncomplete: false,
     parameters: { Station: 'EDB' },
     contexts: [],
     metadata:
      { intentId: 'c35552da-b1b3-41ac-afb4-177533126c08',
        webhookUsed: 'false',
        webhookForSlotFillingUsed: 'false',
        intentName: 'request_live_departures' },
     fulfillment: { speech: '', messages: [Object] },
     score: 1 },
  status: { code: 200, errorType: 'success' },
  sessionId: '12345' }
```

Рис. 5.39

9. Проанализируйте отличия. В этом высказывании пользователя мы не указали название станции. Поэтому агент возвращается с вопросом (в речевом исполнении): `Departures at which station?` (т. е. «отправление с какой станции?»). Также обратите внимание, что параметр `Station` пуст, а параметру `actionIncomplete` присвоено значение `true`.
10. Теперь измените пользовательское высказывание на `waverley` и выполните код снова – рис. 5.40.
11. Рассмотрим результат. Нет исполнения диалога, а для параметра `Station` задано значение `EDB` (это код станции

Эдинбург-Уэверли). Параметру `actionIncomplete` присвоено значение `false`.

Мы создали сценарий Node.js для вызова агента Dialogflow с использованием библиотеки Node.js. Мы также перезапустили программу для имитации диалога. Мы использовали агента в качестве модуля для распознавания высказываний на естественном языке и управления беседой. Теперь нам нужно связать это с основной программой *index.js*, чтобы получить результаты от Transform API и задействовать обмен посредством SMS.

Интеграция агента Dialogflow в SMS-чат-бота

Теперь, когда у нас есть агент Dialogflow и интерфейс Node.js для взаимодействия с ним, перейдем к их интеграции в двухстороннего SMS-чат-бота. Для этого нам нужно пересмотреть файл *index.js*, представляющий собой веб-приложение, которое отвечает на входящие SMS-сообщения. Мы скопируем и переработаем код из файла *apiai.js*, чтобы вызвать агента Dialogflow из файла *index.js*.

1. В вебхуке Twilio вызовите агента Dialogflow с высказыванием пользователя. Мы также будем использовать номер телефона пользователя как `sessionId`, чтобы контекст диалога не был утерян.

```
// Вебхук Twilio
app.post('/sms/', function (req, res) {
    //передача боту
    var sessionId = req.body.From;
    var userUtterance = req.body.Body;
    // Dialogflow
    var apiai = require('apiai');
    var apiai1 = apiai("YOUR_DEVELOPER_ACCESS_TOKEN");

    var requestAPIAI = apiai1.textRequest(userUtterance, {
        sessionId: sessionId
    });

    var botSays = '';
    requestAPIAI.on('response', function(response) {
        console.log(response);
        if (response.result.actionIncomplete){
            botSays = response.result.fulfillment.speech;
            console.log('BotSays: ' + botSays);
            var twiml = new twilio.TwimlResponse();
            twiml.message(botSays);
```

```
            res.writeHead(200, {'Content-Type': 'text/xml'});
            res.end(twiml.toString());
        }
        else {
            getTrainInfo(
                response.result.metadata.intentName,
                response.result.parameters, res);
            }
        });

        requestAPIAI.on('error', function(error) {
            console.log(error);
        });
    requestAPIAI.end();
})
```

Мы используем флаг `actionIncomplete` в ответе агента Dialogflow, чтобы решить, использовать ли диалог в качестве ответа бота или вызвать модуль сервера для поиска информации о поездке.

2. Создайте новый модуль для получения информации о поездке:

```
function getTrainInfo(intent, parameters, res){
    if (intent == 'request_live_departures'){
        return getLiveDepartures(parameters.Station, res);
    }
    else if (intent == 'request_next_train'){
        return getNextTrain(parameters.fromStation,
        parameters.toStation, res);
    }
    else {
        var botSays = 'Working on it.';
        console.log('BotSays: ' + botSays);

        var twiml = new twilio.TwimlResponse();
        twiml.message(botSays);
        res.writeHead(200, {'Content-Type': 'text/xml'});
        res.end(twiml.toString());
    }
}
```

3. Создайте модули для получения актуального расписания поездов. Мы повторно используем код из файла *SendTrainNotification.js*, который применяли для отправки односторонних уведомлений.

```
function getLiveDepartures(source, res){
    var request = require('request');
    var url = 'http://transportapi.com/v3/uk/train/station/'
              + source +
                  '/live.json?
              app_id=ВАШ_ИДЕНТИФИКАТОР_ПРИЛОЖЕНИЯ&app_key=ВАШ_КЛЮЧ_ПРИЛОЖЕНИЯ';
    request(url, function (error, response, body) {
        if (response){
            var json = JSON.parse(body);
            if (json.departures){
                var botSays = summarize(json.departures.all, 5);
                console.log('BotSays: ' + botSays);

                var twiml = new twilio.TwimlResponse();
                twiml.message(botSays);
                res.writeHead(200, {'Content-Type': 'text/xml'});
                res.end(twiml.toString());
            } else {
                var botSays = 'No Departures found!'
                console.log('BotSays: ' + botSays);

                var twiml = new twilio.TwimlResponse();
                twiml.message(botSays);
                res.writeHead(200, {'Content-Type': 'text/xml'});
                res.end(twiml.toString());
            }
        } else {
            console.log('error:', error); // Вывести ошибку при возникновении
            var botSays = 'Error in fetching trains info. Sorry!';
            console.log('BotSays: ' + botSays);

            var twiml = new twilio.TwimlResponse();
            twiml.message(botSays);
            res.writeHead(200, {'Content-Type': 'text/xml'});
            res.end(twiml.toString());
        }
    });
}

function summarize(departures, n){
    var out = '';
    if (departures.length > 0){
        out = 'Live departures:\n';
        for (var i=0; i < n; i++){
```

```
            var service = departures[i];
            var serviceSummary = service.operator_name + ":" +
                service.destination_name + "@" +
                service.expected_departure_time;
            out += serviceSummary + "\n";
        }
    } else {
        out = 'There are no trains from ' + source;
    }
    return out;
}
```

Теперь добавьте функцию, позволяющую получить информацию о следующем поезде от указанной станции к месту назначения.

```
function getNextTrain(source, destination, res){
    var request = require('request');
    var url = 'http://transportapi.com/v3/uk/train/station/'
               + source + '/live.json?
               app_id=ВАШ_ИДЕНТИФИКАТОР_ПРИЛОЖЕНИЯ&app_key=ВАШ_КЛЮЧ_ПРИЛОЖЕНИЯ';
    request(url, function (error, response, body) {
        if (response){
            var json = JSON.parse(body);
            if (json.departures){
                var botSays = getNextTrainToDestination(
                                        destination,
                                 json.departures.all);
                console.log('BotSays: ' + botSays);

                var twiml = new twilio.TwimlResponse();
                twiml.message(botSays);
                res.writeHead(200, {'Content-Type': 'text/xml'});
                res.end(twiml.toString());
            } else {
                var botSays = 'No Departures found!'
                console.log('BotSays: ' + botSays);
                var twiml = new twilio.TwimlResponse();
                twiml.message(botSays);
                res.writeHead(200, {'Content-Type': 'text/xml'});
                res.end(twiml.toString());
            }
        } else {
            console.log('error:', error);
            // Вывести ошибку при возникновении
```

```
            var botSays = 'Error in fetching trains info. '
                          + 'Sorry!';
            console.log('BotSays: ' + botSays);
            var twiml = new twilio.TwimlResponse();
            twiml.message(botSays);
            res.writeHead(200, {'Content-Type': 'text/xml'});
            res.end(twiml.toString());
        }
    });
}
```

Теперь напишите функцию, позволяющую узнать о следующем поезде до места назначения.

```
function getNextTrainToDestination(destination, allDepartures){
    if (allDepartures){
        for (var i=0; i < allDepartures.length; i++){
            var service = allDepartures[i];
            if (service.destination_name ==
                getStationName(destination)){
                var serviceSummary = service.operator_name + ":"
                + service.destination_name + "@" +
                    service.expected_departure_time + "\n";
                return serviceSummary;
            }
        }
    }
    return null;
}

function getStationName(stationCode){
    if (stationCode == 'GLC'){
        return 'Glasgow Central';
    }
    else if (stationCode == 'EDB'){
        return 'Edinburgh';
    }
}
```

4. Сохраните и разместите файлы с измененным кодом на платформе Heroku.
5. Протестируйте чат-бота, отправив SMS. Проверьте работу расписания, отправив запрос с текстом `Live departures`. Бот запросит

название станции перед отображением информации об отправлениях в реальном времени.

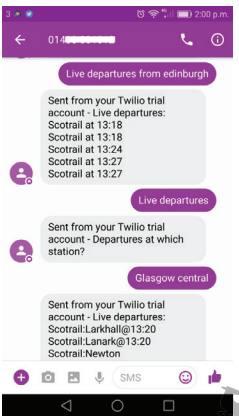


Рис. 5.40

Проверьте, выводятся ли последующие поезда, отправив запрос с текстом `Get next train`. Бот запросит станции отправления и назначения, после чего ответит на запрос.

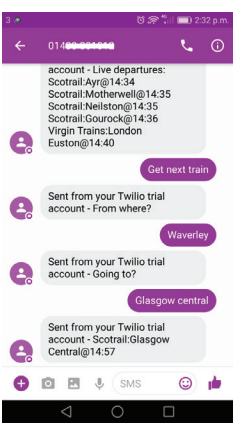


Рис. 5.41

Вы также можете запросить ту же информацию другими способами, чтобы проверить, насколько агент Dialogflow распознает цели пользователя.

На этом все. Вы создали двухстороннего SMS-чат-бота, который поддерживает ввод на естественном языке.

Заключение

Превосходно! Надеюсь, вы отлично провели время, изучая и разрабатывая транспортного чат-бота, предоставляющего пользователям полезную информацию о поездках. Помимо создания чат-бота, мы также изучили коммуникационный интерфейс Twilio, позволяющий надстроить чат-бота поверх платформы SMS. Надеюсь, вы пообщались со своим ботом посредством SMS-сообщений и поняли, насколько легко получить полезную информацию без посещения Интернета. Мы изучили несколько диалоговых задач, а также смогли реализовать некоторые из них. Тем не менее вы можете перейти к более сложным задачам, таким как планирование поездок, на основе модели чат-бота, которую мы построили в этой главе.

Мы также узнали, как задействовать чат-бота для обработки высказываний на естественном языке пользователя, создав агента Dialogflow и подключив его к SMS-боту. Этот прием вы можете использовать при разработке чат-ботов для других платформ. Теперь вы знаете, что существует много способов, с помощью которых модули можно подключать и воспроизводить. Я рекомендую вам протестировать их все и разработать свою систему в соответствии с потребностями.

Надеюсь, вам понравилось работать над чат-ботом для платформы SMS в этой главе. В следующих главах вы узнаете, как развернуть чат-боты на других интересных платформах социальных сетей, таких как Twitter и голосовых платформах типа Alexa и Google Assistant.

Ссылки

- Документация по Twilio: **www.twilio.com/docs/**.
- Документация по Dialogflow: **dialogflow.com/docs/**.

Глава 6. Поиск ресторана

В предыдущих главах мы рассматривали управление потоками диалогов в двух вариантах: в чат-ботах, созданных с нуля (глава 3) и с использованием таких графических инструментов, как Dialogflow (глава 4) и Chatfuel (глава 1). На платформах Dialogflow и Chatfuel поток диалога определялся разработчиком с помощью визуальных элементов (таких как формы), настраиваемых в специальном веб-редакторе. Тем не менее это не всегда лучший способ сообщить системе, как управлять диалогом. С другой стороны, мы разработали простой менеджер диалогов на языке Java. Управление диалогами может представлять собой трудную задачу в сложных беседах человека с чат-ботом. К счастью, у нас есть инструментальные средства, которые позволяют создавать модули управления диалогами, используя специально созданные для этой цели библиотеки кода.

В этой главе мы рассмотрим, как разработать модуль управления беседой с использованием существующей библиотеки – Bot Builder SDK. Во-первых, мы разберемся с платформой MS Bot, в которую входит компонент Bot Builder SDK. Мы установим необходимое программное обеспечение и библиотеки и научимся создавать чат-ботов с помощью SDK, тестировать их в эмуляторе и разворачивать в облаке. Затем мы узнаем о расширенных возможностях представления и устройствах, для которых может быть разработан поток диалога. Потом мы рассмотрим сервис Zomato для извлечения данных о ресторанах и интегрируем его функционал в чат-бота, созданного с помощью SDK Bot Builder. И наконец, разместим его в Skype.

В ходе этой главы вы:

- разберете основы MS Bot Framework;
- разработаете чат-бота с библиотекой Botbuilder Node.js;
- зарегистрируете бота с помощью Bot Framework;
- разместите бота в облаке;
- изучите типы сообщений и карт;
- научитесь управлять контекстом и потоком диалога;

- интегрируете в разработки интерфейс данных Zomoto;
- интегрируете бота в Skype.

Платформа MS Bot

MS Bot Framework – программное обеспечение Microsoft для разработки чат-ботов. В этом фреймворке представлены три продукта: SDK Bot Builder, портал Bot Framework и каналы. Bot Builder SDK – это инструментарий для создания чат-ботов. В нем есть библиотеки классов и код, которые представляют различные элементы диалога. Они позволяют ускорить процесс разработки чат-ботов, в отличие от их создания с нуля. Портал Bot Framework используется для регистрации бота с целью эффективного управления им. Также на портале доступно множество инструментов для аналитики и диагностики. Наконец, фреймворк обеспечивает единый подход к интеграции с несколькими каналами.

Существует огромное количество каналов, в которые вы можете интегрировать своего бота, включая Skype, Facebook Messenger, Kik, Telegram, Slack, MS Teams и Twilio. Вы также можете создать клиента веб-чата, используя портал, который может быть встроен в любой веб-сайт. В дополнение к описанным трем инструментам существует еще два, которые очень полезны в процессе разработки: эмулятор канала и инспектор каналов.

Эмулятор канала

Прежде чем мы начнем, нам нужно установить программное обеспечение, называемое **эмулятором канала**. Мы будем использовать его для эмуляции канала (например, Skype) и локального подключения к боту в целях разработки и тестирования. Вы сможете общаться с ботом, а также проверять отправленные и полученные сообщения, чтобы выявить какие-либо неполадки. Чтобы загрузить эмулятор канала, перейдите на страницу **github.com/Microsoft/BotFramework-Emulator/releases/tag/v3.5.36**. Загрузите версию, соответствующую операционной системе вашего компьютера.

Создание бота

Теперь рассмотрим процесс создания чат-бота. Мы воспользуемся библиотекой `botbuilder` и создадим бота с помощью платформы Node.js.

1. Создайте проект Node.js под названием `foodie-bot`.

```
> npm init
```

2. Установите две библиотеки, которые нам понадобятся.

```
> npm install botbuilder --save
> npm install restify -save
```

3. Создайте файл с именем *app.js*.
4. В файл *app.js* вставьте следующий код (из обучающих материалов Bot Framework):

```
var restify = require('restify');
var builder = require('botbuilder');

// Настройка сервера Restify
var server = restify.createServer();
server.listen(process.env.port || process.env.PORT || 3978, function() {
    console.log('%s listening to %s', server.name, server.url);
});

// Создание соединителя чат-бота для связи со службой Bot Framework
var connector = new builder.ChatConnector({
    appId: process.env.MICROSOFT_APP_ID,
    appPassword: process.env.MICROSOFT_APP_PASSWORD
});

// Отслеживание сообщений от пользователей
server.post('/foodiebot', connector.listen());

// Отправка их сообщений обратно . как попугай!
var bot = new builder.UniversalBot(connector, function (session) {
    session.send("You said: %s", session.message.text);
});
```

Обратите внимание, что есть два класса – `UniversalBot` и `ChatConnector`, которые предоставляет Node.js SDK фреймворка Bot Framework. `UniversalBot` – это класс, в котором мы определяем поток беседы, а класс `ChatConnector` соединяет бота с каналом чата. В предыдущем коде мы использовали метод `session.send()` для отправки текстовых сообщений в канал чата.

5. Сохраните файл.
6. Запустите эмулятор. В адресной строке введите показанный ниже адрес:
   `http://localhost:3978/api/messages`.
   На этом этапе вам не нужно указывать идентификатор приложения или пароль.
7. Эмулятор подключится к боту (запущен файлом *app.js*). Приложение начнет регистрировать сообщения в консоли, как показано ниже:

```
C:\Users\Srini\Dropbox\_Book\workspace\foodie-bot>node app.js
restify listening to http://[::]:3978
WARN: ChatConnector: receive - emulator running without security enabled.
ChatConnector: message received.
WARN: ChatConnector: receive - emulator running without security enabled.
ChatConnector: message received.
WARN: ChatConnector: receive - emulator running without security enabled.
```

Рис. 6.1

8. В текстовом поле эмулятора введите сообщение боту и отправьте его. Вы увидите, что бот продублировал сообщение.

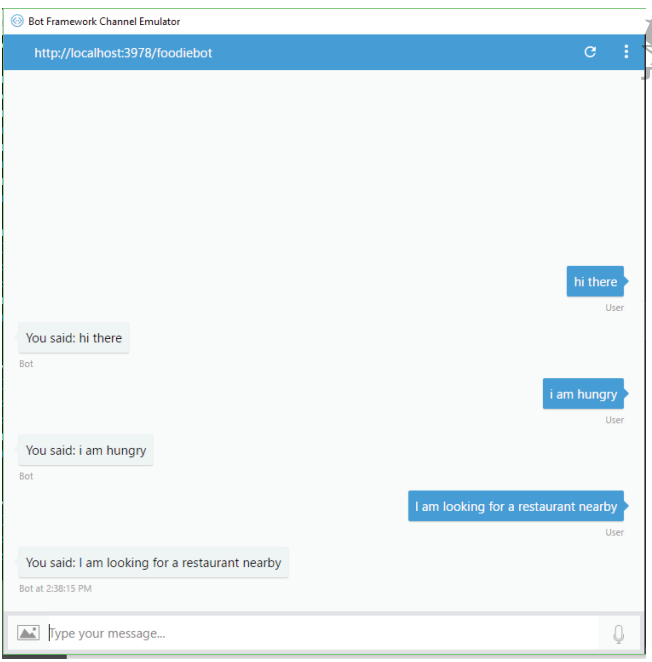


Рис. 6.2

9. Поздравляем! Вы только что создали своего первого бота с помощью фреймворка Bot Framework.

Развертывание бота

На данный момент у нас есть наш бот, работающий на локальной машине и взаимодействующий в эмуляторе канала. Теперь настроим его на разговор с пользователем через веб-сайт. Для этого нам нужно зарегистрировать бота в специальном каталоге Bot Framework. Выполните следующие действия.

1. Перейдите на страницу Bot Framework по адресу **dev.botframework.com**.
2. Создайте учетную запись, если это необходимо. Авторизуйтесь в системе.
3. Перейдите на вкладку **My bots** (Мои боты).
4. Щелкните мышью по кнопке **Create a bot** (Создать бота).

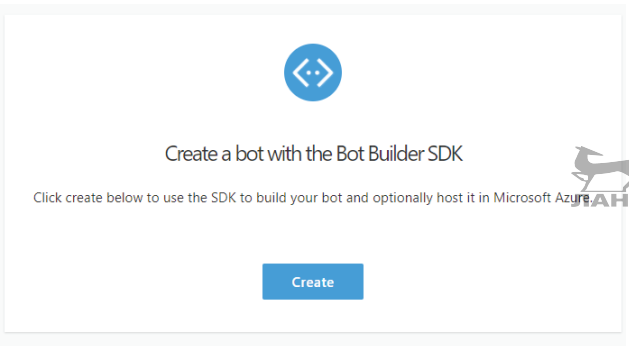


Рис. 6.3

5. Нажмите кнопку **Create** (Создать). Установите переключатель в положение **Register an existing bot built using Bot Builder SDK** (Зарегистрировать существующего бота, созданного с помощью SDK Bot Builder) и нажмите кнопку **OK** – рис. 6.4.
6. Прокрутите страницу вниз до пункта **Configuration** (Конфигурация). Щелкните мышью по ссылке **Create Microsoft App ID and Password** (Создать идентификатор и пароль приложения Microsoft) – рис. 6.5.
   Скопируйте идентификатор приложения и пароль и сохраните их.

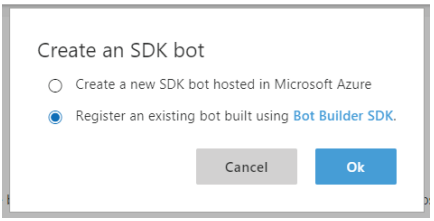


Рис. 6.4

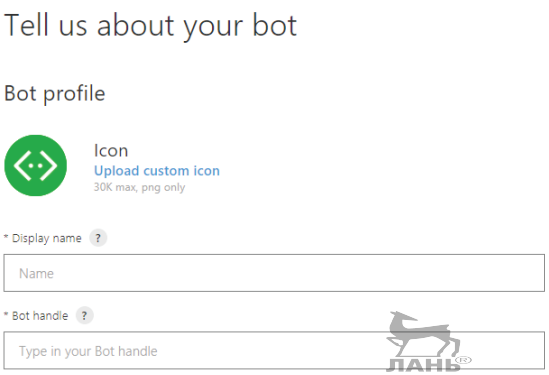


Рис. 6.5

7. Вернитесь к файлу *app.js* и замените переменные идентификатора и пароля приложения скопированными значениями. Сохраните файл. В качестве альтернативы мы можем установить настройки как параметры конфигурации.
8. Теперь вы готовы разместить бота в облаке и связать его с реестром Bot Framework. Для этого нам нужно создать файл `Procfile`. Создайте файл с именем `Procfile`, который сообщает Heroku, как запустить приложение:

```
web: node app.js
```

9. Создайте веб-приложение Heroku, выполнив следующую команду:

```
> heroku create foodie-bot-sj
```

10. Также нужен репозиторий Git для хранения кода нашего бота:

```
> git init
> git add .
> git commit -m initial-commit
```

11. Наконец, опубликуем код:

```
> git push heroku master
```

Теперь нам нужно указать идентификатор и пароль приложения в качестве переменных конфигурации в Heroku:

```
> heroku config:set MICROSOFT_APP_PASSWORD=<ПАРОЛЬ_ВАШЕГО_ПРИЛОЖЕНИЯ>
> heroku config:set MICROSOFT_APP_ID=<ИДЕНТИФИКАТОР_ВАШЕГО_ПРИЛОЖЕНИЯ>
```

12. Опубликовав код в облаке, мы можем протестировать его с помощью эмулятора канала. Введите URL-адрес бота вместе с идентификатором и паролем приложения и щелкните мышью по кнопке **CONNECT** (Подключиться).

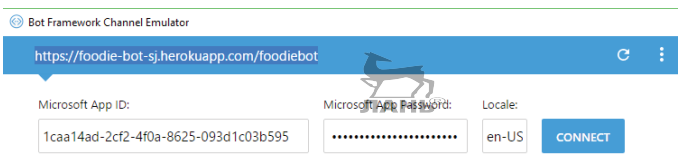


Рис. 6.6

13. После подключения введите свое сообщение и отправьте боту. Вы увидите, как бот дублирует сообщения, которые вы отправляете, – рис. 6.7.

Хорошая работа! Ваш бот опубликован в облаке и готов к развертыванию в Skype и других каналах, к чему мы вернемся позже в этой главе.

Другие типы сообщений

Теперь, когда мы настроили чат-бота и протестировали его в эмуляторе, рассмотрим другие варианты обмена сообщениями.

Отправка более одного сообщения одновременно

Во-первых, мы можем отправлять более одного сообщения одновременно. Поэтому, когда до чат-бота доходит очередь, он может отправлять несколько сообщений с помощью метода `session.send()`:

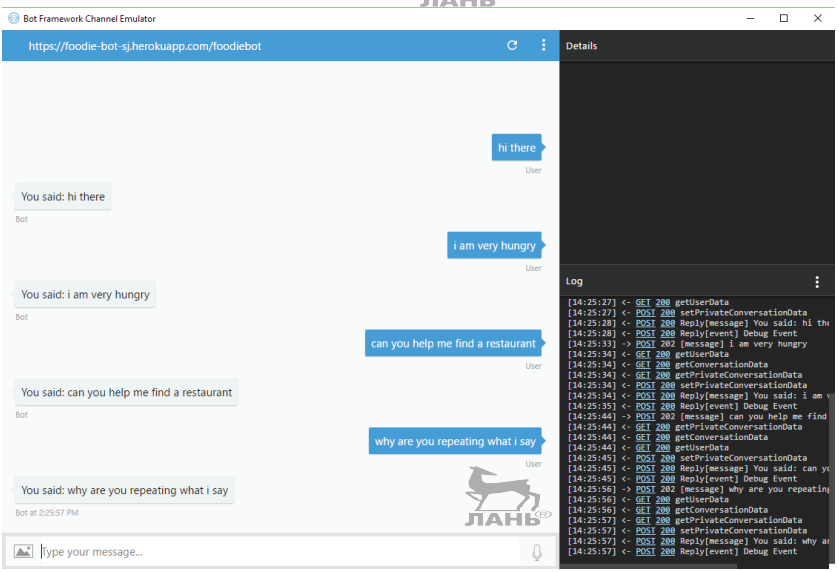


Рис. 6.7

```
var bot = new builder.UniversalBot(connector, [
    function (session) {
        session.send('Hello there!');
        session.send('Welcome to New India restaurant!');
        });
    }
]);
```

Запрос информации у пользователей

Чтобы запросить информацию у пользователей, используйте метод `builder.Prompts.text()`, как показано ниже:

```
var bot = new builder.UniversalBot(connector, [
    function (session) {
        builder.Prompts.text(session, 'Hi! What is your name?');
    },
    function (session, results) {
        session.endDialog('Hello ' + results.response + '!
                          My name is FoodieBot!');
    }
]);
```

Метод `builder.Prompts.text()` может использоваться для получения текстовых данных, таких как имена людей и названия городов. Ответы можно получить с помощью кода `results.response`. Используйте предыдущий код, заменив определение переменной `bot` в файле *app.js*.

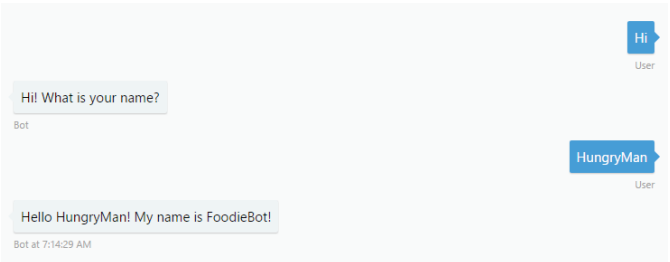


Рис. 6.8

Вы можете использовать числовые данные с помощью метода `builder.Prompts.number()` – рис. 6.9.

```
var bot = new builder.UniversalBot(connector, [
    function (session) {
        builder.Prompts.number(session, 'Booking a table!
                               For how many people?');
    },
    function (session, results) {
        session.endDialog('Ok. Looking for a table for ' +
                          results.response + ' people.');
    }
]);
```

Вы также можете попросить пользователей выбрать один из заданных параметров с помощью метода `builder.Prompts.choice()`.

```
var bot = new builder.UniversalBot(connector, [
    function (session) {
        builder.Prompts.choice(session, 'Booking a table!
        Any specific cuisine?', ['Indian', 'Chinese', 'Italian']);
    },
    function (session, results) {
        session.endDialog('Ok. Looking for a ' +
```

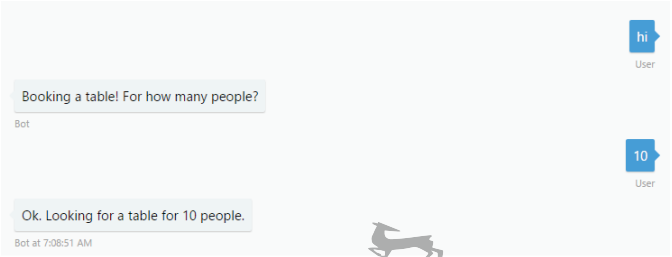


Рис. 6.9

```
        results.response.entity + ' restaurant.');
    }
]);
```

Обратите внимание, что метка варианта (например, `Indian`) хранится в сущности `results.response.entity`.

Рис. 6.10

Вы также можете предоставить выбор в показанном ниже формате, а не в массиве:

```
builder.Prompts.choice(session, 'Booking a table! Any specific cuisine?',
'Indian|Chinese|Italian');
```

И можете запросить дату и время и проанализировать различные входные данные, такие как `tomorrow at 2pm`, `Saturday at 8` или `next Friday`, используя класс `EntityRecognizer`, следующим образом:

```
builder.Prompts.time(session, “So when is the party?”);

session.dialogData.partyDate = builder.EntityRecognizer.resolveTime([results.
response]);
```

Развернутые сообщения

Теперь, когда мы знаем, как обслуживать сообщения и запросы, займемся внешним видом, добавив изображения и карты. Для этого мы применим карту Hero – шаблон для представления информации в расширенном формате с использованием изображений, URL-адресов и т. д. Ниже показан пример:

```
var bot = new builder.UniversalBot(connector, [
    function (session) {
        var msg = new builder.Message(session);
        msg.attachmentLayout(builder.AttachmentLayout.carousel)
        msg.attachments([
            new builder.HeroCard(session)
                .title(“Chennai Kitchen”)
                .subtitle(“Authentic South Indian Restaurant”)
                .text(“Great tasting dosas. 5 star reviews.”)
                .images([builder.CardImage.create(session,
                    ‘https://images.pexels.com/photos/221143/
                    pexels-photo-221143.jpeg?
                    w=940&h=650&auto=compress&cs=tinysrgb’)])
                .buttons([
                    builder.CardAction.imBack(session,
                    “book_table:chennai_kitchen”, “Book a table”)
                ]),
            new builder.HeroCard(session)
                .title(“Mumbai Tandoor”)
                .subtitle(“Best Indian Restaurant in town”)
                .text(“Amazing reviews!”)
                .images([builder.CardImage.create(session,
                    ‘https://images.pexels.com/photos/45844/
                    spices-white-pepper-nutmeg-45844.jpeg?
                    w=940&h=650&auto=compress&cs=tinysrgb’)])
                .buttons([
```

```
                    builder.CardAction.imBack(session,
                    "book_table:mumbai_tandoor", "Book a table")
                ])
        ]);
        session.send(msg)
    }
]);
```

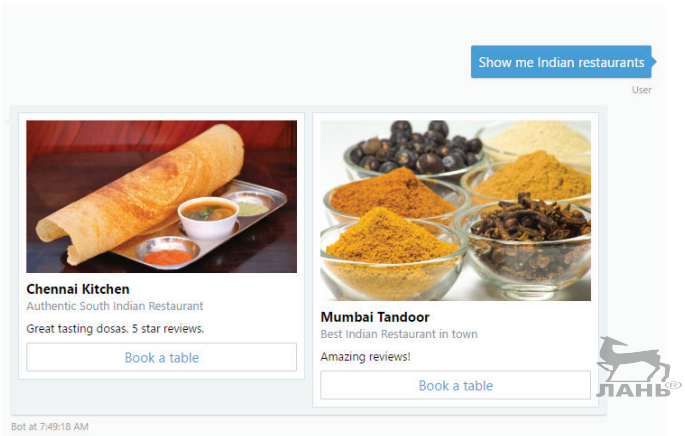


Рис. 6.11

Для каждой карты Hero можно указать название, подзаголовок, текст, изображение и кнопку ответа. В примере кнопки запрограммированы на отправку ответных сообщений боту с помощью метода `imBack()`. Также вы можете запрограммировать открытие веб-страницы с помощью метода `openUrl()` следующим образом:

```
builder.CardAction.openUrl(session, 'https://mumbaitandoor.com/bookTable','Book a table');
```

Существуют и другие типы карт – к примеру, карта миниатюр, адаптивная карта, звуковая карта и карта анимации. Полный список карт можно найти в документации фреймворка Bot Framework по адресу **docs.microsoft.com/en-us/bot-framework/nodejs/bot-builder-nodejs-send-rich-cards**.

Миниатюрные карты похожи на карты Hero, только меньше размером. Вы можете создавать миниатюрные карты с помощью класса `ThumbnailCard`, как показано ниже:

```
new builder.ThumbnailCard(session)
                .title("Chennai Kitchen")
                .subtitle("Authentic South Indian Restaurant")
                .text("Great tasting dosas. 5 star reviews.")
                .images([builder.CardImage.create(session,
                       'https://images.pexels.com/photos/221143/
                       pexels-photo-221143.jpeg?w=940&h=650&
                       auto=compress&cs=tinysrgb')])
                .buttons([
                    builder.CardAction.imBack(session,
                    "book_table:chennai_kitchen",
                    «Book a table»)
                ])
```

Запустите предыдущий код в эмуляторе.

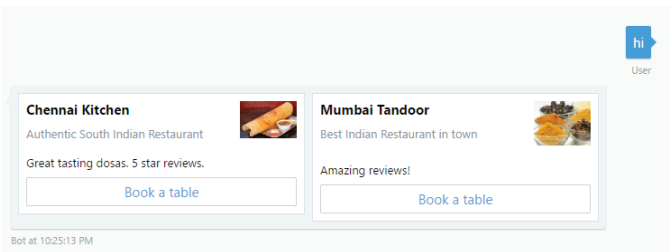


Рис. 6.12

Создадим карту для вывода изображений в формате GIF. Класс `AnimationCard` может использоваться для отображения анимированных изображений.

```
new builder.AnimationCard(session)
        .title('Microsoft Bot Framework')
        .subtitle('Animation Card')
        .image(builder.CardImage.create(session,
        'https://makeYourOwnCurry.com/curryAnimation.jpeg'))
        .media([
            { url: 'http://i.giphy.com/Ki55RUbOV5njy.gif' }
        ])
```

Карты аудио и видео могут использоваться для представления аудио- и видеоинформации:

```
// Видеокарта
new builder.VideoCard(session)
        .title('Chicken Tikka')
        .subtitle('from Sanjeev Kapoor Khazana')
        .text('Authentic Chicken Tikka recipe by Chef
            Harpal Singh Sokhi')
        .image(builder.CardImage.create(session,
            'https://commons.wikimedia.org/wiki/
            File:Chicken_Tikka_(1).jpg'))
        .media([
            { url: 'http://fakevideourl.com/makingofchickentikka.mp4' }
        ])
        .buttons([
            builder.CardAction.imBack(session, "order:chicken_tikka", "Order
Chicken Tikka")
])

// Аудиокарта
new builder.AudioCard(session)
   .title('Delicious Chicken Tikka')
   .subtitle('Must have at Mumbai Tandoor')
   .text('User')
   .image(builder.CardImage.create(session,
          'https://commons.wikimedia.org/wiki/
          File:Chicken_Tikka_(1).jpg'))
   .media([
         { url: 'http://fakeaudiourl.com/reviews1.wav' }
   ])
   .buttons([
   builder.CardAction.imBack(session, "order:chicken_tikka",
   «Order Chicken Tikka»)
])
```

Взглянем, как эти карты выглядят в эмуляторе, – рис. 6.13.

В дополнение есть специальная карта, называемая чековой книжкой, которая предоставляет информацию в формате квитанции. Ее можно использовать для представления подробного отчета с платежной информацией:

```
new builder.ReceiptCard(session)
    .title('James White')
    .facts([
        builder.Fact.create(session, '12345', 'Order Number'),
        builder.Fact.create(session, 'VISA 2392-****',
```

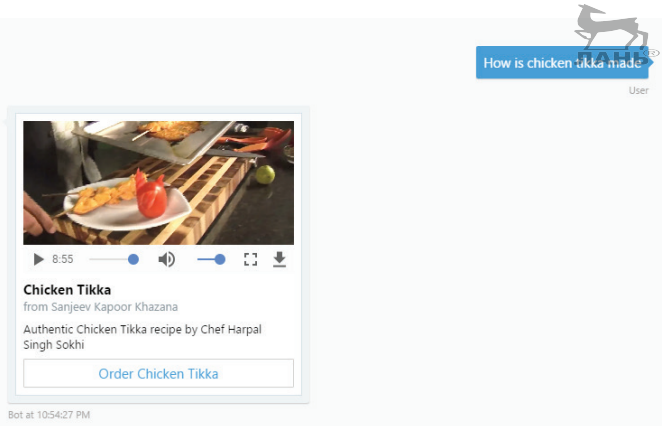


Рис. 6.13

```
                                'Payment Method')
    ])
    .items([
        builder.ReceiptItem.create(session, 'J 6.50', 'Chicken Tikka')
            .quantity(1),
        builder.ReceiptItem.create(session, 'J 5.00', 'Garlic Naan')
            .quantity(2)
    ])
    .tax('J 1.15')
    .total('J 12.65')
    .buttons([
        builder.CardAction.imBack(session, 'sendemail',
                                  'Send by email')
    ])
```

Запустите эмулятор – рис. 6.14.

Наконец, есть карта, которая может использоваться для аутентификации пользователя, отображая запрос входа в систему. Этот поток можно запустить с помощью карты `SignIn`.

```
new builder.SigninCard(session)
        .text('Mumbai Tandoor Login')
        .button('Login', 'https://mumbaitandoor.com/login')
        ]);
```

При нажатии на карту `SignIn` пользователь переходит на веб-страницу, где может пройти аутентификацию – рис. 6.15.

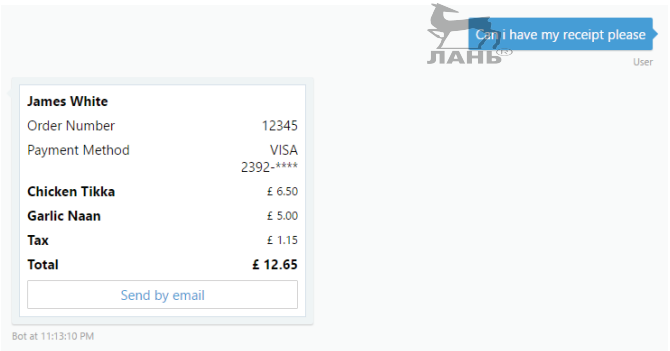


Рис. 6.14

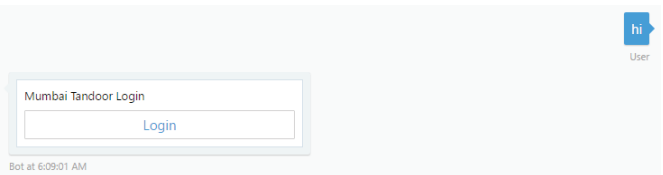


Рис. 6.15

Теперь, когда мы изучили карты, перейдем к реализации потока диалога.

Поток диалога

Теперь, когда у нас все настроено для тестирования чат-бота и мы исследовали различные способы предоставления информации пользователю, рассмотрим способы управления потоком диалога. Базовая доступная модель – это модель waterfall (водопад), где диалог состоит из последовательности шагов. Рассмотрим пример бронирования столика в ресторане, где диалог продолжается следующим образом: получить время бронирования, количество людей за столом и имя пользователя.

```
// Диалоги бота
var bot = new builder.UniversalBot(connector, [
    function (session) {
        session.send('Welcome to New India restaurant!');
        builder.Prompts.time(session, 'Table reservations.
                            What time?');
```

```
    },
    function (session, results) {
        session.dialogData.timeOfReservation =
        builder.EntityRecognizer.resolveTime([results.response]);
        builder.Prompts.number(session, “And how many people?”);
    },
    function (session, results) {
        session.dialogData.numberOfPeople = results.response;
        builder.Prompts.text(session, “And your name?”);
    },
    function (session, results) {
        session.dialogData.nameOnReservation = results.response;
        session.send(‘Great! Your reservation is booked!’);
    }
]);
```

Проверим в эмуляторе:

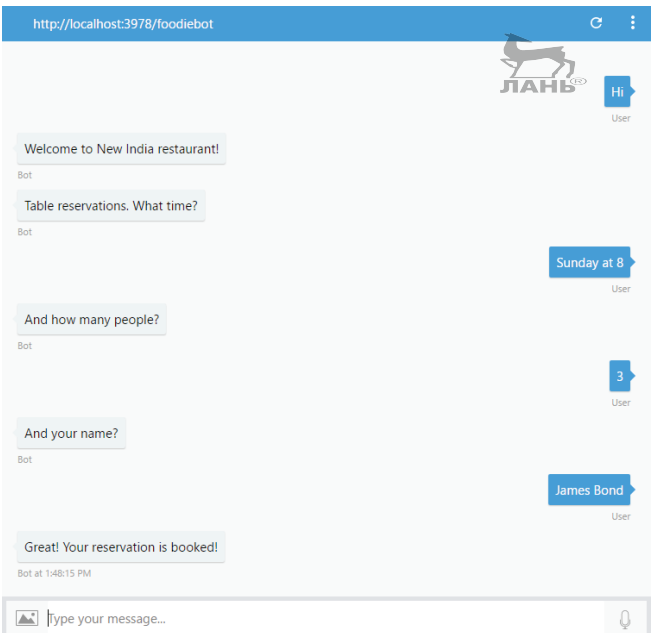


Рис. 6.16

Проанализируем код, чтобы понять, что происходит. Мы начали с построения бота с использованием класса `UniversalBot`. Когда мы

создаем бота, то определяем шаги waterfall-диалога как набор функций. Это основной диалог. Каждая функция является шагом в этом диалоге. На каждом шагу бот что-то сообщает или запрашивает у пользователя. В случае запроса он ожидает ответа от пользователя. Ответ хранится в `results.response`, который обновляется до состояния диалога, управляемого `session.dialogData`. Это присвоение происходит в последующих шагах, бот отправляет следующее высказывание или запрос для получения дополнительной информации. Как мы уже говорили ранее, существует множество способов получения и проверки информации.

Не всегда возможно отобразить весь диалог в виде массива функций. А если повторяются части диалога? Будучи программистами, мы обрабатываем эти ситуации, используя функции и методы. Метод должен быть четко определенным фрагментом кода, выполняющим конкретную задачу, и может быть вызван, если это требуется методом, например `main`. Возьмем, к примеру, задачу оплаты при размещении заказа. Если вы находитесь за столом или просите счет, вам придется делать платежи одним и тем же способом. Будет задан тот же набор вопросов: оплата картой или наличными, номер карты, имя на карте, код CVV и т. д. Представьте себе диалог об оплате между пользователем и ботом. Будет ли этот диалог использоваться более чем в одном сценарии? Было бы неплохо сохранить шаг диалога насчет платежа отдельно и вызывать процесс всякий раз, когда нужно произвести платеж. Это то, что мы можем выполнить с помощью метода `dialog()`.

Боту, который мы создаем с использованием класса `UniversalBot`, могут быть даны диалоговые навыки для выполнения множества задач, таких как платежи и перечисление товаров, с помощью метода `dialog()`. Затем их можно вызвать, когда это необходимо, из основного диалога. Каждый метод `dialog()` может использоваться для определения дополнительного диалога (поддиалога) и структурно будет независимым waterfall-диалогом. Теперь создадим основной диалог и внедрим в него два поддиалога с просьбой о заказе и запросом оплаты.

```
// Основной диалог
var bot = new builder.UniversalBot(connector, [
    function (session) {
        session.send("Welcome to New India restaurant.");
        session.beginDialog('askForOrder');
```

```
    },
    function (session) {
        session.beginDialog('askForPayment');
    },
    function (session) {
        session.send('Thanks for your order!');
        session.send('Order summary:
        ${session.conversationData.order}<br/>'+
            'Payment card number:
        ${session.conversationData.cardNumber}<br/>');
        session.endDialog();
    }
]);

// Запрос заказа
bot.dialog('askForOrder', [
    function (session) {
        builder.Prompts.text(session, 'Whats your order?');
    },
    function (session, results) {
        session.conversationData.order = results.response;
        session.endDialog();
    }
]);

// Запрос оплаты
bot.dialog('askForPayment', [
    function (session) {
        builder.Prompts.text(session, 'Whats the card number?');
    },
    function (session, results) {
        session.conversationData.cardNumber = results.response;
        builder.Prompts.text(session, 'Whats the CVV number?');
    },
    function (session, results) {
        session.conversationData.cardCVVNumber = results.response;
        session.send('Thanks for the payment!');
        session.endDialog();
    }
])
```

В предыдущем коде используются три диалога: основной, запрос заказа и запрос оплаты. В основном диалоге мы используем метод `session.beginDialog()` для вызова поддиалогов – рис. 6.17.

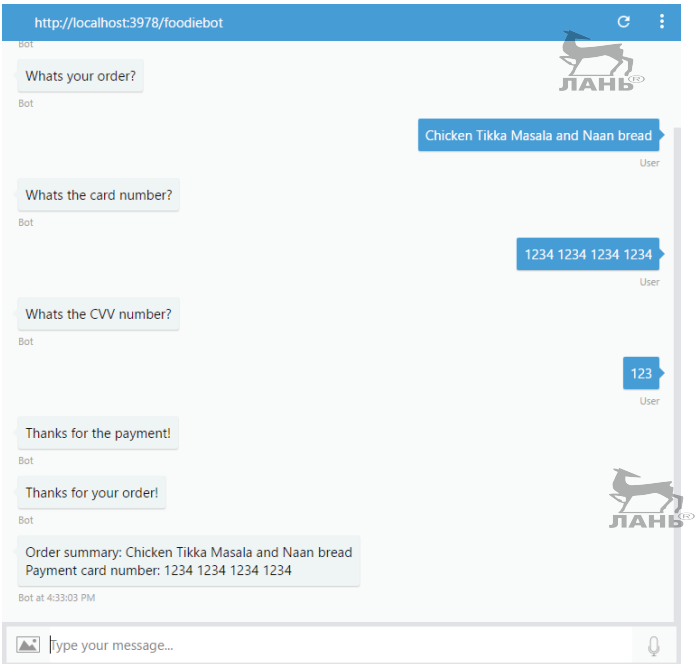


Рис. 6.17

Также обратите внимание, что мы использовали код `session.dialogData` для хранения информации из пользовательских высказываний. Но теперь мы используем метод `session.conversationData`. Мы рассмотрим разницу между ними позже.

На этом этапе вам нужно понять концепцию **стека диалога.** Вначале стек диалога содержит основной диалог. Поскольку поддиалоги вызываются из корня, они помещены поверх о0сновного диалога. Поддиалоги могут сами вызывать другие поддиалоги. В свою очередь, они помещены над ними. Когда поддиалог завершен, бот возвращает следующий диалог в стеке и продолжает делать это до тех пор, пока они не закончатся.

Реакция на пользовательские высказывания

Теперь у нас есть «диалог по умолчанию», который начинается одинаково, независимо от того, что говорит пользователь. Вы могли бы сказать привет, попросить помощь или отправить любое другое высказывание, и бот ответил бы приветственным сообщением. Другой

способ, с помощью которого начать диалог, основан на том, что говорит пользователь.

Теперь рассмотрим, как реагировать, если пользователь просит о помощи (help) в середине диалога.

```
bot.dialog('help', function (session, args, next) {
    session.endDialog("Hi. I can take food orders.<br/>
                       Say 'continue' to continue?");
})
.triggerAction({
    matches: /^help$/i,
});
```

Ответы на пользовательские высказывания можно сделать, добавив функцию triggerAction() с высказываниями, указанными как регулярные выражения в предложении matches. Добавьте показанный выше код в файл app.js и перезапустите сервер. Теперь диалог может вестись следующим образом:

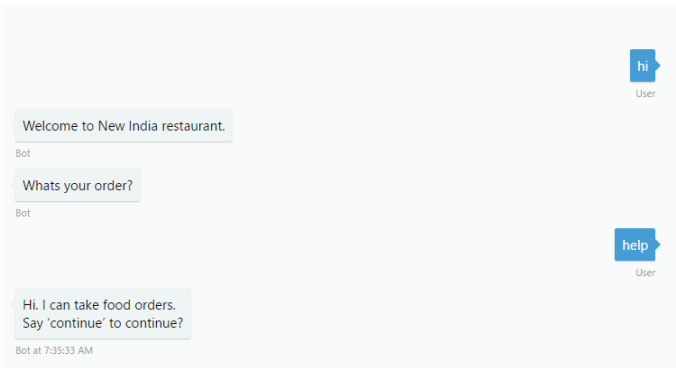


Рис. 6.18

Добавленный код позволяет боту отвечать на высказывание help пользователя. Обратите внимание, как здесь переопределяется текущее ожидание информации о заказе, которую ждет бот.

Существуют два других способа интерпретации пользовательских высказываний: специфический распознаватель и использование служб NLU, таких как LUIS. Сначала попробуем специфический распознаватель. Добавьте в код бота следующий код:

```
bot.recognizer({
    recognize: function (context, done) {
```

```
        var intent = { score: 0.0 };
        if (context.message.text) {
            switch (context.message.text.toLowerCase()) {
                case 'help':
                    intent = { score: 1.0, intent: 'get-help' };
                    break;
                case 'goodbye':
                    intent = { score: 1.0, intent: 'say-goodbye' };
                    break;
            }
        }
        done(null, intent);
    }
});
```

И создайте соответствующие поддиалоги для сущностей:

```
bot.dialog('help', [
    function (session) {
        session.send('I can help you look for a
                      restaurant or order a takeaway!');
        session.endDialog();
    }
]).triggerAction({
    matches: 'get-help'
});

bot.dialog('goodbye', [
    function (session) {
        session.send('Goodbye now!');
        session.endConversation();
    }
]).triggerAction({
    matches: 'say-goodbye'
});
```

Хотя шаблоны высказываний могут быть заданы с параметрами совпадений для каждого поддиалога, лучше организовать их как цели с использованием глобального распознавателя для всех поддиалогов. Это необходимо, чтобы не дублировать регулярные выражения. Как только цели определены, их можно использовать для запуска соответствующих поддиалогов, как показано ниже – рис. 6.19:

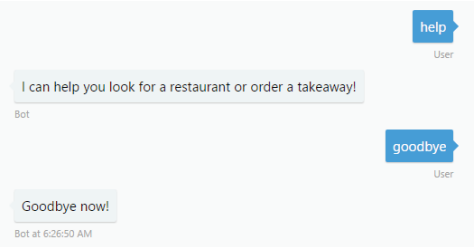


Рис. 6.19

Примечание. LUIS – это служба машинного обучения, разработанная корпорацией Microsoft для распознавания естественного языка. Он похож на Google Dialogflow и Amazon Lex. Любой из этих инструментов может быть интегрирован в бота для распознавания естественного языка, если другие распознаватели не работают должным образом. Чтобы получить дополнительные сведения, обратитесь к официальной документации по адресу **docs.microsoft.com/en-us/bot-framework/nodejs/bot-builder-nodejs-recognize-intent-luis**.

Сохранение целостности контекста

Обработка пользовательских высказываний с использованием поддиалогов может привести к выходу диалога из контекста:

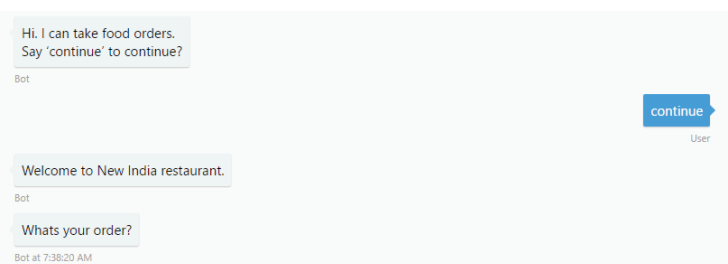


Рис. 6.20

Что случится с диалогом, если пользователь набрал слово `continue`, как указано в справочном сообщении? Продолжится ли диалог? Нет. Бот, кажется, полностью забыл, что он делал раньше. Это связано с тем, что стек диалога очищается при обработке пользовательских высказываний. Очищать стек нужно, когда пользователь хочет сменить тему диалога, но не тогда, если пользователь просит помощи.

Существует способ сохранения контекста, даже когда пользователи прерывают вопросы и замечания. Это можно сделать, добавив параметр `onSelectAction` к поддиалогу, который вызывается. Это приведет к сохранению диалогового стека:

```
bot.dialog('help', function (session, args, next) {
    session.endDialog("Hi there. I can take food orders.");
})
.triggerAction({
    matches: /^help$/i,
    onSelectAction: (session, args, next) => {
        session.beginDialog(args.action, args);
    }
});
```

Суть решения – определить, где должен очищаться стек диалога, а где это не нужно делать. Например, если пользователь запрашивает помощь, лучше не очищать контекст, поскольку запрос помощи может быть связан с контекстом. Однако если пользователь переключается на другую задачу (например, запрашивает бронирование столика, когда заказывает еду), нужно очистить контекст, поскольку нецелесообразно возвращаться к еде после того, как столик был забронирован.

Переключение контекста

Возникают случаи, когда пользователь хочет переключиться с одной задачи на другую. Тогда не нужно сохранять стек диалога. Если не использовать параметр `onSelectAction`, мы можем стереть стек диалога. Тем не менее рекомендуется сообщить пользователю, что бот собирается отказаться от текущей задачи, чтобы заняться следующей. Это можно сделать, используя параметр `confirmPrompt` в методе `triggerAction()`:

```
bot.dialog('askForOrder', [
    .
])
.triggerAction({
    matches: /^order food$/i,
    confirmPrompt: "Your food order will be abandoned. Are you sure?"
});

// Поиск ресторана
bot.dialog('searchRestaurant', [
```

```
    function (session) {
        session.send('Searching for a restaurant!');
        builder.Prompts.text(session, 'Where?');
    },
    function (session, results) {
        session.conversationData.searchLocation = results.response;
        session.endDialog();
    }
])
    .triggerAction({
    matches: /^restaurant search$/i,
    confirmPrompt: 'Your restaurant search task
    will be abandoned. Are you sure?'
});
```

Добавив метод `triggerAction()` в диалог `askForOrder`, мы сможем отвечать на запросы пользователей, желающих заказать еду в любой момент диалога. Однако бот будет предупреждать их о том, что любая другая выполняемая задача (например, резервирование столика) будет остановлена.

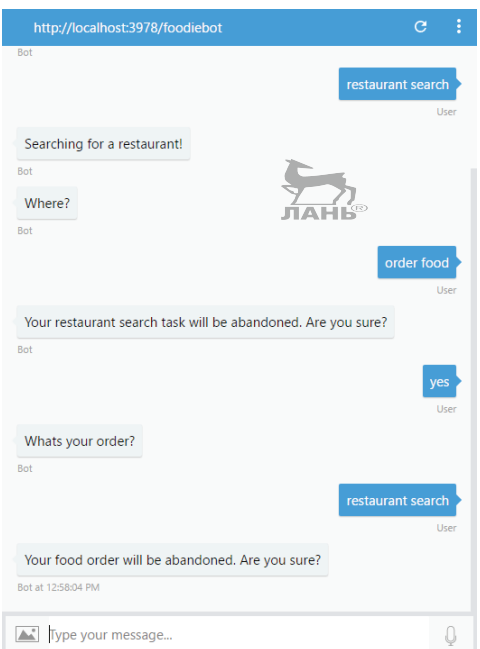


Рис. 6.21

Обратите внимание, что когда пользователь переключается с одной задачи на другую, требуется подтверждение.

Распознавание контекста естественного языка

Данный способ также может понадобиться при распознавании контекста. Например, бот может предоставить пользователю меню, когда он обращается за помощью во время заказа. Рассмотрим, как высказывания могут обрабатываться с учетом контекста. Для этого нужно создать вспомогательный диалог и добавить его к существующему, чтобы поддиалог мог запускаться, когда высказывание пользователя соответствует предоставленному шаблону.

```
// Заказ
bot.dialog('askForOrder', [
    function (session) {
        builder.Prompts.text(session, 'Whats your order?');
    },
    function (session, results) {
        session.conversationData.order = results.response;
        session.endDialog();
    }
])
.beginDialogAction('orderHelpAction', 'orderHelp',
                   { matches: /^help$/i });

// Контекстная подсказка для заказа
bot.dialog('orderHelp', function(session, args, next) {
    var msg = "You can order for Chicken Tikka Masala,
    Paneer Butter Masala, Naan and Briyani.";
    session.endDialog(msg);
})
```

Обратите внимание, как метод `beginDialogAction()` используется для привязки поддиалога `orderHelp`. Поддиалог `orderHelp` запускается, когда пользователь говорит `help` при заказе пищи, – рис. 6.22.

В предыдущем диалоге просьба `help` не имеет стандартного ответа. Вместо этого выводится контекстный.

Завершение общения

Наконец, нужно закончить диалог, когда задачи завершены. Это делается путем информирования пользователя о завершении задач,

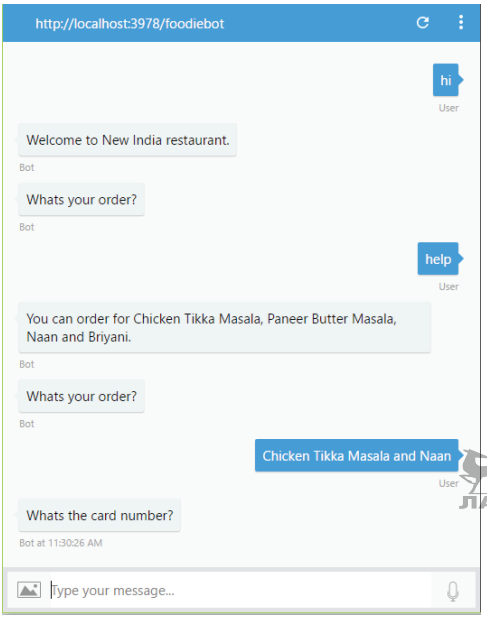


Рис. 6.22

очистке стека диалога и сбросе объекта `session.conversationData`. Для этого используйте метод `session.endConversation()`. Итак, переписываем наш основной диалог с помощью метода `session.endConversation()`:

```
// Основной диалог
var bot = new builder.UniversalBot(connector, [
    function (session) {
        session.send("Welcome to New India restaurant.");
        session.beginDialog('askForOrder');
    },
    function (session) {
        session.beginDialog('askForPayment');
    },
    function (session) {
        session.send('Thanks for your order!');
        session.send('Order summary:
${session.conversationData.order}<br/>'+
            'Payment card number.
        ${session.conversationData.cardNumber}<br/>');
```

```
        session.endConversation();
    }
]);
```

Вы также можете установить диалог по умолчанию, который вызывается, когда пользователь прощается (Goodbye) и заканчивает диалог:

```
bot.dialog('endConversation', [
    session.endConversation("Goodbye!")
])
.endConversationAction(
    "endTasks", "Ok. Goodbye.",
    {
        matches: /^goodbye$/i,
        confirmPrompt: "Cancelling current task. Are you sure?"
    }
);
```

Теперь рассмотрим, как мы можем сохранить контекст беседы.

Состояние диалога

Состояние диалога можно сохранить в виде пар ключ/значение. Для этого доступны четыре хранилища данных. Они размещаются внутри объекта session.

- dialogData: весь разговор делится на диалоги (основной диалог и поддиалоги, вызванные функцией beginDialog()). Состоянием каждого диалога можно управлять по отдельности. Это осуществляется с помощью метода session.dialogData, который мы использовали в предыдущих примерах. В нем хранятся данные, относящиеся к текущему поддиалогу, и каждый поддиалог имеет свою собственную копию dialogData. Когда диалог завершается (т. е. выполняется функция endDialog()) и удаляется из стека диалога, эти данные удаляются.
- conversationData: хранит данные, относящиеся ко всему диалогу, и разделяется между всеми участниками (то есть пользователями) в диалоге. Очищается при завершении диалога или при выполнении функции endConversation(). Доступ к этим данным можно получить с помощью метода session.conversationData.

- `privateConversationData`: здесь хранятся данные, относящиеся ко всему диалогу, как `conversationData`, но отдельно для каждого участника диалога. Данные очищаются при завершении диалога или при выполнении функции `endConversation()`. Доступ к этим данным можно получить с помощью метода `session.privateConversationData`.
- `userData`: здесь могут быть сохранены личные данные, относящиеся к пользователю. Это неизменные данные на протяжении всего общения. Такие данные, как имя пользователя, возраст, пол, адрес, номер телефона, электронная почта и информация о транзакции, могут храниться здесь и использоваться в диалогах. Доступ к этим данным можно получить с помощью метода `session.userData`.

Итак, мы рассмотрели ряд концепций в наборе инструментов Bot Builder с рабочими примерами. Мы рассмотрели, как настроить основной диалог и передать общение в поддиалоги. Мы изучили, как обрабатывать пользовательские высказывания локально и глобально, а также использовать различные объекты данных для хранения данных пользователей и диалогов. Теперь перейдем к созданию бота с данными о ресторанах.

Начало работы с Zomato

Во-первых, нам нужно создать учетную запись на сайте Zomato и получить ключ API.

1. Перейдите на страницу **developers.zomato.com/api**.

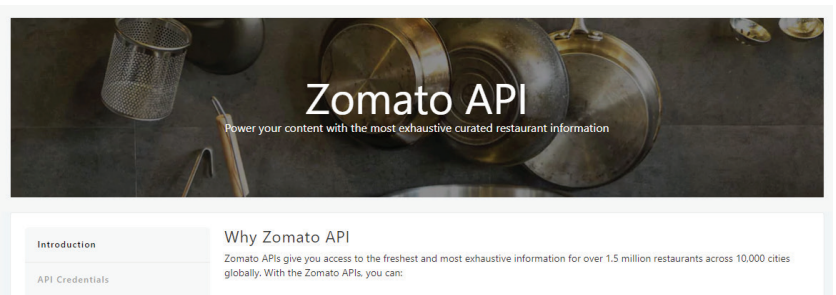


Рис. 6.23

2. Щелкните мышью по кнопке **Generate API Key** (Создать ключ API).

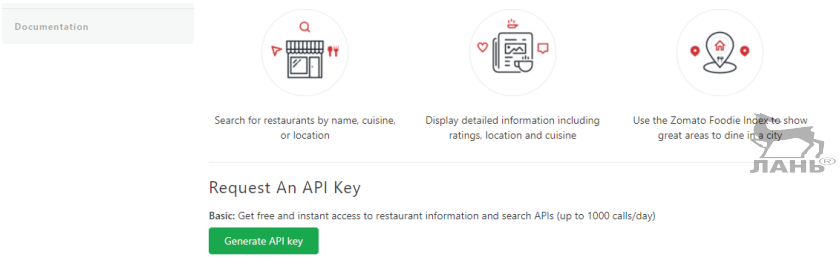


Рис. 6.24

3. Зарегистрируйтесь.

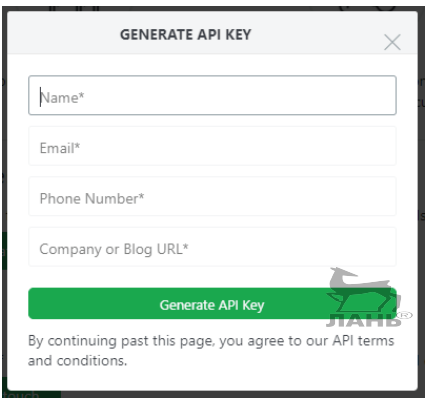


Рис. 6.25

4. Подтвердите свой адрес электронной почты и снова нажмите кнопку **Generate API Key** (Создать ключ API) – рис. 6.26.
5. Скопируйте ключ API.

Получение данных из Zomato

Основной URL-адрес для извлечения данных – **https://developers.zomato.com/api/v2.1/**. Каждый запрос, отправленный на сервер Zomato, должен содержать в заголовке ключ API:

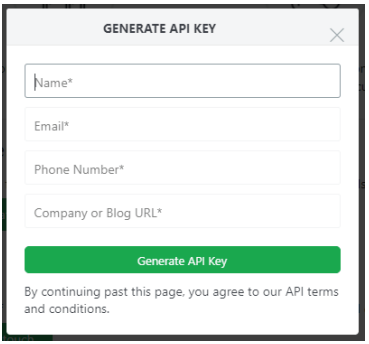


Рис. 6.26

```
curl -X GET --header "Accept: application/json" --header "user-key:ВАШ_КЛЮЧ_API"
"https://developers.zomato.com/api/v2.1/categories"
```

В предыдущем HTTP-вызове `GET` дескриптор извлекает различные категории доступных ресторанов. Категории включают доставку, на вынос и перекус внутри. Ниже представлен список доступных дескрипторов.

Дескриптор	**Описание**
`/categories`	*Категории ресторанов: перекус внутри, на вынос, доставка и многое другое*
`/cities`	*Запрос информации о городах по всему миру. Получите идентификатор города в системе Zomato с помощью этого запроса, чтобы использовать его в поисковых запросах. Поиск осуществляется по названию или координатам (широта и долгота) города*
`/collections`	*Запрос списка ресторанов с учетом идентификатора города или координат широта/долгота*
`/cuisines`	*Запрос списка кухонь в городе*
`/establishments`	*Запрос списка категорий ресторанов, доступных в городе*
`/locations`	*Запрос идентификатора местоположения для поиска. Это идентификаторы в разных масштабах: город, подзона, зона, ориентир и т. д. Используется для поиска ресторанов*
`/search`	*Поиск ресторанов в определенном месте по идентификатору, кухне, категории и т. д.*
`/reviews`	*Запрос отзывов о конкретном ресторане*

Теперь, когда мы изучили данные, которые служба Zomato предоставляет о ресторанах, перейдем к созданию бота с помощью Zomato и Bot Builder SDK.

Служба для поиска ресторанов

Создадим чат-бота, который умеет искать рестораны на основе целей и предпочтений пользователя. Начнем с создания модулей Node.js для получения данных на основе пользовательских настроек с сервера Zomato. Создайте файл с именем *zomato.js*. Добавьте модуль запросов в библиотеки Node.js, выполнив в консоли следующую команду:

```
> npm install request -save

В файл zomato.js добавьте следующий код:

var request = require('request');

var baseURL = 'https://developers.zomato.com/api/v2.1/';
var apiKey = 'ВАШ_КЛЮЧ_API';

var catergories = null;
var cuisines = null;

getCategories();
getCuisines(76);
```

Замените значение `ВАШ_КЛЮЧ_API` своим ключом Zomato. Создадим функции для получения списка категорий и кухонь при запуске. Эти запросы не нужно выполнять, когда пользователь запрашивает поиск в ресторане, потому что эта информация в значительной степени статична.

```
function getCuisines(cityId){
    var options = {
        uri: baseURL + 'cuisines',
        headers: {
            'user-key': apiKey
        },
        qs: {'city_id':cityId},
        method: 'GET'
        }
    var callback = function(error, response, body) {
            if (error) {
                console.log('Error sending messages: ', error)
            } else if (response.body.error) {
```

```
                console.log('Error: ', response.body.error)
            } else {
                console.log(body);
                cuisines = JSON.parse(body).cuisines;
            }
        }
    request(options,callback);
}
```

С помощью предыдущего кода вы получите список кухонь, доступных в определенном городе (обозначенном идентификатором в системе Zomato). Добавим код для идентификации списка категорий:

```
function getCategories(){
    var options = {
        uri: baseURL + 'categories',
        headers: {
            'user-key': apiKey
        },
        qs: {},
        method: 'GET'
        }
    var callback = function(error, response, body) {
            if (error) {
                console.log('Error sending messages: ', error)
            } else if (response.body.error) {
                console.log('Error: ', response.body.error)
            } else {
                categories = JSON.parse(body).categories;
            }
        }
    request(options,callback);
}
```

Теперь, когда у нас есть основные функции, напишите код поиска ресторана:

```
function getRestaurant(cuisine, location, category){
    var cuisineId = getCuisineId(cuisine);
    var categoryId = getCategoryId(category);

    var options = {
        uri: baseURL + 'locations',
```

```
        headers: {
            'user-key': apiKey
        },
        qs: {'query':location},
        method: 'GET'
        }
    var callback = function(error, response, body) {
            if (error) {
                console.log('Error sending messages: ', error)
            } else if (response.body.error) {
                console.log('Error: ', response.body.error)
            } else {
                console.log(body);
                locationInfo = JSON.parse(body).location_suggestions;
                search(locationInfo[0], cuisineId, categoryId);
            }
        }
    request(options,callback);
}

function search(location, cuisineId, categoryId){
    var options = {
        uri: baseURL + 'search',
        headers: {
            'user-key': apiKey
        },
        qs: {'entity_id': location.entity_id,
            'entity_type': location.entity_type,
            'cuisines': [cuisineId],
            'categories': [categoryId]},
        method: 'GET'
    }
    var callback = function(error, response, body) {
            if (error) {
                console.log('Error sending messages: ', error)
            } else if (response.body.error) {
                console.log('Error: ', response.body.error)
            } else {
                console.log('Found restaurants:')
                var results = JSON.parse(body).restaurants;
                console.log(results);
            }
        }
    request(options,callback);
}
```

Показанный выше код позволяет искать рестораны по расположению, кухне и категории. Например, вы можете найти индийские рестораны с возможностью доставки заказа в районе Ньюингтон города Эдинбурга. Теперь нам нужно интегрировать этот код в чат-бота. Создадим отдельный файл index.js. Сначала основной код:

```
var restify = require('restify');
var builder = require('botbuilder');
var request = require('request');

var baseURL = 'https://developers.zomato.com/api/v2.1/';
var apiKey = 'ВАШ_КЛЮЧ_API';

var catergories = null;
var cuisines = null;

getCategories();
//setTimeout(function(){getCategoryId('Delivery')}, 10000);

getCuisines(76);
//setTimeout(function(){getCuisineId('European')}, 10000);

// Настройка сервера Restify
var server = restify.createServer();
server.listen(process.env.port || process.env.PORT || 3978, function() {
    console.log('%s listening to %s', server.name, server.url);
});

// Подключение чата к службе Bot Framework
var connector = new builder.ChatConnector({
    appId: process.env.MICROSOFT_APP_ID,
    appPassword: process.env.MICROSOFT_APP_PASSWORD
});

// Ожидание сообщений от пользователей
server.post('/foodiebot', connector.listen());
```

Добавьте код диалога бота для осуществления поиска ресторана. Спроектируем бота, способного запросить кухню, категорию и местоположение, прежде чем приступать к поиску ресторана:

```
var bot = new builder.UniversalBot(connector, [
    function (session) {
```

```
        session.send("Hi there! Hungry? Looking for a restaurant?");
        session.send("Say 'search restaurant' to start searching.");
        session.endDialog();
    }
]);

// Поиск ресторана
bot.dialog('searchRestaurant', [
    function (session) {
        session.send('Ok. Searching for a restaurant!');
        builder.Prompts.text(session, 'Where?');
    },
    function (session, results) {
        session.conversationData.searchLocation = results.response;
        builder.Prompts.text(session, 'Cuisine? Indian, Italian, or anything
else?');
    },
    function (session, results) {
        session.conversationData.searchCuisine = results.response;
        builder.Prompts.text(session, 'Delivery or Dine-in?');
    },
    function (session, results) {
        session.conversationData.searchCategory = results.response;
        session.send('Ok. Looking for restaurants');
        getRestaurant(session.conversationData.searchCuisine,
                      session.conversationData.searchLocation,
                      session.conversationData.searchCategory,
                      session);
    }
])
    .triggerAction({
    matches: /^search restaurant$/i,
    confirmPrompt: 'Your restaurant search task will be abandoned. Are you sure?'
});
```

Обратите внимание, что мы вызываем функцию `getRestaurant()` с четырьмя параметрами. Три из них – это те, которые мы уже определили: кухня, местоположение и категория (`cuisine`, `location` и `category`). Мы добавили еще одно: `session`. В нем передается указатель сеанса, который может использоваться для отправки сообщений в эмулятор, когда данные готовы. Обратите внимание, как меняются функции `getRestaurant()` и `search()`:

```
function getRestaurant(cuisine, location, category, session){
    var cuisineId = getCuisineId(cuisine);
    var categoryId = getCategoryId(category);
    var options = {
        uri: baseURL + 'locations',
        headers: {
            'user-key': apiKey
        },
        qs: {'query':location},
        method: 'GET'
        }
    var callback = function(error, response, body) {
            if (error) {
                console.log('Error sending messages: ', error)
            } else if (response.body.error) {
                console.log('Error: ', response.body.error)
            } else {
                console.log(body);
                locationInfo = JSON.parse(body).location_suggestions;
                search(locationInfo[0], cuisineId, categoryId, session);
            }
        }
    request(options,callback);
}

function search(location, cuisineId, categoryId, session){
    var options = {
        uri: baseURL + 'search',
        headers: {
            'user-key': apiKey
        },
        qs: {'entity_id': location.entity_id,
             'entity_type': location.entity_type,
             'cuisines': [cuisineId],
             'category': categoryId},
        method: 'GET'
    }
    var callback = function(error, response, body) {
            if (error) {
                console.log('Error sending messages: ', error)
            } else if (response.body.error) {
                console.log('Error: ', response.body.error)
            } else {
                console.log('Found restaurants:')
                console.log(body);
```

```
                //var results = JSON.parse(body).restaurants;
                //console.log(results);
                var resultsCount = JSON.parse(body).results_found;
                console.log('Found:' + resultsCount);
                session.send('I have found ' + resultsCount +
                            ' restaurants for you!');
                session.endDialog();
            }
        }
    request(options,callback);
}
```

Как только результаты получены, бот отвечает с помощью метода `session.send()` и завершает диалог рис. 6.27 и 6.28.

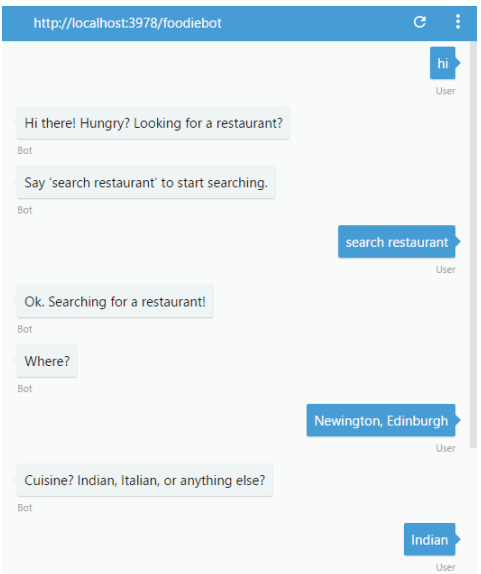


Рис. 6.27

Теперь, когда результаты поиска могут быть получены, оформим их более привлекательно, используя карты. Для этого нам нужна функция, которая может принимать результаты поиска и превращать их в массив карт.

```
function presentInCards(session, results){
    var msg = new builder.Message(session);
```

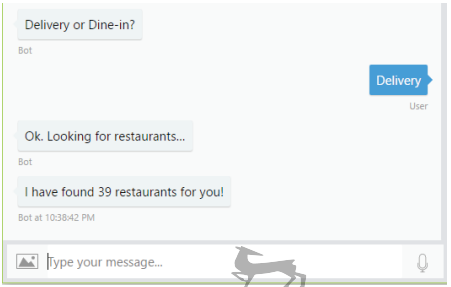


Рис. 6.28

```
    msg.attachmentLayout(builder.AttachmentLayout.carousel)
    var heroCardArray = [];
    var l = results.length;
    if (results.length > 10){
        l = 10;
    }
    for (var i = 0; i < l; i++){
        var r = results[i].restaurant;
        var herocard = new builder.HeroCard(session)
            .title(r.name)
            .subtitle(r.location.address)
            .text(r.user_rating.aggregate_rating)
            .images([builder.CardImage.create(session, r.thumb)])
            .buttons([
                builder.CardAction.imBack(session,
                        "book_table:" + r.id,
                        "Book a table")
            ]);
        heroCardArray.push(herocard);
    }
    msg.attachments(heroCardArray);
    return msg;
}
```

И мы вызываем эту функцию из функции `search()`:

```
function search(location, cuisineId, categoryId, session){
    var options = {
        uri: baseURL + 'search',
        headers: {
            'user-key': apiKey
        },
```

```
        qs: {'entity_id': location.entity_id,
            'entity_type': location.entity_type,
            'cuisines': [cuisineId],
            'category': categoryId},
        method: 'GET'
    }
    var callback = function(error, response, body) {
            if (error) {
                console.log('Error sending messages: ', error)
            } else if (response.body.error) {
                console.log('Error: ', response.body.error)
            } else {
                console.log('Found restaurants:')
                console.log(body);
                var results = JSON.parse(body).restaurants;
                var msg = presentInCards(session, results);
                session.send(msg);
                session.endDialog();
            }
        }
    request(options,callback);
}
```

Вот как выглядит результат – рис. 6.29.

Подключение к Skype

Теперь, когда чат-бот для поиска ресторанов готов, опубликуйте в облаке Heroku, как это делали ранее. Не забудьте отредактировать файл *Procfile*, поскольку нам нужно запускать файл *index.js*, а не *app.js*. Установив чат-бота в качестве веб-приложения на платформе Heroku, мы все настроили для развертывания бота в Skype и других каналах.

1. Перейдите в панель управления на платформе Bot Framework по адресу **dev.botframework.com/bots** – рис. 6.30.
2. Выберите своего бота. Вы увидите, что ваш бот уже подключен к двум каналам: **Skype и Web Chat**, – рис. 6.31.
3. Вы также можете просмотреть клиентский интерфейс чата в правой части страницы. Если он не отображается, откройте его, нажав кнопку **Test** (Тест). Интерфейс похож на эмулятор. Поэкспериментируйте, сказав боту `Hi`, – рис. 6.32.

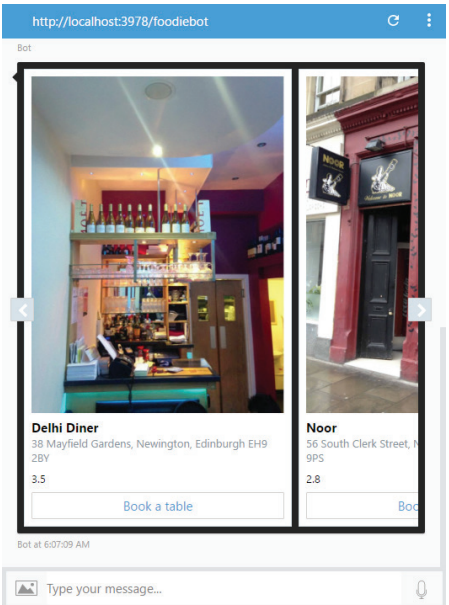


Рис. 6.29

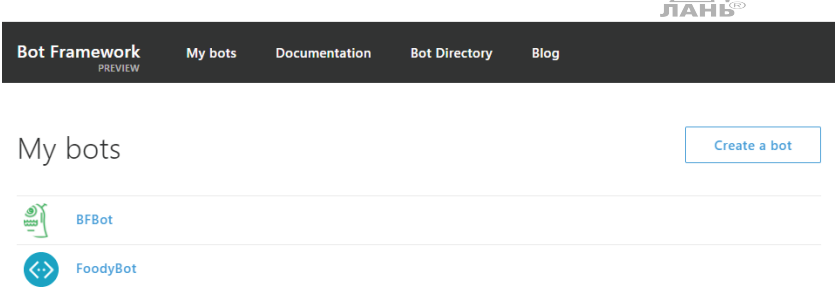


Рис. 6.30

Диалог с ботом должен происходить как в облаке, а не локальном хосте.

4. Щелкните мышью по ссылке **Skype**,– рис 6.33.
5. Щелкните мышью по ссылке **Add to Contacts** (Добавить в контакты) – рис. 6.34.
6. Щелкните мышью по ссылке **Open Skype** (Открыть Skype) – рис. 6.35.
7. Добавьте бота в свои контакты и начните беседу – рис. 6.36, 6.37.

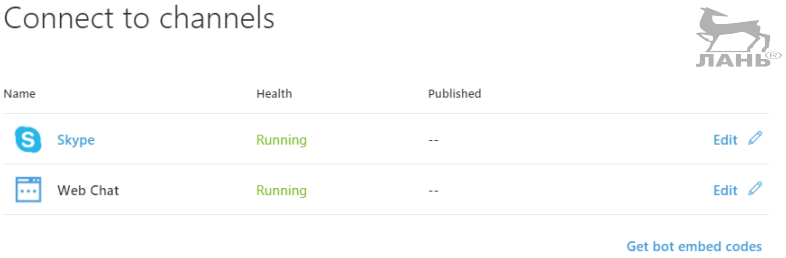


Рис. 6.31

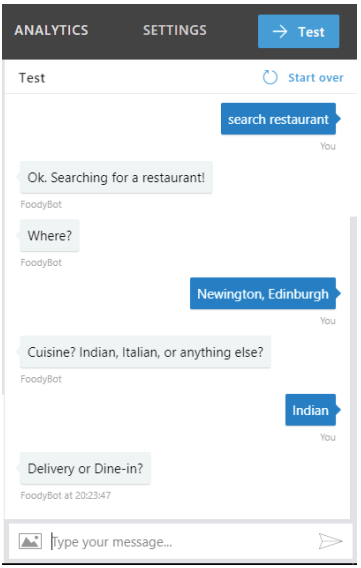


Рис. 6.32

Поздравляю! Вы создали бота для мессенджера Skype. Теперь, когда работа с Bot Builder SDK позади, изучите все другие способы, о которых я рассказал, но мы не экспериментировали. Добавьте диалог профилирования пользователя, в котором бот узнает о предпочтениях пользователя. Расширьте диалог поиска по ресторанам, включив вопросы, основанные на обзорах, рейтингах и ценовом диапазоне, которые задаются, когда рестораны найдены и нужна дальнейшая сортировка. Добавьте поддержку на естественном языке, чтобы пользователь мог легко оперировать задачами.

Рис. 6.33

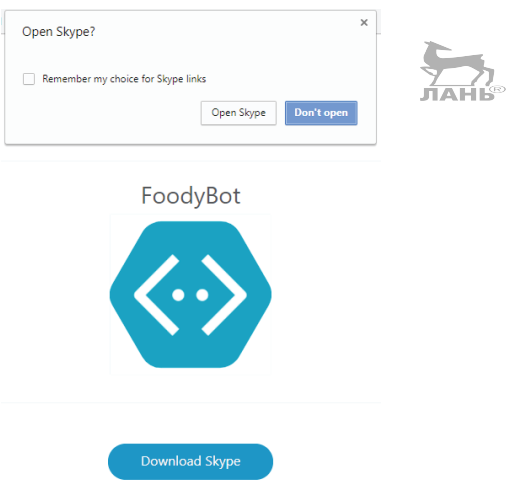


Рис. 6.34

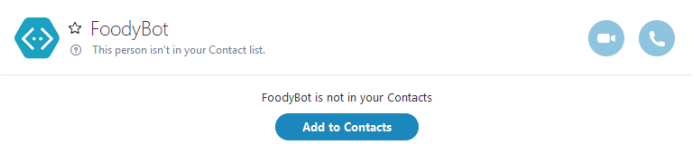


Рис. 6.35

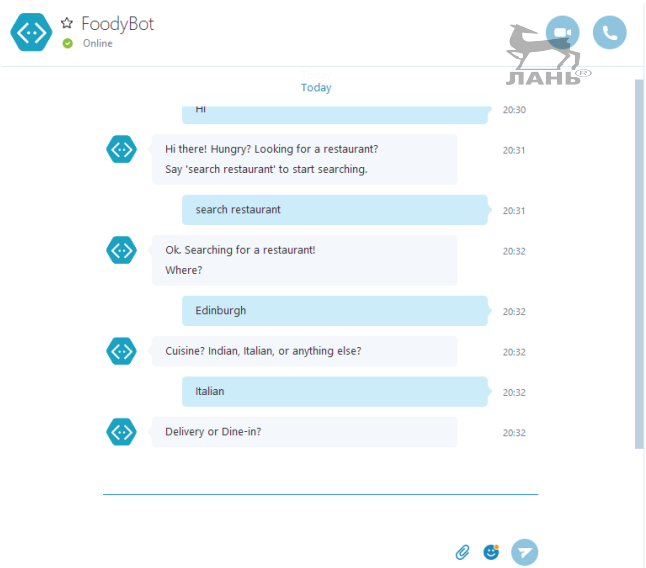


Рис. 6.36

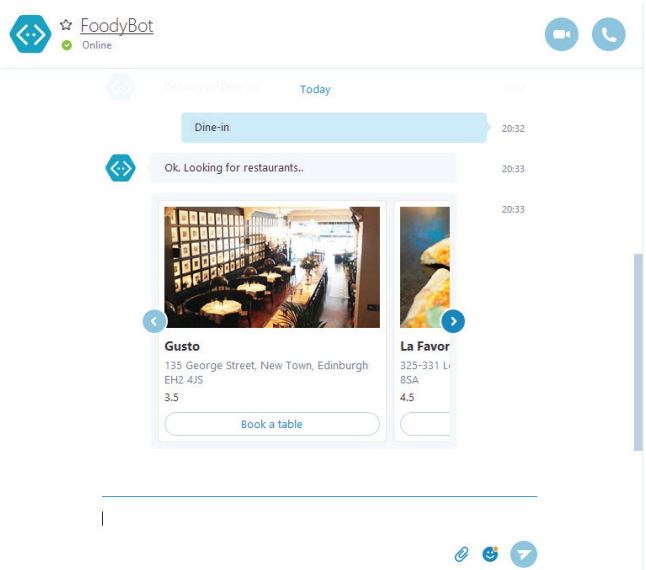


Рис. 6.37

Заключение

В предыдущих главах вы многое изучили и узнали о том, как создать чат-бота, используя различные инструменты. К ним относятся среды разработки, такие как Chatfuel, инструменты для распознавания естественного языка, такие как Dialogflow и SDK для конкретных каналов, например Messenger SDK. Однако когда дело доходит до кодирования потока для управления диалогом, мы либо использовали инструменты на основе форм, либо выстраивали его с нуля. Но есть еще один способ. Платформа Bot Framework содержит инструмент SDK Bot Builder, который может использоваться для разработки диалоговых модулей, управляющих тем, как проистекает диалог между ботом и пользователем. SDK моделирует элементы диалогов удобным способом, позволяя разработчикам быстро и легко создавать чаты. Это предоставляет разработчикам бóльшую гибкость, чем простое перетаскивание, и экономит время и силы, по сравнению с разработкой бота с нуля.

В этой главе вы узнали, как разработать бота с помощью SDK Bot Builder и развернуть его на канале Skype. В следующей главе мы перейдем к более продвинутому боту, который сильно отличается от тех, которые мы изучили до сих пор, – Twitter-боту!

Ссылки

- Документация по Microsoft Bot Builder SDK для Node.js: **docs.microsoft.com/en-us/azure/bot-service/nodejs/bot-builder-nodejs-overview**.

Глава 7. Новостной бот

До сих пор мы изучали, как создавать чаты и развертывать их на различных платформах обмена сообщениями, таких как Facebook Messenger, Skype и даже SMS. Мы рассмотрели различные способы использования инструментов, таких как Dialogflow, в контексте создания чат-ботов. В этой главе мы рассмотрим, как чат-боты можно интегрировать в Twitter, социальную платформу. Существует много подобных социальных медиаплатформ, таких как Facebook и LinkedIn, которые настроены на различные контексты и рынки. То, что мы рассмотрим в этой главе, концептуально относится ко всем этим платформам.

Twitter – это онлайн-платформа новостей и социальных сетей, где пользователи могут публиковать сообщения, которые называются твитами. В сети насчитывается около 328 млн активных пользователей по всему миру: около 317 млн активных пользователей ежедневно взаимодействуют с примерно 500 млн твитов каждый день (по данным Statista на 2017 г.). Twitter используется предприятиями как инструмент для вхождения бренда, анонсов продуктов и обслуживания клиентов.

В этой главе мы рассмотрим, как создавать чат-ботов с поддержкой сети Twitter, чтобы твиты можно было публиковать и отвечать на них автоматически и быстро. Мы создадим новостного бота на платформе Twitter, который регулярно обновляет текущие новости на своей странице. Он также будет разделять подход к своим подписчикам, отправляя им новости по темам, которые их интересуют. Сначала мы рассмотрим API Twitter и создадим основные модули для твитов, поиска и ретвитов. Затем мы исследуем источники данных новостей по всему миру. Далее мы разработаем простого бота, который публикует новости на своей странице. Наконец, мы создадим персонализированного новостного бота, который будет анализировать твиты, публикуемые пользователями, об интересующих их темах и создавать новостные твиты для них.

В ходе этой главы вы:

- освоите API Twitter;
- разработаете бота, который отслеживает хештеги;
- создадите Twitter-бота, который публикует твиты и ретвиты;
- интегрируете NewsAPI и рейтинговые твиты;
- интегрируете MongoDB для отслеживания интересов пользователей;
- создадите диалогового бота, уточняющего интересы пользователей и персонализирующего новости;
- научитесь использовать планировщик Heroku, чтобы публиковать персонализированные новости с регулярной очередностью.

Начало работы с приложением Twitter

Для начала рассмотрим платформу разработчика на сайте Twitter. Начнем с создания приложения Twitter и последующего изучения того, как мы можем отбирать по интересам и публиковать новостные заметки подписчикам.

1. Авторизуйтесь на сайте Twitter по адресу **www.twitter.com**. Если у вас нет учетной записи Twitter, создайте ее.
2. Перейдите на страницу **apps.twitter.com** – панель управления приложениями Twitter.
3. Щелкните мышью по кнопке **Create New App** (Создать новое приложение).

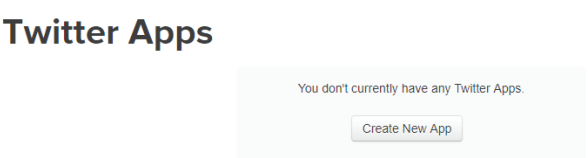


Рис. 7.1

4. Создайте приложение, заполнив форму и указав название, описание и веб-сайт (полный URL-адрес). Установите флажок согласия с условиями разработки и щелкните мышью по кнопке **Create your Twitter application** (Создать свое приложение Twitter) – рис. 7.2.
5. Вы увидите панель управления вашего приложения. Взгляните на вкладки – рис. 7.3.
6. Перейдите на вкладку **Keys and Access Tokens** (Ключи и токены доступа) – рис. 7.4.

Create an application

Application Details

Name *

MyNewsBot

Your application name. This is used to attribute the source of a tweet and in user-facing authorization screens. 32 characters max.

Description *

Personalised News Tweeter Bot

Your application description, which will be shown in user-facing authorization screens. Between 10 and 200 characters max.

Website *

http://chatbotgu.ru

Your application's publicly accessible home page, where users can go to download, make use of, or find out more information about your application. This fully-qualified URL is used in the source attribution for tweets created by your application and will be shown in user-facing authorization screens.
(If you don't have a URL yet, just put a placeholder here but remember to change it later.)

Callback URL

Where should we return after successfully authenticating? OAuth 1.0a applications should explicitly specify their oauth_callback URL on the request token step, regardless of the value given here. To restrict your application from using callbacks, leave this field blank.

Рис. 7.2

MyNewsBot

Test OAuth

Details | Settings | Keys and Access Tokens | Permissions

Personalised News Tweeter Bot

http://chatbotgu.ru

Organization

Information about the organization or company associated with your application. This information is optional.

Organization	None
Organization website	None

Рис. 7.3

MyNewsBot

Test OAuth

Details | Settings | Keys and Access Tokens | Permissions

Application Settings

Keep the "Consumer Secret" a secret. This key should never be human-readable in your application.

Consumer Key (API Key)	
Consumer Secret (API Secret)	
Access Level	Read and write (modify app permissions)
Owner	ChatbotGuru
Owner ID	

Рис. 7.4

7. Скопируйте *ключ клиента* (**Consumer Key (API Key)**) и *секрет клиента* (**Consumer Secret (API Secret)**) и сохраните их.
8. Прокрутите страницу вниз до раздела **Your Access Token** (Ваш токен доступа).

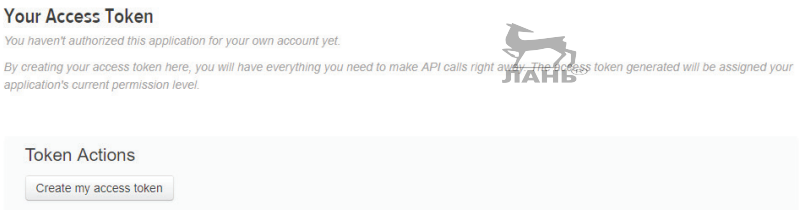


Рис. 7.5

9. Нажмите кнопку **Create my access token** (Создать токен доступа), чтобы создать новый токен для вашего приложения, – рис. 7.6.

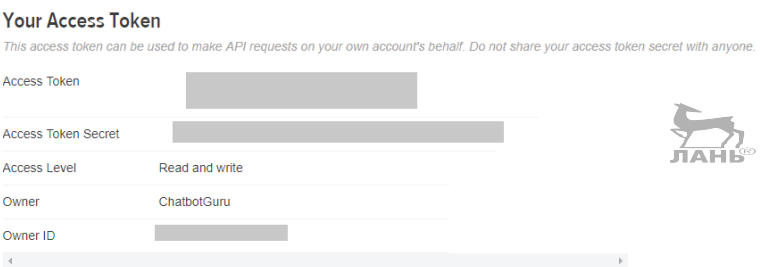


Рис. 7.6

10. Скопируйте токен доступа (**Access Token**) и секрет токена доступа (**Access Token Secret**) и сохраните их.

Теперь у нас есть все необходимые данные для создания приложения Twitter.

Создание Twitter-бота

Разработаем простого Twitter-бота. Этот бот будет отслеживать твиты и отбирать те, которые содержат определенный хештег. Все твиты с заданным хештегом будут выводиться в консоль. Это очень простой бот, который позволит нам начать работу. В следующих разделах мы рассмотрим более сложные аспекты.

1. Перейдите в корневой каталог и создайте новую программу *Node.js*, выполнив команду `npm init`.

```
C:\Users\Srini\Dropbox\_Book\workspace\twitterbot>npm init
This utility will walk you through creating a package.json file.
It only covers the most common items, and tries to guess sensible defaults.

See `npm help json` for definitive documentation on these fields
and exactly what they do.

Use `npm install <pkg> --save` afterwards to install a package and
save it as a dependency in the package.json file.

Press ^C at any time to quit.
name: (twitterbot)
version: (1.0.0)
description: my news bot
entry point: (index.js)
test command:
git repository:
keywords:
author:
license: (ISC)
About to write to C:\Users\Srini\Dropbox\_Book\workspace\twitterbot\package.json:

{
  "name": "twitterbot",
  "version": "1.0.0",
  "description": "my news bot",
  "main": "index.js",
  "scripts": {
    "test": "echo \"Error: no test specified\" && exit 1"
  },
  "author": "",
  "license": "ISC"
}
```

Рис. 7.7

2. Выполните команду `npm install twitter --save`, чтобы установить библиотеку Twitter Node.js, – рис. 7.8.
   Выполните команду `npm install request --save`, чтобы установить библиотеку Request. Мы воспользуемся ею, чтобы выполнять HTTP-запросы `GET` к источникам новостей.

```
C:\Users\Srini\Documents\workspace\SMSBot>npm install twilio
npm WARN package.json smsbot@1.0.0 No repository field.
npm WARN package.json smsbot@1.0.0 No README data
twilio@3.6.1 node_modules\twilio
├── deprecate@1.0.0
├── scmp@0.0.3
├── rootpath@0.1.2
├── xmlbuilder@9.0.1
├── q@2.0.3 (weak-map@1.0.5, pop-iterate@1.0.1, asap@2.0.6)
└── request@2.81.0 (aws-sign2@0.6.0, tunnel-agent@0.6.0, forever-agent@0.6.1, oauth-sign@0.8.2, is-type
darray@1.0.0, caseless@0.12.0, safe-buffer@5.1.1, stringstream@0.0.5, aws4@1.6.0, isstream@0.1.2, json-
stringify-safe@5.0.1, extend@3.0.1, performance-now@0.2.0, uuid@3.1.0, qs@6.4.0, combined-stream@1.0.5,
 mime-types@2.1.16, tough-cookie@2.3.2, form-data@2.1.4, hawk@3.1.3, http-signature@1.1.1, har-validato
r@4.2.1)
├── moment@2.18.1
├── jsonwebtoken@7.4.2 (ms@2.0.0, lodash.once@4.1.1, xtend@4.0.1, jws@3.1.4, joi@6.10.1)
└── lodash@4.0.0
```

Рис. 7.8

3. Просмотрите содержимое файла *package.json*, расположенного в корневом каталоге:

```
{
  “name”: “twitterbot”,
  “version”: “1.0.0”,
  “description”: “my news bot”,
  “main”: “index.js”,
  “scripts”: {
  “test”: “echo \”Error: no test specified\” && exit 1”
},
  “author”: “”,
  “license”: “ISC”,
  “dependencies”: {
    “request”: “^2.81.0”,
    «twitter»: «^1.7.1»
  }
}
```

4. Создайте файл *index.js* со следующим кодом:

```
//index.js

var TwitterPackage = require(‘twitter’);
var request = require(‘request’);

console.log(“Hello World! I am a twitter bot!”);

var secret = {
    consumer_key: ‘ВАШ_КЛЮЧ_КЛИЕНТА’,
    consumer_secret: ‘ВАШ_СЕКРЕТ_КЛИЕНТА’,
    access_token_key: ‘КЛЮЧ_ВАШЕГО_ТОКЕНА_ДОСТУПА’,
    access_token_secret: ‘СЕКРЕТ_ВАШЕГО_ТОКЕНА_ДОСТУПА’
}

var Twitter = new TwitterPackage(secret);
```

В показанном выше коде укажите свои ключи и секреты для сохранения в соответствующих переменных. Нам пока не нужен пакет `request`, мы задействуем его позднее.

5. Теперь создадим *слушателя хештега* для отслеживания твитов с указанным хештегом:

```
// Поток Twitter

var hashtag = '#brexit'; // Укажите любой хештег для отслеживания вместо #brexit
console.log('Listening to:' + hashtag);
Twitter.stream('statuses/filter', {track: hashtag}, function(stream) {
    stream.on('data', function(tweet) {
        console.log('Tweet:@' + tweet.user.screen_name +
                    '\t' + tweet.text);
        console.log('------')
    });
    stream.on('error', function(error) {
        console.log(error);
    });
});
```

Замените значение `#brexit` хештегом, который вы хотите прослушивать. Укажите популярный хештег, чтобы опробовать код в действии.

6. Запустите файл *index.js* с помощью команды `node index.js`.
7. Вы увидите поток твитов от пользователей Twitter по всему миру, которые использовали указанный вами хештег.

```
C:\Users\Srini\Dropbox\_Book\workspace\twitterbot>node index.js
Hello World! I am a twitter bot!
Listening to:#brexit
Tweet:@belanisiya       RT @weloveeconomics: The UK economy grew faster after joining the EU but popu
lism (no facts!) made people believe the opposite.…
------
Tweet:@BazzieSmith      RT @jlivingstone100: Brexit support fading it seems -'Marketplace did not enc
ounter any farmers at the show who admitted voting for Brex…
------
Tweet:@frankietaggart   RT @brexitcountdow1: Brexit is 13850 hours away. #brexit
------
Tweet:@pm_kristin       RT @laute_europaeer: Ist nach Abschluss der #Brexit -Verhandlungen ein 2. #Re
ferendum über das Abkommen zwischen #UK und #EU notwendig?
htt…
------
Tweet:@GillianRAdams    RT @nickreeves9876: Shocking how seldom the question of Freedom of Movement i
s couched in terms of the loss of our Right to live &amp; work in…
------
Tweet:@hpw_llp  RT @TheLawSociety: What is the European Court of Justice and why does it matter? A #B
rexit Q&amp;A https://t.co/C0Vi6J0fm4 https://t.co/8ankRoV…
------
Tweet:@h11billies       RT @LeedsEurope: Leeds for Europe Stop Brexit Day of Action! 16 Sep - Rally,
Question Time &amp; Social with @RCorbettMEP @emmyzen &amp; mor…
------
Tweet:@KarenMc10        RT @trevdick: Britain 2017 with Brexit looming is peddling backwards to 1971.
..
Memories are not made of this...
#Brexit is Barmy https://t…
```

Рис. 7.9

Мои поздравления! Вы создали своего первого Twitter-бота. Мы воспользуемся модулем слушателя хештегов для разработки более сложного бота.

Обзор SDK Twitter

В предыдущем разделе мы рассмотрели, как отслеживать твиты по хештегам. Теперь изучим программное обеспечение SDK Twitter, чтобы рассмотреть возможности, которыми мы можем наградить нашего Twitter-бота.

Обновление статуса

Вы также можете обновить статус в своей ленте Twitter, используя следующий код модуля *обновления статуса*:

```
tweet ('I am a Twitter Bot!', null, null);

function tweet(statusMsg, screen_name, status_id){
    console.log('Sending tweet to: ' + screen_name);
    console.log('In response to:' + status_id);
    var msg = statusMsg;
    if (screen_name != null){
        msg = '@' + screen_name + ' ' + statusMsg;
    }
    console.log('Tweet:' + msg);
    Twitter.post('statuses/update', {
            status: msg
        }, function(err, response) {
            // при возникновении ошибки во время твита
            if (err) {
                console.log('Something went wrong while TWEETING.');
                console.log(err);
            }
            else if (response) {
                console.log('Tweeted!!!');
                console.log(response)
            }
    });
}
```

Закомментируйте код слушателя хештегов и вместо него добавьте показанный выше код обновления статуса, а потом запустите файл. Бот опубликует твит в соответствии с кодом – рис 7.10.

Помимо публикации твитов в ленте, вы также можете ответить на другой твит (или обновление статуса). Аргумент `screen_name` используется для создания ответа. `tweet. screen_name` – это имя

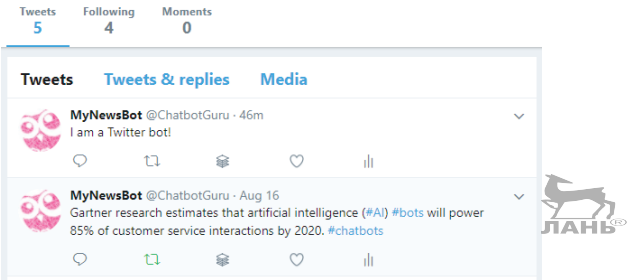


Рис. 7.10

пользователя, разместившего твит. Мы рассмотрим такое поведение чуть позже.

Ретвиты сообщений

Вы можете поделиться твитом со своими подписчиками, используя следующий код *ретвита статуса*:

```
var retweetId = '899681279343570944';
retweet(retweetId);

function retweet(retweetId){
    Twitter.post('statuses/retweet/', {
        id: retweetId
    }, function(err, response) {
        if (err) {
            console.log('Something went wrong while RETWEETING.');
            console.log(err);
        }
        else if (response) {
            console.log('Retweeted!!!');
            console.log(response)
        }
    });
}
```

Поиск твитов

Вы также можете выполнить поиск последних или популярных твитов по хештегам, используя следующий код *поиска хештегов:*

```
search('#brexit', 'popular')
function search(hashtag, resultType){
    var params = {
        q: hashtag, // REQUIRED
        result_type: resultType,
        lang: 'en'
    }

    Twitter.get('search/tweets', params, function(err, data) {
        if (!err) {
            console.log('Found tweets: ' + data.statuses.length);
            console.log('First one: ' + data.statuses[1].text);
        }
        else {
            console.log('Something went wrong while SEARCHING.');
        }
    });
}
```

Обзор новостной службы

Теперь создадим бота, который будет регулярно публиковать новостные статьи для своих подписчиков. Затем мы доработаем его, чтобы организовать персонализированный подход к пользователям, проводя диалоги с прямым обменом сообщениями с ботом. Чтобы создать новостного бота, нам нужен источник новостных публикаций. Мы рассмотрим новостной ресурс под названием NewsAPI.org. NewsAPI – это служба, которая агрегирует новости из примерно 70 изданий по всему миру.

Настройка NewsAPI

Первым делом создадим учетную запись на сайте службы NewsAPI и сгенерируем ключ API.

1. Перейдите на сайт **newsapi.org** – рис. 7.11.
2. Нажмите кнопку **Get API key** (Получить API-ключ).
3. Зарегистрируйтесь, указав свой адрес электронной почты.
4. Сгенерируйте свой ключ API.
5. Изучите источники новостей по адресу **https://newsapi.org/v1/sources?apiKey=**`ВАШ_API_КЛЮЧ`.

   Доступно около 70 источников информации по всему миру, включая популярные, такие как BBC News, Associated Press,

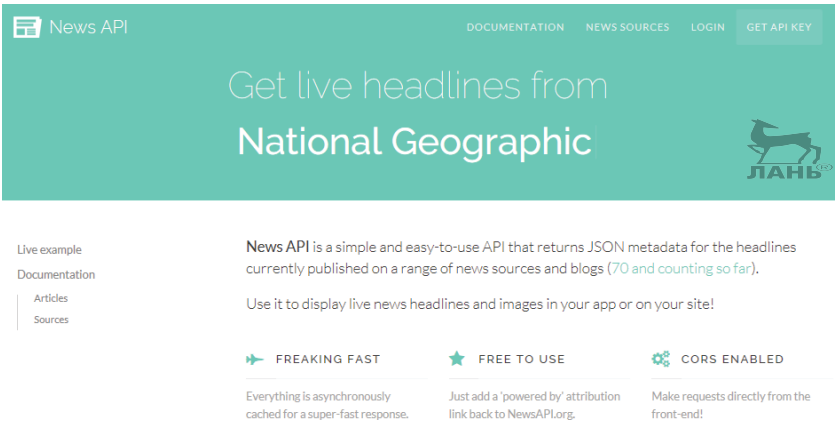


Рис. 7.11

Bloomberg и CNN. Обратите внимание, что каждый источник имеет собственный тег категории. Доступные варианты: `business`, `entertainment`, `gaming`, `general`, `music`, `politics`, `science-and-nature`, `sport` и `technology`. Вы также увидите, что каждый источник имеет метку языка (`en`, `de`, `fr`) и метку страны (`au`, `de`, `gb`, `in`, `it`, `us`). Ниже показана информация об источнике BBC-News:

```
{
    «id»: «bbc-news»,
    “name”: “BBC News”,
    “description”: “Use BBC News for up-to-the-minute news,
    breaking news, video, audio and feature stories.
    BBC News provides trusted World and UK news as well as
    local and regional perspectives. Also entertainment,
    business, science, technology and health news.”,
    “url”: “http://www.bbc.co.uk/news”,
    “category”: “general”,
    “language”: “en”,
    “country”: “gb”,
    “urlsToLogos”: {
        “small”: “”,
        “medium”: “”,
        “large”: “”
    },
    “sortBysAvailable”: [
```

```
        «top»
        ]
}
```

6. Запросите источники определенной категории, языка или страны, указав адрес **https://newsapi.org/v1/sources?category=business&apiKey=**ВАШ_API_КЛЮЧ.

Ниже приведен фрагмент ответа на запрос на все источники в категории бизнеса (business).

```
"sources": [
    {
        "id": "bloomberg",
        "name": "Bloomberg",
        "description": "Bloomberg delivers business
        and markets news, data, analysis, and video
        to the world, featuring stories from Businessweek
        and Bloomberg News.",
        "url": "http://www.bloomberg.com",
        "category": "business",
        "language": "en",
        "country": "us",
        "urlsToLogos": {
            "small": "",
            "medium": "",
            "large": ""
        },
        "sortBysAvailable": [
        "top"
        ]
    },
    {
        "id": "business-insider",
        "name": "Business Insider",
        "description": "Business Insider is a fast-growing
        business site with deep financial, media, tech, and
        other industry verticals. Launched in 2007, the
        site is now the largest business news site on the web.",
        "url": "http://www.businessinsider.com",
        "category": "business",
        "language": "en",
        "country": "us",
        "urlsToLogos": {
            "small": "",
```

```
            "medium": "",
            "large": ""
        },
        "sortBysAvailable": [
        «top»,
        «latest»
        ]
    },
    .
]
```

7. Запросите публикации: https://**newsapi.org/v1/articles?source=bbc-news&apiKey=**`ВАШ_API_КЛЮЧ`.
   Ниже приведен пример ответа:

```
«articles»: [
    {
        "author": "BBC News",
        "title": "US Navy collision: Remains found in
                  hunt for missing sailors",
        "description": "Ten US sailors have been missing since Monday's
                        collision with a tanker near Singapore.",
        "url": "http://www.bbc.co.uk/news/world-us-canada-41013686",
        «urlToImage»:
        «https://ichef1.bbci.co.uk/news/1024/cpsprodpb/80D9/
                      production/_97458923_mediaitem97458918.jpg",
        "publishedAt": "2017-08-22T12:23:56Z"
    },
    {
        "author": "BBC News",
        "title": "Afghanistan hails Trump support in 'joint struggle'",
        "description": "President Ghani thanks Donald Trump for
                        supporting Afghanistan's battle against the
                        Taliban.",
        "url": "http://www.bbc.co.uk/news/world-asia-41012617",
        «urlToImage»:
        «https://ichef.bbci.co.uk/images/ic/1024x576/p05d08pf.jpg»,
        «publishedAt»: «2017-08-22T11:45:49Z»
    },
    .
]
```

Для каждой статьи предоставляются поля `author`, `title`, `description`, `url`, `urlToImage` и `publishedAt`. Теперь, когда мы изучили

источник данных, который предоставляет актуальные новости в разных категориях, продолжим создание новостного бота.

Создание новостного Twitter-бота

Теперь, когда мы изучили NewsAPI, нашли источник актуальных новостей и разузнали немного из того, что умеет API Twitter, объединим полученные знания и разработаем бота, публикующего твиты интересных новостных сюжетов сначала в собственной ленте, а затем персонально для каждого из подписчиков.

1. Разработаем модуль публикации твитов, отображающий рейтинговые новости из источника. Показанный ниже код содержит функцию `tweet()`, которую мы создали ранее:

```
topNewsTweeter('cnn', null);

function topNewsTweeter(newsSource, screen_name, status_id){
    request({
            url: 'https://newsapi.org/v1/articles?source='
            + newsSource +
                        '&apiKey=ВАШ_API_КЛЮЧ',
            method: 'GET'
        },
        function (error, response, body) {
            // ответ бота
            if (!error && response.statusCode == 200) {
                var botResponse = JSON.parse(body);
                console.log(botResponse);
                tweetTopArticle(botResponse.articles, screen_name);
            } else {
                console.log('Sorry. No new');
            }
        });
}

function tweetTopArticle(articles, screen_name, status_id){
    var article = articles[0];
    tweet(article.title + " " + article.url, screen_name);
}
```

Запустите программу, чтобы получить новости из источника CNN и опубликовать самую свежую статью в Twitter:

```
C:\Users\Srini\Dropbox\_Book\workspace\twitterbot>node index.js
Hello World! I am a twitter bot!
{ status: 'ok',
  source: 'cnn',
  sortBy: 'top',
  articles:
   [ { author: 'Nicole Chavez, CNN',
       title: 'Hurricane Harvey strengthens to Category 2',
       description: 'As heavy rain and gusty winds move in over Texas, coastal residents are deciding whethe
r to flee their homes or to stay put and brace for a potentially life-threatening hurricane.',
       url: 'http://www.cnn.com/2017/08/25/us/hurricane-harvey/index.html',
       urlToImage: 'http://i2.cdn.cnn.com/cnnnext/dam/assets/170825094917-01-hurricane-harvey-nws-super-teas
e.jpg',
       publishedAt: '2017-08-25T10:25:21Z' },
     { author: null,
       title: 'What Hurricane Harvey looks like from space - CNN Video',
       description: 'The International Space Station captured video of Hurricane Harvey from about 220 miles
 above the Earth.',
       url: 'http://www.cnn.com/videos/us/2017/08/25/hurricane-harvey-space-station-sje-lon-orig.cnn',
       urlToImage: 'http://i2.cdn.cnn.com/cnnnext/dam/assets/170825100553-hurricane-harvey-space-station-3-s
uper-tease.jpg',
       publishedAt: null },
```

Рис. 7.12

Так выглядит сообщение в Twitter:

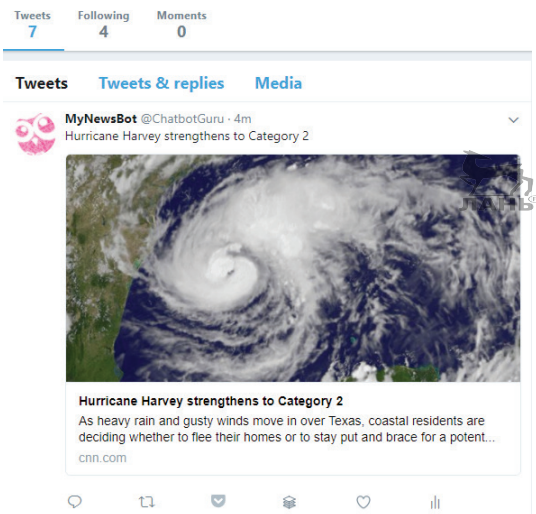


Рис. 7.13

2. Теперь создадим модуль, который публикует новости источника, случайно выбранного из списка:

```
function tweetFromRandomSource(sources, screen_name, status_id){
    var max = sources.length;
    var randomSource = sources[Math.floor(Math.random() * (max + 1))];
```

```
    //topNewsTweeter(randomSource, screen_name, status_id);
}
```

3. Вызовем модуль публикации твитов, после того как получим список источников:

```
function getAllSourcesAndTweet(){
    var sources = [];
    console.log('getting sources.')
    request({
        url: 'https://newsapi.org/v1/sources?
            apiKey=ВАШ_API_КЛЮЧ',
            method: 'GET'
        },
        function (error, response, body) {
            //ответ бота
            if (!error && response.statusCode == 200) {
                // Вывод тела ответа
                var botResponse = JSON.parse(body);
                for (var i = 0; i < botResponse.sources.length;
                    i++){
                    console.log('adding ' +
                                botResponse.sources[i].id)
                    sources.push(botResponse.sources[i].id)
                }
                tweetFromRandomSource(sources, null, null);
            } else {
                console.log('Sorry. No news sources!');
            }
        });
}
```

4. Создайте новый JS-файл с именем *tweeter.js*. В файле *tweeter.js* вызовите функцию `getSourcesAndTweet()`, чтобы запустить процесс публикации новостей:

```
//tweeter.js

var TwitterPackage = require('twitter');
var request = require('request');

console.log("Hello World! I am a twitter bot!");

var secret = {
```

```
    consumer_key: 'ВАШ_КЛЮЧ_КЛИЕНТА',
    consumer_secret: 'ВАШ_СЕКРЕТ_КЛИЕНТА',
    access_token_key: 'КЛЮЧ_ВАШЕГО_ТОКЕНА_ДОСТУПА',
    access_token_secret: 'СЕКРЕТ_ВАШЕГО_ТОКЕНА_ДОСТУПА'
}

var Twitter = new TwitterPackage(secret);
getAllSourcesAndTweet();
```

5. Запустите файл *tweeter.js* в консоли. Этот бот будет публиковать новостные твиты при каждом вызове. Он будет случайным образом выбирать рейтинговые новостные сюжеты из примерно 70 источников новостей.

Персонализация новостного бота

Теперь у нас есть новостной бот, который публикует новости (или обновления статуса) в своей ленте. Мы рассмотрим, как настроить его для постоянного запуска время от времени. Разработаем более интересного бота, который сможет своим подписчикам отправлять твиты с новостями, которые могут им понравиться. Другими словами, давайте персонализировать новости!

1. Предположим, что у нас есть информация о пользователях и их интересах. Добавьте в файл *tweeter.js* показанный ниже код. На данный момент мы жестко кодируем информацию о пользователях в самом коде. Позже рассмотрим, как можно ее хранить и извлекать из базы данных.

```
var userInterests = [{'screen_name':'srinivasancj',
                          'user_interest': 'technology'}];
```

2. Создайте модуль `tweetUserSpecificNews` с модулем `userInterests` для получения источников, относящихся к определенной категории. Вызовите функцию `tweetUserSpecificNews()` после функции `getAllSourcesAndTweet()`.

```
tweetUserSpecificNews();

function tweetUserSpecificNews(){
console.log('Tweeting personalised news');
    for (var i=0; i < userInterests.length; i++){
        var user = userInterests[i];
```

```
        var screen_name = user.screen_name;
        var interest = user.user_interest;
        var status_id = user.in_reply_to_status_id;
        // Запрос источников
        request({
            url: 'https://newsapi.org/v1/sources?category=' +
                  interest +
                          '&apiKey=ВАШ_API_КЛЮЧ',
            method: 'GET'
        },
        function (error, response, body) {
            if (!error && response.statusCode == 200) {
                // Вывод тела ответа
                var botResponse = JSON.parse(body);
                console.log(botResponse);
                var sources = [];
                for (var i = 0; i < botResponse.sources.length;
                    i++)
                {
                    console.log('adding ' +
                                botResponse.sources[i].id)
                    sources.push(botResponse.sources[i].id)
                }
                tweetFromRandomSource(sources, screen_name, status_id);
            } else {
                console.log('Sorry. No news in this category.');
            }
        });
    }
}
```

С указанием значения screen_name твита от пользователя твиты, отправленные ботом, рассматриваются как ответ на оригинальный твит пользователя. Поэтому эти твиты не ограничиваются публикацией в ленте бота. Вместо этого они отправляются непосредственно пользователю и поэтому персонализируются для него.

3. Теперь, когда у нас есть бот, который при запуске добавляет в свою ленту случайные новостные заметки и отправляет персонализированные новости своим подписчикам, давайте настроим его автоматический постоянный запуск, скажем, один раз в час. Для этого нам сначала нужно создать веб-приложение и разместить его в облаке.

4. Создайте папку *bin* и переместите в нее файл *tweeter.js*.
5. Переименуйте файл *tweeter.js* в *tweeter* и добавьте показанный ниже код в первой строке. Код в ней инструктирует платформу Heroku, какой интерпретатор использовать для запуска сценария:

```
#!/usr/bin/env node
```

6. В корневом каталоге проекта создайте файл с именем *Procfile* со следующим кодом:

```
web: node index.js
```

7. Создайте локальный репозиторий Git для проекта, добавьте файлы и выполните начальную инициализацию. Введите следующие команды в консоли:

```
> git init
> git add .
> git commit -m initial-commit-personalised-bot
```

8. Создайте приложение Heroku:

```
> heroku create my-twitter-bot
```

9. Выгрузите приложение в облако Heroku:

```
> git push heroku master
```

10. Добавьте планировщик Heroku:

```
> heroku addons:add scheduler
```

11. Откройте панель управления Heroku в браузере, перейдя по адресу **dashboard.heroku.com/apps**.
12. Выберите свою программу Twitter-бота и откройте панель управления приложением.
13. На вкладке **Overview** (Обзор) в разделе **Installed add-ons** (Установленные надстройки) вы увидите пункт **Heroku Scheduler** (Планировщик Heroku). Щелкните мышью по нему – рис. 7.14.
14. На открывшейся странице отображаются запланированные задачи для приложения. Поскольку мы только что создали это

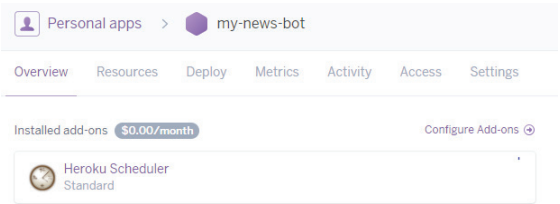


Рис. 7.14

приложение, список будет пуст. Щелкните мышью по ссылке **Add new job** (Добавить новое задание).

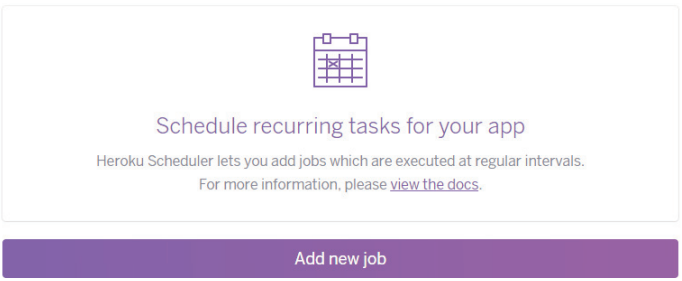


Рис. 7.15

15. Введите название программы, которая должна запускаться по расписанию, в нашем случае это **tweeter**, и нажмите кнопку **Save** (Сохранить).

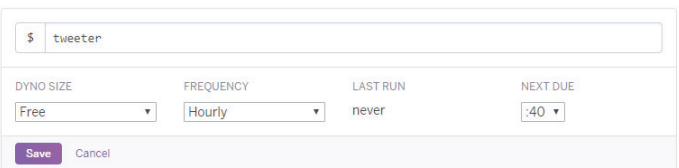


Рис. 7.16

16. Планировщик теперь будет запускать Twitter-бота постоянно, отправляя персонализированные новости подписчикам, – рис. 7.17.

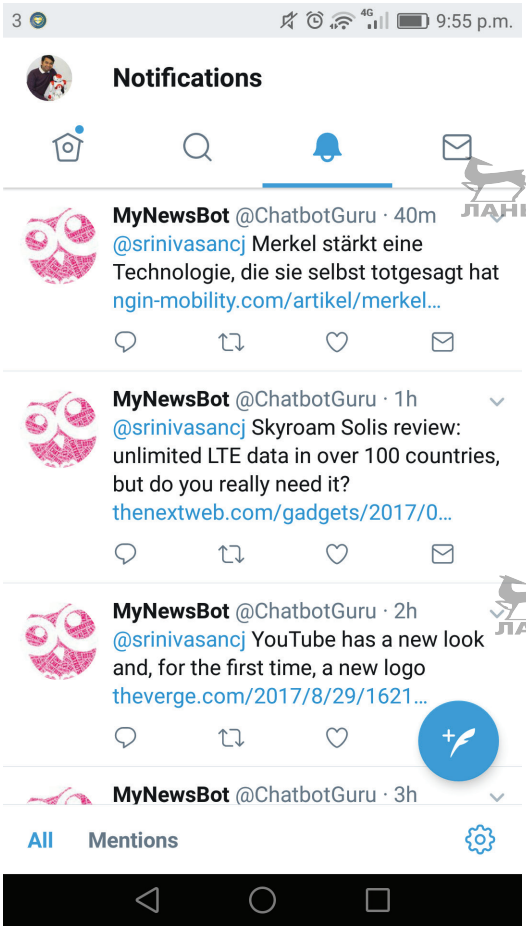


Рис. 7.17

Создание базы данных интересов пользователей

Теперь, когда у нас есть регулярно запускающееся веб-приложение, отправляющее персонализированные новости пользователям, перейдем к следующему шагу – созданию базы данных, из которой Twitter-бот может получить информацию о пользователях. На данный момент мы жестко закодировали эту информацию в коде

бота, что не очень удобно. Чтобы создать базу данных, применим хостинговую службу MongoDB на сайте **mlab.com**.

> **Примечание.** Имейте в виду, что, согласно законодательству, если вы решите сделать бота общедоступным, вам, возможно, придется сообщить пользователям, что бот будет хранить персональные данные пользователей.

Чтобы создать базу данных для Twitter-бота, выполните следующие действия.

1. Перейдите на сайт **www.mlab.com** и щелкните мышью по ссылке на **Sign Up** (Зарегистрироваться).

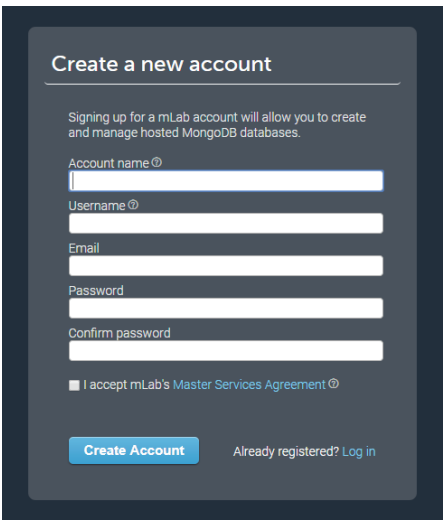


Рис. 7.18

2. Заполните данные аккаунта и щелкните мышью по ссылке **Create Account** (Создать учетную запись).
3. Проверьте свою электронную почту и перейдите по ссылке в отправленном письме. Откроется панель управления учетной записью – рис. 7.19.
4. Нам нужна служба MongoDB. На панели управления щелкните мышью по кнопке **Create new** (Создать новую) – рис. 7.20.
5. Выберите поставщика облачных услуг, который вам нравится, из следующих трех вариантов: веб-служба Amazon, облачная платформа Google или Microsoft Azure.

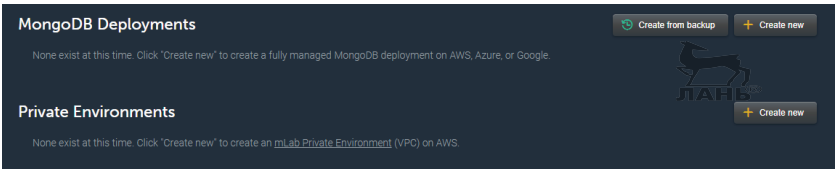


Рис. 7.19

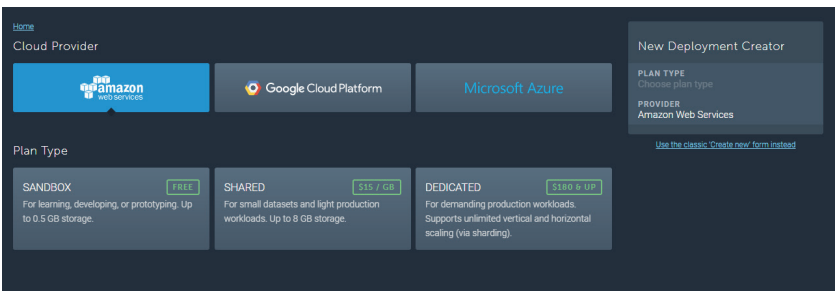


Рис. 7.20

6. Щелкните мышью по ссылке **Sandbox** (Песочница). Песочница – это тип среды выполнения, в которой вы можете безопасно экспериментировать с программами, прежде чем переносить их в рабочую среду. Щелкните мышью по кнопке **Continue** (Продолжить).

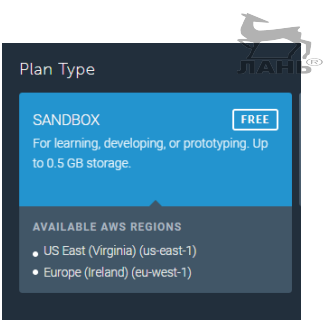


Рис. 7.21

7. Выберите регион AWS и щелкните мышью по кнопке **Continue** (Продолжить) – рис. 7.22.
8. Укажите завершающие данные, имя базы данных и щелкните мышью по кнопке **Continue** (Продолжить) – рис. 7.23.

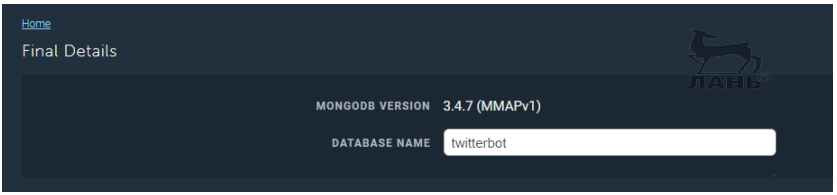


Рис. 7.22

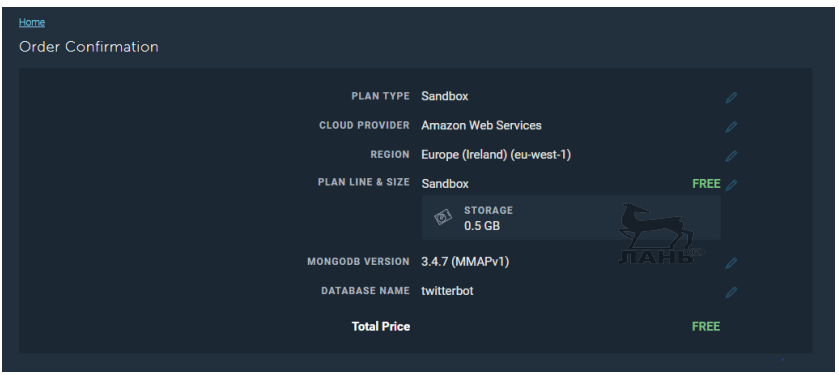


Рис. 7.23

9. Проверьте данные и щелкните мышью по кнопке **Submit Order** (Оформить заказ), чтобы завершить настройку, – рис. 7.24.

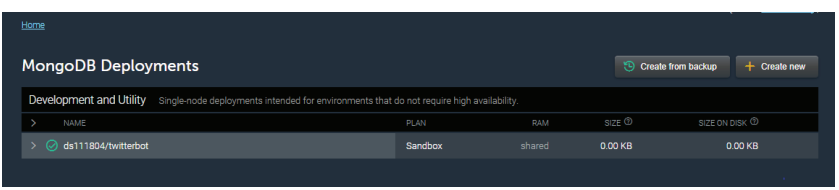


Рис. 7.24

Итак, вы создали базу данных с именем `twitterbot`, которая может использоваться для хранения информации, касающейся интересов пользователя.

10. Создайте новую коллекцию, дважды щелкнув на записи базы данных `twitterbot`, – рис. 7.25.
11. Щелкните мышью по кнопке **Add collection** (Добавить коллекцию). Создайте коллекцию с именем `users`.

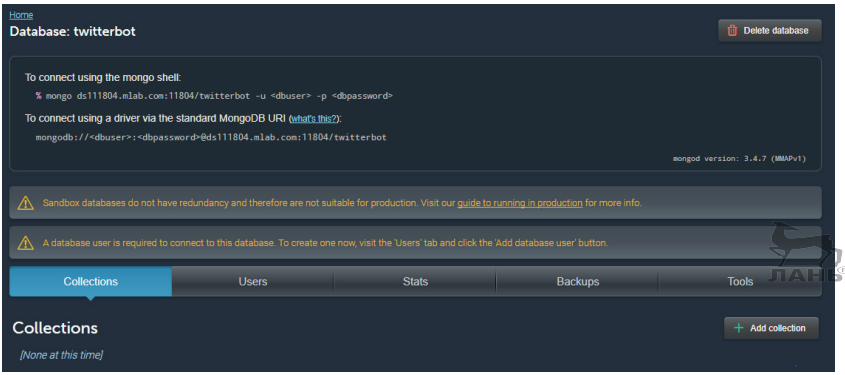


Рис. 7.25

12. Теперь, чтобы подключиться к базе данных, нам необходимо определить ее пользователей. Чтобы создать пользователя, откройте вкладку **Users** (Пользователи) (рядом с вкладкой **Collections** (Коллекции)).
13. Щелкните мышью по ссылке **Add database user** (Добавить пользователя в базу данных).
14. Создайте нового пользователя в базу данных.
15. Сформируйте URL-адрес базы данных. Синтаксис выглядит так:

```
mongodb://<пользователь>:<пароль>@ds111804.mlab.com:11804/twitter bot
```

    Замените значения <пользователь> и <пароль> идентификатором и паролем пользователя, которого вы только что создали. Подождите.
16. Добавим некоторые данные в коллекцию `users` для считывания Twitter-ботом. Перейдите на вкладку **Collections** (Коллекции), а затем в коллекцию `users`.
17. Щелкните мышью по ссылке **Add document** (Добавить документ). Введите объект JSON из списка `user_interests`. Щелкните мышью по кнопке **Create and go back** (Создать и вернуться назад) – рис. 7.26.
18. Теперь ваши документы добавлены в коллекцию – рис. 7.27.

Прекрасно! Теперь у нас есть база данных документов MongoDB в облаке (Mlab.com) для хранения информации об интересах пользователя. А сейчас выясним, как использовать эту базу данных для хранения и получения информации об интересах пользователя с помощью бота.

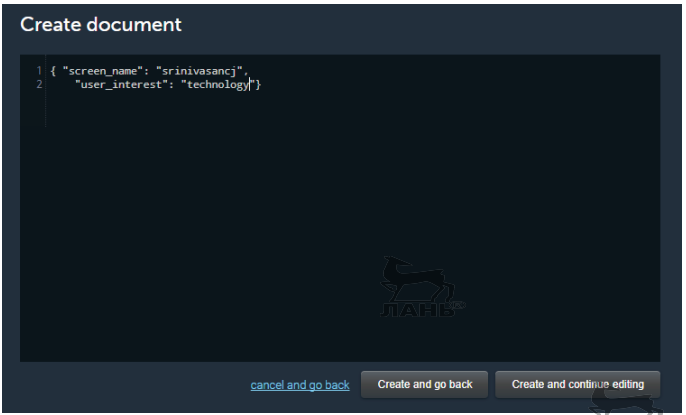


Рис. 7.26

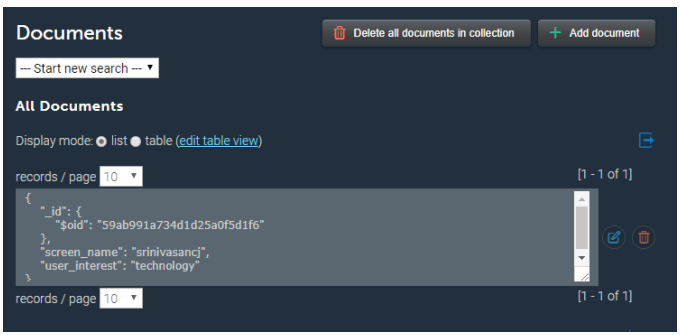


Рис. 7.27

Доступ к базе интересов пользователей

В предыдущем разделе мы создали базу данных документов MongoDB, в которой можем хранить информацию о пользователях. Эти сведения будут использоваться Twitter-ботом для персонализации новостей. В данном разделе мы увидим, как бот может подключиться к базе данных в режиме реального времени.

1. Установите библиотеку `mongodb` (версия 2.2.31) с помощью команды `npm install`:

```
> npm install mongodb@2.2.31 -save
```

2. Протестируйте соединение, создав программу Node.js с именем *mongodb.js* и следующим кодом:

```
var MongoClient = require('mongodb').MongoClient;

// URL подключения
var url = 'URI_ВАШЕЙ_БД_MONGO';

// Использовать метод connect для подключения к серверу
MongoClient.connect(url, function(err, db) {
  console.log("Connected successfully to server");
  db.close();
});
```

3. Запустите код, чтобы узнать, может ли программа подключаться к вашей базе данных.
4. Добавьте функцию в файл *mongodb.js*, позволяющую найти все документы в вашей коллекции:

```
function findDocuments(db, callback) {
    // Получение коллекции документов
    var collection = db.collection('users');
    // Поиск документов
    collection.find({}).toArray(function(err, docs) {
        callback(docs);
    });
}
```

5. Замените вызов `db.close()` в `mongodb.js` вызовом предыдущей функции, как показано ниже:

```
var userInterests = [];

// Использовать метод connect для подключения к серверу
MongoClient.connect(url, function(err, db) {
  console.log("Connected successfully to server");
  findDocuments(db, function(docs) {
      for (var i = 0; i < docs.length; i++){
          var user = {};
          user.screen_name = docs[i].screen_name;
          user.user_interest = docs[i].user_interest;
          userInterests.push(user);
      }
      db.close();
```

```
      console.log(userInterests);
  });
});
```

6. Выполните код, чтобы увидеть, извлекает ли программа документ, который мы поместили в коллекцию.

```
C:\Users\Srini\Dropbox\_Book\workspace\twitterbot>node mongodb.js
Connected successfully to server
[ { screen_name: 'srinivasancj', user_interest: 'technology' } ]
```

Рис. 7.28

Вы должны увидеть документ, который мы сохранили в коллекции users. Позже мы рассмотрим, как добавить больше пользовательских данных в коллекцию с помощью Twitter-бота.

7. Теперь, когда мы можем успешно прочитать сведения об интересах пользователей, подключим базу данных к боту, чтобы он мог отправлять персонализированные новости пользователям на основе информации, найденной в базе данных.
8. Откройте файл программы *tweeter* в папке *bin*. Замените объявление переменной userInterests следующим кодом:

```
var MongoClient = require('mongodb').MongoClient;
// URL подключения
var url = 'YOUR_MONGO_DB_URI';
var userInterests = [];
loadUserData();
```

9. Определите функцию loadUserData(), которая будет загружать пользовательские данные из базы данных документов:

```
function loadUserData(){
    // Использовать метод connect для подключения к серверу
    MongoClient.connect(url, function(err, db) {
        console.log("Connected successfully to server");

        findDocuments(db, function(docs) {
            //console.log("Found the following records");
            for (var i = 0; i < docs.length; i++){
                var user = {};
                user.screen_name = docs[i].screen_name;
                user.user_interest = docs[i].user_interest;
                userInterests.push(user);
```

```
            }
            db.close();
            console.log(userInterests);
            // Твит для тех последователей, которые
            // проявили интерес к конкретным категориям
            tweetUserSpecificNews();
        });
    });
}
```

10. Обратите внимание, как теперь вызывается функция `tweetUserSpecificNews()` после загрузки в массив `userInterests` информации из базы данных. Удалите вызов этой функции, если он используется вне функции `loadUserData()`.
11. Запустите программу в консоли (в папке *bin*), чтобы протестировать ее.

```
> node tweeter
```

Программа позволит извлечь пользовательские интересы из базы данных, запросить все источники, выбрать источники новостей, подходящих под конкретного пользователя, случайно выбрать одну и передать рейтинговую новость из выбранного источника.

```
C:\Users\Srini\Dropbox\_Book\workspace\twitterbot\bin>node tweeter
Hello World! I am the hourly twitter bot!
Connected successfully to server
[ { screen_name: 'srinivasancj', user_interest: 'technology' } ]
Tweeting personalised news
Sending tweet to: srinivasancj
Tweet:@srinivasancj Why technology could be the answer to solving dyslexia
 https://thenextweb.com/contributors/2017/09/02/technology-answer-solving-
dyslexia/
Tweeted!!!
```

Рис. 7.29

Прекрасно! Мы теперь на один шаг ближе к разговаривающему Twitter-боту. Единственный фрагмент головоломки, которого пока нет, – это способ сообщить боту о том, что интересует пользователя, без необходимости вручную создавать документы в базе данных.

Информирование бота об интересах пользователей

Было бы круто, если бы пользователи бота сообщали ему о том, что им интересно, а что нет? Затем бот мог бы персонализировать

новости на основе полученных сведений. Диалог мог бы вестись примерно так:

```
Пользователь: Я интересуюсь политикой.
Бот каждый час публикует новости из мира политики.
Пользователь: Присылай научные новости.
Бот каждый час публикует новости из мира политики и науки.
Пользователь: Больше не присылай новости из мира политики.
Бот каждый час публикует новости из мира науки.
```

Чтобы сделать этот диалог с Twitter-ботом возможным, нам нужен слушатель, который отслеживает твиты, отправленные боту пользователями, и информирует его об их интересах. Эта информация на естественном языке нуждается в анализе, распознавании и обновлении сведений в базе данных. Бот каждый час будет сканировать персонализированные новости на основе информации, которую получает из базы данных. Звучит неплохо, да? Рассмотрим слушатель твитов.

1. Откройте файл *index.js*. Сохраните код импорта библиотеки Twitter и укажите свои данные учетной записи:

```
var TwitterPackage = require('twitter');
var request = require('request');

var secret = {
    consumer_key: 'ВАШ_КЛЮЧ_КЛИЕНТА',
    consumer_secret: 'ВАШ_СЕКРЕТ_КЛИЕНТА',
    access_token_key: 'КЛЮЧ_ВАШЕГО_ТОКЕНА_ДОСТУПА',
    access_token_secret: 'СЕКРЕТ_ВАШЕГО_ТОКЕНА_ДОСТУПА'
}

var Twitter = new TwitterPackage(secret);
console.log("Hello World! I am a twitter bot!");
```

2. Настройте слушатель твитов так, чтобы он распознавал все сообщения, отправленные боту. Канал обработки моего Twitter-бота – @chatbotguru.

```
// Слушаем входящие твиты
Twitter.stream('user', {}, function(stream) {
    stream.on('data', function(tweet) {
        console.log(tweet);
```

```
        var ct = /@chatbotguru/i;
        var userUtt = 'null';
        if (tweet.text.search(ct) != -1){
            userUtt = tweet.text.replace(ct, '');
            console.log('Tweet Msg:' + userUtt);
            console.log('Tweet from:' + '@' +
            tweet.user.screen_name);
        }
    });
});
```

3. Запустите код:

```
> node index.js
```

4. Отправьте твит из своего личного аккаунта Twitter в учетную запись бота. Например, `@chatbotguru I am interested in business news`. Этот твит будет получен слушателем, которого мы только что создали:

```
C:\Users\Srini\Dropbox\_Book\workspace\twitterbot>node index.js
Hello World! I am a twitter bot!
Tweet Msg: I am interested in business news.
Tweet from:@srinivasancj
```

Рис. 7.30

Вы также увидите твит, полученный ботом в вашем приложении Twitter:

Рис. 7.31

5. Теперь поработаем с сообщением, которое мы получили от пользователя. Рассмотрим высказывание по интересующей теме (политика, бизнес и т. д.) и переданные эмоции (заинтересован/не заинтересован). Затем полученные сведения используются для

добавления или удаления записи интереса пользователя в базе данных.

```
if (tweet.text.search(ct) != -1){
    userUtt = tweet.text.replace(ct, '');
    console.log('Tweet Msg:' + userUtt);
    console.log('Tweet from:' + '@' + tweet.user.screen_name);

    var userInterest = getInterestedGenre(userUtt);
    var userSentiment = getSentiment(userUtt);

    var user = { 'screen_name' : tweet.user.screen_name,
                 'user_interest' : userInterest};

    console.log(user);

    // Использовать метод connect для подключения к серверу
    MongoClient.connect(url, function(err, db) {
        console.log("Connected successfully to server");
        var collection = db.collection('users');
        if (userSentiment == 'positive'){
            collection.insertMany([user], function(err, result) {
                if (err){
                    console.log(err);
                } else {
                    console.log("Inserted a user interest into the collection");
                    db.close();
                }
            });
        } else {
            collection.deleteOne(user, function(err, result) {
                console.log(err);
                console.log("Deleted a user interest from the collection");
                db.close();
            });
        }
    });
}
```

6. Определите функции `getInterestedTopic()` и `getSentiment()`:

```
function getSentiment(text){
    if (text.search('not interested') != -1){
        return 'negative';
```

```
    }
    if (text.search('no more') != -1){
        return 'negative';
    }
    if (text.search('don\'t send') != -1){
        return 'negative';
    }
    if (text.search('no ') != -1){
        return 'negative';
    }
    if (text.search('dont like ') != -1){
        return 'negative';
    }
    if (text.search('unsubscribe ') != -1){
        return 'negative';
    }
    if (text.search('don\'t follow ') != -1){
        return 'negative';
    }
    if (text.search('stop ') != -1){
        return 'negative';
    }
    return 'positive';
}

function getInterestedGenre(text){
    if (text.search('tech') != -1 ||
        text.search('technology') != -1 ){
        return 'technology';
    }
    else if (text.search('all kinds') != -1){
        return 'general';
    }
    else if (text.search('politics') != -1 ||
        text.search('political') != -1){
        return 'politics';
    }
    else if (text.search('sports') != -1){
        return 'sport';
    }
    else if (text.search('business') != -1){
        return 'business';
    }
}
```

7. Это очень простые определения, чтобы было проще понять. Вы можете использовать инструменты распознавания NLU, такие как Dialogflow, для использования более сложных функций. Чтобы предыдущий код работал, нам также нужно импортировать библиотеку MongoDB и указать это в файле *index.js*:

```
var MongoClient = require('mongodb').MongoClient;
var url = 'YOUR_MONGO_DB_URI';
```

8. Запустите код еще раз, отправьте твит и проверьте, все ли работает. Вы увидите, что бот теперь умеет вставлять/удалять записи из базы данных. Перейдите к сайту **mlab.com** и убедитесь в этом:

Рис. 7.32

Получилось! Ваш Twitter-бот теперь хранит обновления в базе данных и передает персонализированные новости всем своим пользователям. И наконец, внесите изменения в настройки облака Heroku, чтобы вам не пришлось хранить файл *index.js* на локальном компьютере.

Заключение

Ура! Вы создали своего собственного разговаривающего Twitter-бота. Я настоятельно рекомендую вам развивать тему и изучить другие виды диалога, которые можно реализовать с Twitter-ботом. Как насчет того, чтобы бот занялся поиском и публиковал ретвиты по хештегам автоматически? Как бы вы могли использовать статистику, такую как количество ретвитов, число последователей и т. д.,

чтобы определять приоритет твитов для ретвита? Изучите эти вопросы, чтобы создать лучшего и более полезного Twitter-бота.

В этой главе мы многое рассмотрели. Мы начали с API Twitter и получили представление о том, как мы можем автоматически создавать твиты, публиковать ретвиты и искать сообщения, используя хештеги. Затем мы исследовали API-интерфейс источника новостей, в котором представлены статьи из примерно 70 различных СМИ. Мы объединили его с нашим Twitter-ботом, чтобы создать новую версию бота. Мы изучили, как персонализировать новости для пользователей, используя теги пользовательских интересов, хранящиеся в базе данных MongoDB.

Наконец, мы выяснили, как закончить разработку, создав механизм, позволяющий боту получать твиты с учетом интересов пользователей и хранить их в базе данных. Надеюсь, я дал вам представление о том, насколько полезны Twitter-боты, и замотивировал вас на дальнейшие разработки. В следующей главе мы рассмотрим, как создать голосового бота на платформе Amazon Echo.

Ссылки

- Документация по Twitter API: **developer.twitter.com/en/docs**.
- Документация по MongoDB: **docs.mlab.com**.

Глава 8. Мой телегид

В этой главе мы рассмотрим новых помощников, захватывающих мир, – домашних ассистентов с голосовым управлением. Это чат-боты с ИИ, которые внедряют функции вашего помощника по дому. Такие устройства, к примеру, могут стоять на столе в вашей гостиной и взаимодействовать с вами с помощью голоса. Они называются умными динамиками (смарт-колонками), но это не просто динамики. Эта самые настоящие ассистенты, как и чат-боты, которых мы изучили в предыдущих главах, но в ином исполнении. Одна из самых популярных колонок называется Echo, и разработана она в компании Amazon. Несмотря на то что устройство называется Echo, программное обеспечение называется Alexa.

На рынке есть другие умные динамики (например, Google Home). Тем не менее Amazon Echo пока лидирует в гонке с долей рынка около 70% в 2017 году. Считается, что со временем распространение умных колонок будет увеличиваться экспоненциально[1]. В этой главе мы рассмотрим, как расширить возможности Alexa, внедрив в нее собственные навыки. Сначала мы создадим простой навык – выдача вдохновляющей цитаты, способной сделать ваш день. Затем мы перейдем к созданию навыка, с помощью которого Alexa поможет вам с телепрограммой.

В ходе этой главы вы:

- изучите основы Amazon Alexa;
- разберете и научитесь создавать слоты, цели и модели взаимодействия;
- изучите встроенные слоты и цели;
- создадите сервер управления диалогами на платформе Heroku и интегрируете в него технологии Alexa;
- интегрируете в менеджер диалогов источники данных Quote.rest и API TVMaze;
- развернете навыки на платформе Amazon Echo.

[1] www.emarketer.com/Article/Alexa-Say-What-Voice-Enabled-Speaker-Usage-Grow-Nearly-130-This-Year/1015812.

Amazon Alexa

Amazon Alexa поддерживает собственную платформу разработки, которая позволяет программистам обучать ее новым навыкам. Навыки – это не что иное, как диалоговые возможности, которые можно настроить в Alexa, используя инструменты Amazon и сторонних разработчиков. Для начала работы вам не понадобится умный динамик Amazon Echo, но если вы купили его, будет замечательно. Amazon Echo выпускается в нескольких вариантах: Amazon Echo, Amazon Echo Dot, Amazon Echo Plus, Amazon Echo Spot и Amazon Echo Show. Echo Dot – самая простая версия стоимостью 49 долларов США на момент написания книги. Ее будет достаточно, чтобы разработать и протестировать разработанные навыки.

Рис. 8.1 *Источник: Amazon.com*

В экосистеме Alexa также есть магазин навыков, в котором можно публиковать пользовательские навыки. Опубликованные навыки можно просмотреть и установить на устройство Echo, тем самым добавив их к стандартному набору навыков Alexa. Это можно сделать либо на партнерском сайте Alexa (**alexa.amazon.com**), либо в приложении Amazon Alexa (доступно в магазине Google Play/AppStore), либо на сайте Amazon (**www.amazon.com/alexa-skills/**). Alexa подключается к вашей учетной записи Amazon и может побеседовать с вами о многих вещах: настройке таймеров, погоде, курсах валют, составлении списка дел или о покупках, воспроизведении музыки, чтении книг, покупках на сайте Amazon и т. д. К сожалению, устройство не поддерживает русский язык, но вы можете попрактиковаться с ним на английском языке:

- Alexa, what time is it?
- Alexa, how is the weather?
- Alexa, what is the news update?

- Alexa, tell me a joke.
- Alexa, add eggs to my shopping list.

Разработка бота цитат

1. Чтобы начать работу, вам понадобится учетная запись разработчика Amazon. Зарегистрируйтесь на сайте с URL-адресом **developer.amazon.com/home.html**.
2. После регистрации авторизуйтесь в системе.

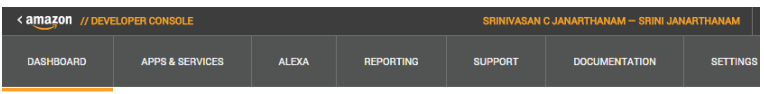


Рис. 8.2

3. На панели управления перейдите на вкладку **Alexa**.
4. Выберите пункт **Alexa Skills Kit** (Набор навыков Alexa) и щелкните мышью по кнопке **Get Started** (Начать).

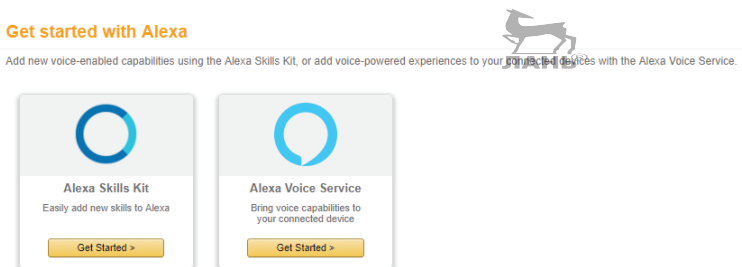


Рис. 8.3

5. Чтобы создать новый навык, щелкните мышью по кнопке **Add a New Skill** (Добавить новый навык).

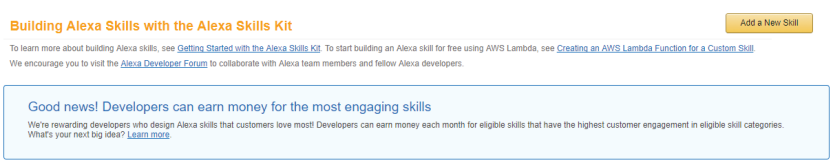


Рис. 8.4

Создание навыков Alexa – двухэтапный процесс:

- настройка навыков на портале разработчика;
- создание модуля управления диалогом.

Настройка навыков

Чтобы освоить инструментарий, мы возьмем за пример разработку навыка, который каждый день выдает вдохновляющую цитату. Таким образом, диалог между пользователем и Alexa может выглядеть следующим образом:

```
Пользователь: Alexa! Ask quote master to inspire me.
Alexa: The whole is greater than the sum of its parts.
```

На портале разработчиков нужно пройти показанный ниже процесс из семи этапов, позволяющий настроить навык и его модель взаимодействия:

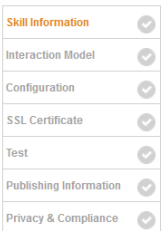


Рис. 8.5

1. На странице с формой настройки навыка установите переключатель **Skill Type** (Тип навыка) в положение **Custom Interaction Model** (Пользовательская модель взаимодействия) – рис. 8.6.
2. Выберите в раскрывающемся списке язык навыка. В настоящее время Alexa поддерживает английский (США, Великобритания, Индия) и немецкий языки.
3. Присвойте навыку название, скажем, **Quote Master**. Это название будет отображаться в приложении Alexa для других пользователей, если они захотят установить его на своих собственных устройствах Echo.
4. Присвойте навыку имя вызова, скажем, `Quote Master`. По этому имени навык вызывается в коде Alexa. Рекомендации по выбору

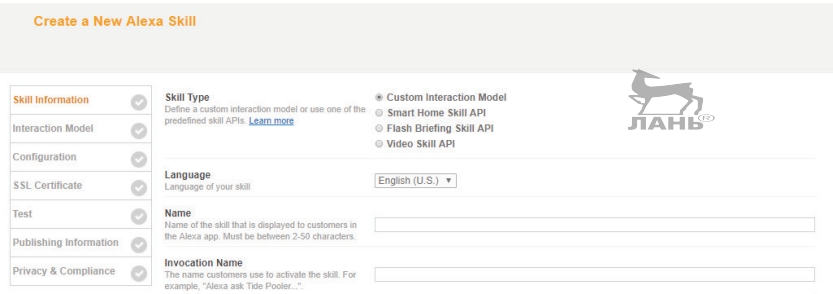


Рис. 8.6

имен вызовов доступны по адресу **developer.amazon.com/docs/custom-skills/choose-the-invocation-name-for-a-custom-skill.html#invocation-name-requirements**. Вы можете использовать несколько навыков в своей версии Alexa, однако два навыка не могут иметь одинаковое имя вызова. В противном случае конфликта, при создании навыка и указании существующего имени, будет выведено уведомление.

5. Оставьте остальные поля со значениями по умолчанию, нажмите кнопку **Save** (Сохранить) и затем нажмите кнопку **Next** (Далее).

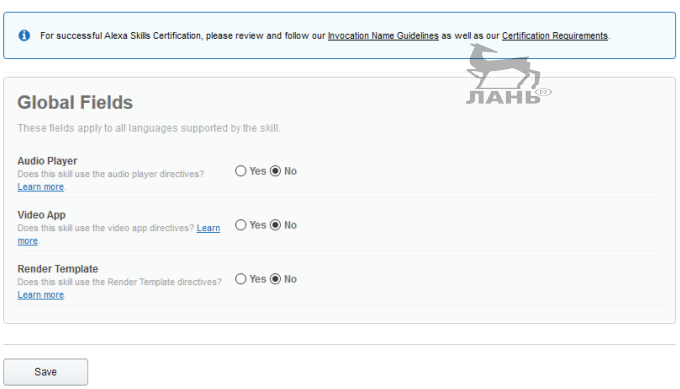


Рис. 8.7

Модель взаимодействия

Прежде чем выстраивать модель взаимодействия, разберемся с несколькими основными концепциями. Модель взаимодействия

определяет тип высказываний, которые Alexa должна распознавать для использования навыка. Рассмотрим в виде целей и примеров. Цели – это группа запросов и ответов, которые пользователь осуществляет при взаимодействии с навыком. Каждая цель может быть выражена разными способами. Например, пользователь может ввести приветствие и сказать следующее:

- `hi`
- `hi there`
- `hello there`
- `what's up`
- `good day`

В вышеупомянутых примерах цель может быть собирательно названа как `greet`.

1. Нажмите кнопку **Launch Skill Builder** (Запустить Skill Builder).

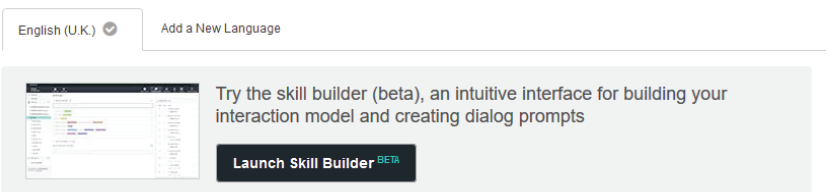


Рис. 8.8

2. Откроется страница инструментария с панелью управления для создания навыков – рис. 8.9.
   На вкладке **Intents** (Цели) показано, что у навыка уже есть три цели, добавленные по умолчанию: `CancelIntent`, `HelpIntent` и `StopIntent`. Это цели, встроенные по умолчанию, которые добавляются к каждому навыку. Это означает, что ваш навык уже распознает запросы на отмену, помощь и остановку.
   Альтернативой панели управления является просмотр той же информации в формате кода, если щелкнуть мышью по пункту **Code Editor** (Редактор кода), – рис. 8.10.
3. Чтобы создать новую цель, щелкните мышью по ссылке **Add+** (Добавить+) на вкладке **Intents** (Цели) – рис. 8.11.
4. Существует два типа целей, которые могут быть созданы или добавлены к навыку. Есть встроенные цели, которые можно добавить. Это заранее определенные цели, которые можно повторно использовать. Такая возможность экономит разработчикам

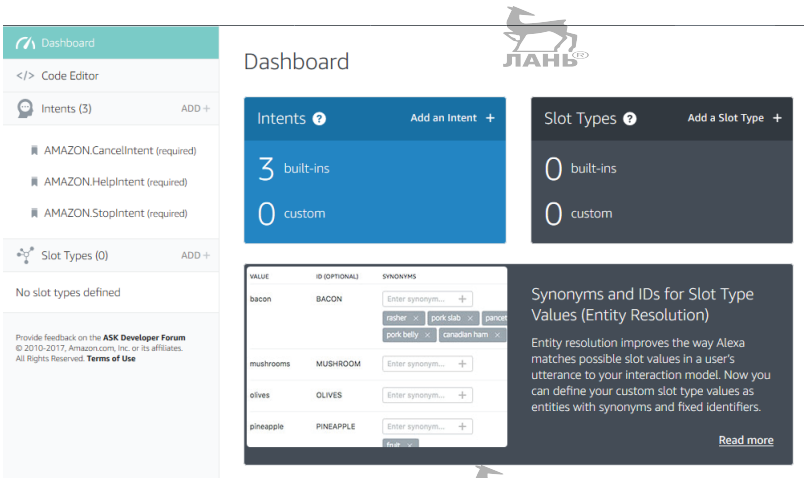


Рис. 8.9

Рис. 8.10

много времени, они назначают подобные цели для каждого навыка, который создают. Существуют также пользовательские цели с индивидуальными настройками, адаптированные под потребности конкретного навыка, – рис. 8.12.

5. Щелкните мышью по ссылке **Use an existing intent from the built-in library** (Использовать существующую цель из встроенной библиотеки), чтобы просмотреть список предопределенных целей, – рис. 8.13.
6. Щелкните мышью по ссылке **Create a new custom intent** (Создать новую пользовательскую цель). Введите название цели, скажем `GetQuote`, и щелкните мышью по ссылке **Create intent** (Создать цель).

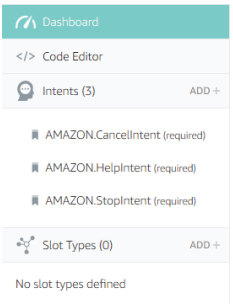


Рис. 8.11

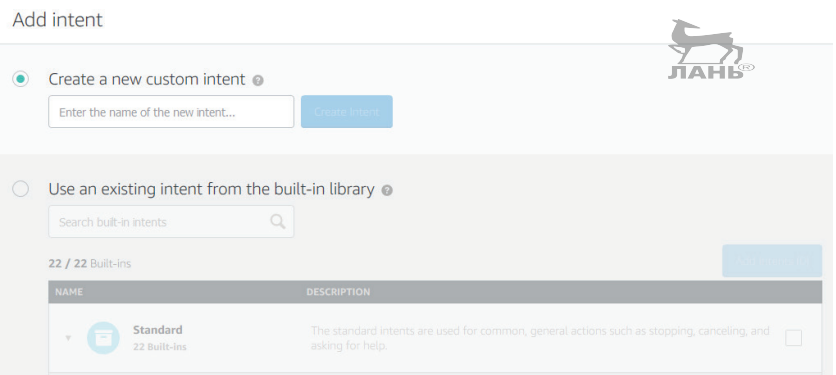


Рис. 8.12

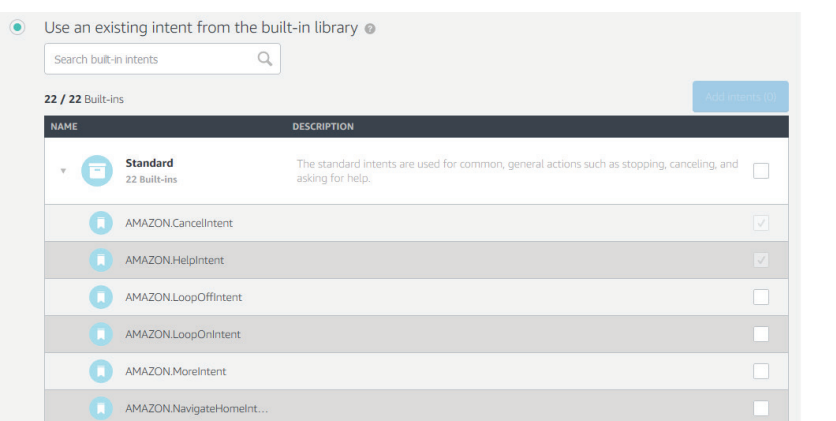


Рис. 8.13

7. Добавьте несколько примеров высказываний, как мы обсуждали ранее. Вводите высказывания по очереди и щелкайте мышью по ссылке **+** или нажмите клавишу **Enter**, чтобы добавить их в список высказываний для выбранной цели.

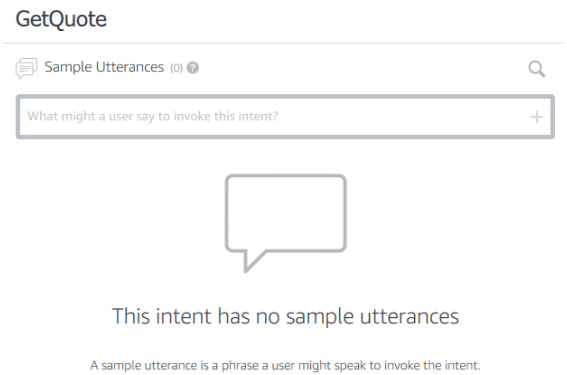


Рис. 8.14

8. Добавим еще одну цель ради интереса: `GetAuthor`. Эта цель служит для извлечения имени автора цитаты.

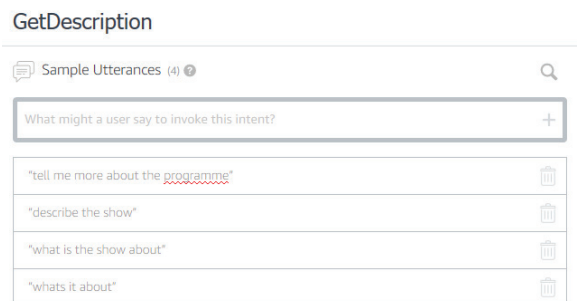


Рис. 8.15

9. Нажмите кнопку **Save Model** (Сохранить модель), а затем **Build Model** (Построить модель) и подождите пару минут, пока выстраивается модель взаимодействия, – рис. 8.16.
10. Теперь, когда мы указали запросы в виде примерных пользовательских высказываний и целей, нажмите кнопку **Skill**

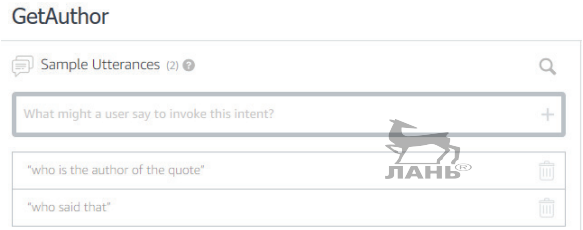


Рис. 8.16

Information (Информация о навыке), чтобы вернуться к странице **Skills Configuration** (Конфигурация навыков). В боковом меню выберите пункт **Configuration** (Конфигурация).

Рис. 8.17

11. Вы перешли к следующему этапу разработки, когда нужно создать модуль для управления диалогом. На этом этапе мы настраиваем конечную точку, в которой нам потребуется менеджер диалогов. У нас есть два варианта: AWS и HTTPS. В случае с AWS вам необходимо создать службу AWS Lambda для размещения конечной точки сервера. В случае HTTPS мы можем разместить веб-сервер на любой облачной платформе по желанию. Мы воспользуемся вариантом HTTPS.
12. В текстовое поле **Default** (По умолчанию) введите URL-адрес веб-приложения менеджера диалогов для вашего навыка. Поскольку мы его еще не создали, введите примерный URL-адрес (например, **https://my-first-alexa-bot.herokuapp.com/alexa**). Проигнорируйте остальные параметры и нажмите кнопку **Next** (Далее).
13. На панели сертификатов SSL нажмите кнопку **Next** (Далее) и перейдите к панели **Test** (Тест).
14. На панели **Test** (Тест) мы можем проверить, способна ли Alexa точно распознать высказывания и классифицировать их цели. Перейдите на вкладку **Service Simulator** (Эмуляция службы) – рис. 8.18.

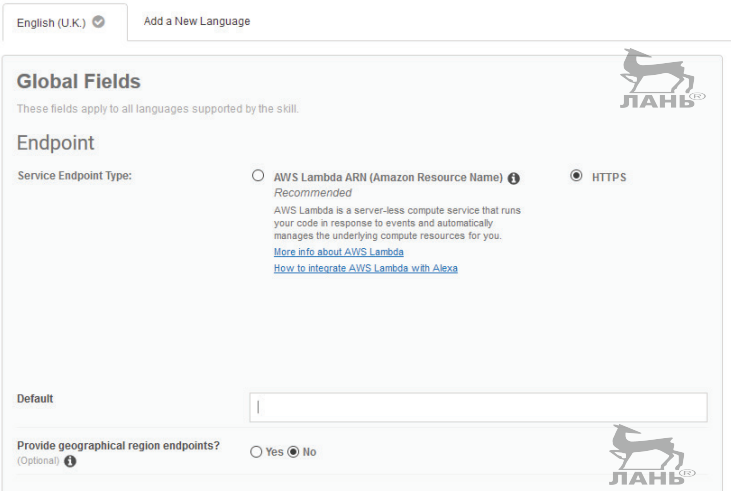


Рис. 8.18

Обратите внимание, что указана конечная точка HTTPS **Server Endpoint**.

15. Введите высказывание на вкладке **Text** (Текст) и нажмите кнопку **Ask *имянавыка*** (Ask *имянавыка*).

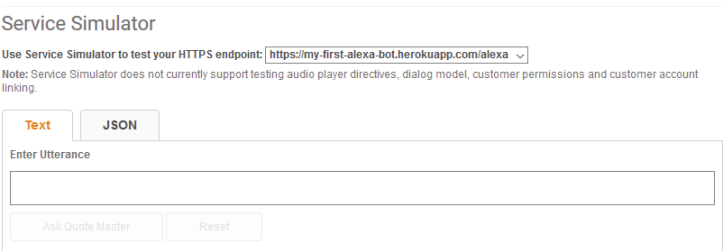


Рис. 8.19

16. Вы увидите, что Alexa проанализировала высказывание и определила цель. Она представлена в виде объекта JSON слева (**Service Request**):

```
.
«request»: {
    “type”: “IntentRequest”,
    “requestId”: “EdwRequestId.a6845d2d-2901-4e8c-b260-3a3a89bd5068”,
```

```
    "intent": {
        "name": "GetQuote",
        "slots": {}
    },
    "locale": "en-GB",
    «timestamp»: «2017-10-11T07:45:24Z»
},
.
```

Справа (**Service Response**) появится сообщение об ошибке, поскольку указанная конечная точка еще не существует. Эта конечная точка должна обрабатывать запрос JSON и отвечать с высказыванием, которое Alexa может выдать пользователю.

Менеджер диалогов – цитаты

Менеджер диалогов – это веб-приложение, которое может получать результирующие цели Alexa и отвечать с помощью бота. Создадим веб-приложение Node.js, которое может принять цели, указанные нами в нашем навыке, и отреагировать соответствующим образом. Чтобы создать модуль, выполните следующие действия:

1. Создайте файл *package.json* с помощью команды `npm init`.

```
> npm init
```

2. Установите модули `express`, `request`, `body-parser` и `hashmap`:

```
> npm install request --save
> npm install express --save
> npm install body-parser --save
> npm install hashmap -save
```

3. Создайте файл *index.js* со следующим содержимым:

```
/*
* Alexa
*/

const request = require('request')
const express = require('express')
const bodyParser = require('body-parser')

const app = express()
```

```
const Hashmap = require('hashmap');

var userContexts = new Hashmap();

app.set('port', (process.env.PORT || 5000))

// Процесс application/x-www-form-urlencoded
app.use(bodyParser.urlencoded({extended: false}))

// Процесс application/json
app.use(bodyParser.json())

// Маршрут
app.get('/', function (req, res) {
    res.send('Hello world, I am Quote Master.')
})

// Включение сервера
app.listen(app.get('port'), function() {
    console.log('running on port', app.get('port'))
}
```

4. Добавьте обработчик маршрута POST для обработки запросов от Alexa. Показанный ниже код содержит заполнители, которые мы заменим в последующих шагах:

```
app.post('/alexa', function (req, res) {
    console.log('Request from Alexa:');
    console.log(req.body);
    //1. добавьте здесь контекстный код

    //2. Получите userIntent из запроса

    //3. установите объект возврата json по умолчанию
    //4. код управления диалогом, чтобы решить, что сказать в ответ на цели поль-
зователя.
})
```

5. Теперь создадим локальную контекстную переменную для хранения всей необходимой нам информации. Эта переменная может использоваться для сбора всех данных, которые должны быть доступны всем сеансам. Чтобы хранить информацию локально во время диалога, мы используем переменную `sessionContext`.

Значение загружается из глобальной переменной userContexts, которая представляет собой хеш-карту, содержащую все контексты сеанса. Alexa каждый раз представляет пользовательские цели с идентификатором сеанса, который одинаков в сеансе. Таким образом, мы можем использовать идентификатор сеанса, чтобы хранить и извлекать контекст в глобальной переменной userContexts.

```
var sessionContext = {
                        'lastUserIntent': 'null',
                        'lastQuote': null
};
var sessionId = req.body.session.sessionId;
if (!userContexts.get(sessionId)){
    userContexts.set(sessionId, sessionContext);
} else {
    sessionContext = usercontexts.get(sessionId);
}
```

6. Мы определяем цель пользователя из запроса (req.body.request.intent.name). Однако если возникает перерыв и когда Alexa завершает сеанс, она возвращает SessionEndedRequest, который необходимо обрабатывать соответствующим образом. Мы обрабатываем его аналогично следующему методу Amazon.StopIntent:

```
var userIntent = '';
if (req.body.request.type === 'SessionEndedRequest'){
    userIntent = 'AMAZON.StopIntent';
} else {
    userIntent = req.body.request.intent.name;
}
console.log('UserIntent:' + userIntent);
```

7. Теперь подготовим дефолтный возвращаемый JSON-код. Ниже приведен его формат. Ключ ssml содержит ответ на языке **Speech Synthesis Markup Language (SSML)** со встроенным текстом. Текст – это то, что Alexa синтезирует в ответ на запрос пользователя.

```
var rjson = {
    "version": "1.0",
    "response": {
```

```
        "shouldEndSession": false,
        "outputSpeech": {
            "type": "SSML",
                "ssml": "<speak> Whole is larger than the sum of its parts!</
speak>"
        }
    }
};
```

8. Дефолтный ответ SSMS должен быть перезаписан ответом, соответствующим запросу пользователя. Поэтому, если пользователь запрашивает цитату, нам нужно получить новую вдохновляющую цитату, а если будет запрошен автор, нам нужно ответить именем автора. Эта логика управления диалогом – то, что мы будем кодировать дальше. Если целью пользователя является получение новой цитаты, мы создаем запрос `GET` для получения новой цитаты из `quotes.rest`. Обратите внимание, как мы сохраняем цитату в контексте, чтобы, если пользователь запрашивает автора цитаты, Alexa узнала ответ из контекста:

```
if (userIntent === 'AMAZON.StopIntent'){
        rjson.response.shouldEndSession = true;
        rjson.response.outputSpeech.ssml = '<speak> Ok.
        Have a good day! </speak>';
        console.log(rjson);
        res.json(rjson);
    }
    else if (userIntent === 'GetQuote'){
        request({
            url: 'http://quotes.rest/qod.json?category=inspire',
            method: 'GET'
        },
        function (error, response, body) {
            // ответ бота
            if (!error && response.statusCode == 200) {
                var q = JSON.parse(body).contents.quotes[0];
                var quote = {'quote': q.quote, 'author': q.author}
                sessionContext.lastQuote = quote;
                usercontexts.set(sessionId, sessionContext);
                rjson.response.outputSpeech.ssml = '<speak>' +
                quote.quote + '</speak>';
                console.log(rjson);
                res.json(rjson);
```

```
            } else {
                console.log('Error: ' + error)
                console.log('Statuscode: ' + response.statusCode)
            }
        });
    }
    else if (userIntent === 'GetAuthor'){

        rjson.response.outputSpeech.ssml =
                '<speak>' + sessionContext.lastQuote.author
                + '</speak>';
        console.log(rjson);
        res.json(rjson);

    }
    else {
        rjson.response.shouldEndSession = true;
        rjson.response.outputSpeech.ssml = '<speak>
        Sorry. I did not get you! </speak>';
        console.log(rjson);
        res.json(rjson);
    }
```

9. Теперь, когда у нас есть веб-приложение для управления диалогом, выложим его в облако на сервер, настроенный на панели **Skill Configuration** (Конфигурация навыков):

```
> git init
> git add .
> git commit -m alexa-quote-bot
> heroku create my-first-alexa-bot
> git push heroku master
```

   Выполнив предыдущие команды в консоли, вы запустите менеджер диалогов на странице **my-first-alexabot.herokuapp.com/alexa**. Именно здесь Alexa отправит запрос `POST`. Ответ будет проанализирован, и высказывание SSMS будет синтезировано для пользователя.
10. Теперь вернитесь на панель **Test** (Тест), введите примерное высказывание (например, `tell me a quote`) и нажмите кнопку **Ask Quote Master** (Создать запрос) – рис. 8.20.
11. Продолжайте диалог. Введите строку `who said that` и нажмите кнопку **Quote Master** (Создать запрос) – рис. 8.21.

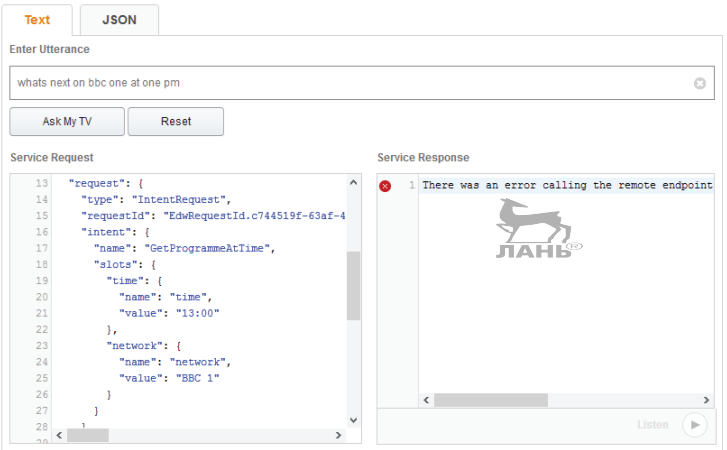


Рис. 8.20

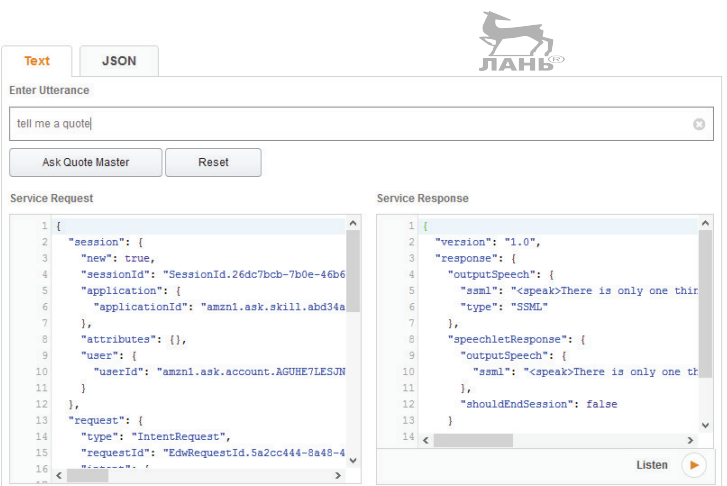


Рис. 8.21

Этот запрос будет определен как цель `GetAuthor`, и на него будет отвечено сообщением с именем автора цитаты.

12. Обратите внимание, что идентификатор `sessionId` для обоих предыдущих запросов одинаковый. Теперь нажмите кнопку **Reset** (Сброс), чтобы сбросить диалог. Любой новый запрос будет иметь другой идентификатор `sessionId`.

Тестирование на Echo

Поздравляю! Ваш первый навык Alexa готов. Вы можете проверить его на своем динамике Echo, если он у вас есть. Все навыки, которые вы разрабатываете, будут добавлены в программное обеспечение Alexa по умолчанию. Вы можете просмотреть их на панели управления Alexa (**alexa.amazon.com**) — перейдите на вкладку **Skills** (Навыки) в меню **Your Skills** (Ваши навыки) в правом верхнем углу.

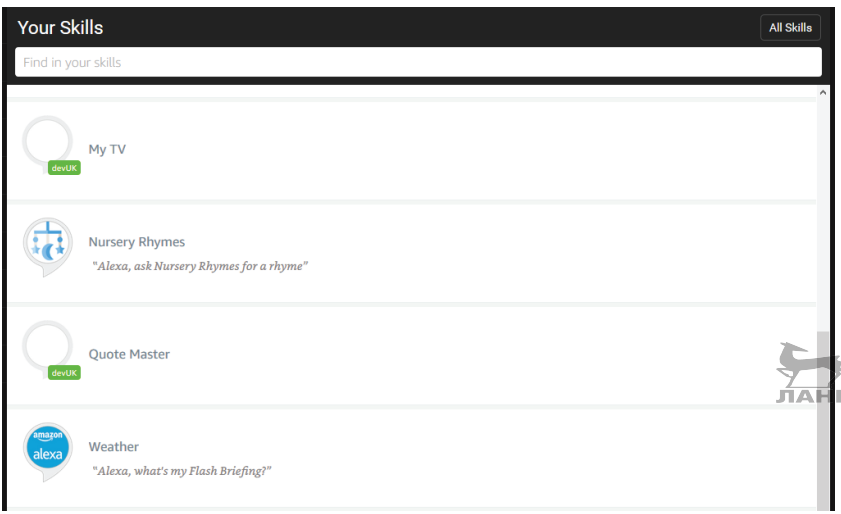


Рис. 8.22

Обратите внимание, что Alexa после прочтения ожидает цитаты. Благодаря этому пользователь может отправить следующий запрос. Параметр активируется установкой значения `false` по умолчанию для параметра `shouldEndSession`. Значение `true` присваивается ему, только если пользователь говорит `stop` или молчит, пока Alexa ждет ответа.

Телегид

Перейдем к более сложному проекту: создание навыка Alexa для получения информации о телепрограмме. В этой задаче мы разработаем навык, который позволит Alexa отвечать на вопросы о телепрограмме. Например, Alexa сможет ответить на следующие вопросы:

- Какая следующая передача на канале BBC One?
- Что идет на канале ITV сейчас?
- О чем эта передача?
- Что покажут на ITV в 14:00?

Построение модели взаимодействия

В дополнение к концепциям, которые мы изучили при разработке навыка цитат, в этой задаче мы рассмотрим более сложные диалоговые элементы – слоты. Рассмотрим семиэтапный процесс создания нового навыка под названием `My TV`.

1. Заполните информацию о навыке (**Skill Information**) и перейдите на страницу модели взаимодействия.
2. Щелкните мышью по кнопке **Launch Skill Builder** (Запустить Skill Builder) и откройте панель управления. Как и прежде, вы увидите, что по умолчанию в модель уже добавлены три цели.
3. Как вы могли заметить, в предыдущих вопросах есть элементы, которые можно перечислить и классифицировать как сущности. Например, названия BBC One, BBC Two и ITV могут быть классифицированы как телесети (телеканалы). В Alexa это знание может быть передано в систему в виде слотов. Чтобы создать слот, выполните следующие шаги: в разделе **Intents** (Цели) в меню слева вы найдете раздел **Slot Types** (Слоты). Нажмите кнопку **Add+** (Добавить+).

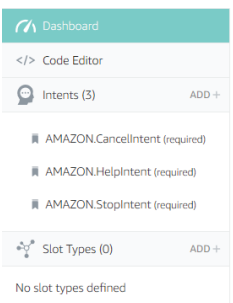


Рис. 8.23

4. Как и в случае с целями, существует два типа слотов – встроенные (предустановленные) и пользовательские слоты – рис. 8.24. Мы создадим пользовательский слот под названием `Network`. Щелкните мышью по кнопке **Create Slot Type** (Создать слот) – рис. 8.25.

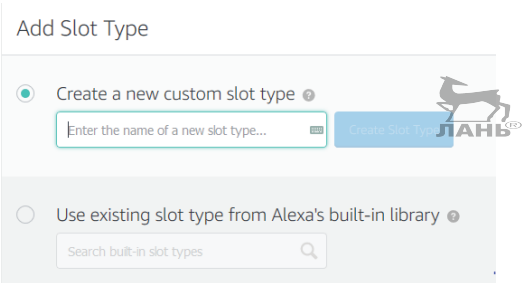


Рис. 8.24

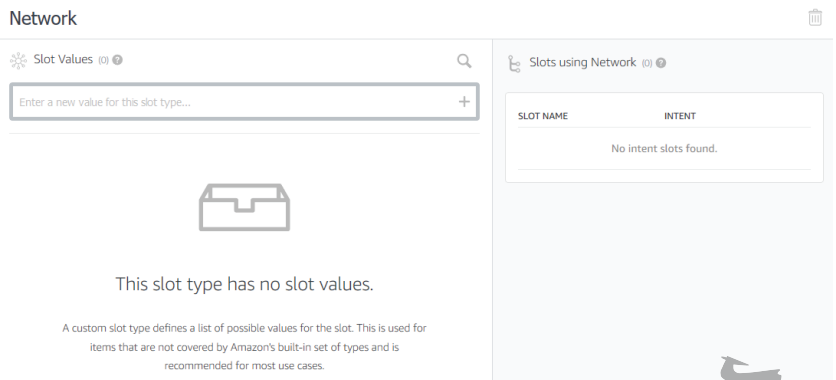


Рис. 8.25

5. На этой странице нам нужно указать все телеканалы, которые программное обеспечение Alexa должно идентифицировать во время диалога с пользователем. Составим небольшой список: `Channel 4`, `BBC One`, `BBC Two` и `ITV`. Позднее можно добавить другие каналы.
6. Добавим первый канал: `BBC One`. Щелкните мышью по ссылке **+** в конце текстового поля. Добавьте все варианты значения `BBC One`, `BBC 1`, `BBC one` и т. д. как синонимы.
7. Добавьте все другие телеканалы таким же образом – рис. 8.26.
8. Теперь создадим цели, которые используют слот `Network`. Начнем с `GetNextProgramme`. Эта цель соответствует всем высказываниям, когда пользователь запрашивает следующую программу на данном телеканале. Высказывание примерно выглядит так: `what is the next programme on BBC One?`. Нажмите кнопку **Intents+** (Цели+) и добавьте образцы высказываний. Но вместо упоминания

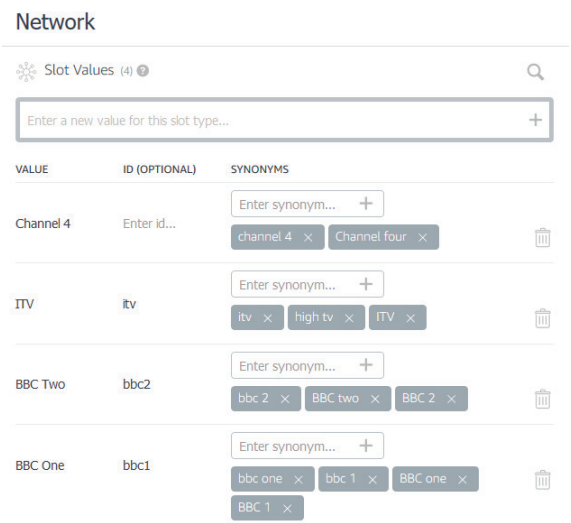


Рис. 8.26

названия телеканала в высказывании нам нужно использовать слот (используя нотацию {}), как показано ниже:

```
whats on {network} now?
what the next show on {network}
whats next on {network}?
```

Когда вы начнете вводить название слота в высказывании, то увидите список слотов, как показано ниже – рис. 8.27.

Несмотря на то что для выполнения задачи требуется слот `network`, пока не настраивайте его обязательным (флажок **REQ**) – рис. 8.28.

9. Добавим еще одну цель – `GetProgrammeAtTime`. Она будет использоваться для тех высказываний, когда пользователь запрашивает название программы на определенном телеканале в указанное время. Примером может служить высказывание `what show is on BBC One at 2pm`.

   Создайте новую цель и добавьте следующие примеры высказываний:

```
what show is on {network} at {time}
whats on {network} at {time}
```

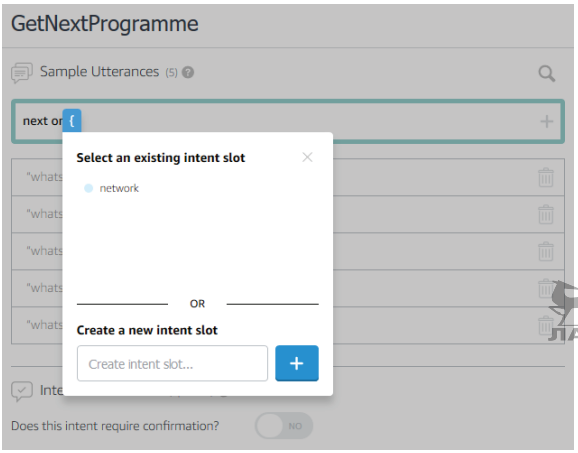


Рис. 8.27

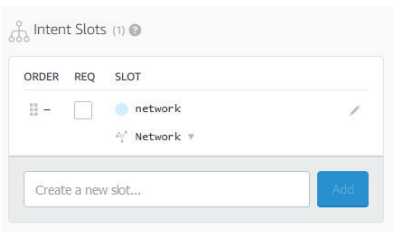


Рис. 8.28

Поскольку время (значение `time`) еще не определено как слот, нам нужно будет определить его позже. Все слоты, связанные с целью, будут перечислены справа, как показано ниже:

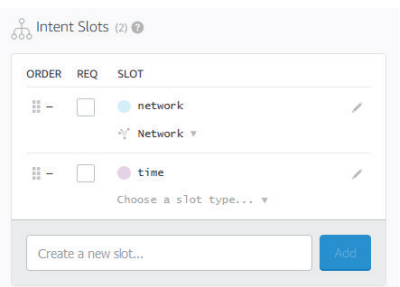


Рис. 8.29

10. Щелкните мышью по ссылке **Choose a slot type** (Выбрать тип слота) и выберите пункт **AMAZON.TIME**. Это один из предопределенных слотов. Установите флажок **REQ**, чтобы слот был обязательным и требовал запроса/ответа, как это было со слотом `network`.
11. Добавьте другие высказывания и сохраните модель.
12. Добавьте последнюю цель – `GetDescription`. Эта цель предназначена для высказываний, когда пользователь хочет узнать дополнительную информацию о передаче. Например, пользователь мог бы сказать `whats the show about` в качестве дополнительного вопроса для двух предыдущих целей.

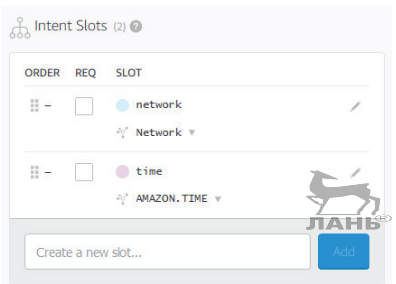


Рис. 8.30

Здесь нет слотов, потому что мы планируем сохранить информацию, требуемую в контексте диалога.

13. Добавьте другие высказывания для цели и сохраните модель:

```
tell me more about the show
describe the show
what is the show about
whats it about
```

14. Проверим модель так же, как мы делали это с навыком цитат. На панели **Test** (Тест) введите несколько высказываний и проверьте, как Alexa анализирует их, – рис. 8.31.

Поскольку у вас нет сервера, настроенного на ответ, вы увидите ошибку.

Настройка сервера для навыка телегида

Теперь, когда у нас есть модель взаимодействия, нам нужен менеджер диалогов, который сможет реагировать на цели пользователя. Настроим сервер для ответа пользователю.

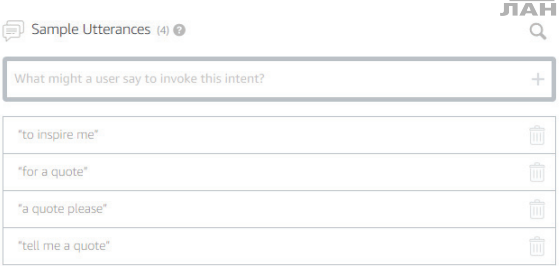


Рис. 8.31

1. На панели **Skill Configuration** (Конфигурация навыков) выберите конфигурацию и щелкните мышью по пункту **HTTPS**.
2. Создайте проект Node.js с помощью команды `npm` и добавьте библиотеки по умолчанию, как это было сделано для сервера цитат, о котором говорилось ранее.
3. Создайте файл *index.js* и импортируйте библиотеки `request`, `express`, `body-parser` и `hashmap`.
4. Создайте веб-сервер с помощью библиотеки Express, как вы делали это ранее:

```
const request = require('request')
const express = require('express')
const bodyParser = require('body-parser')

const Hashmap = require('hashmap');

var usercontexts = new Hashmap();

const app = express()

app.set('port', (process.env.PORT || 5000))

// Процесс application/x-www-form-urlencoded
app.use(bodyParser.urlencoded({extended: false}))

// Процесс application/json
app.use(bodyParser.json())

app.use(express.static('public'))

// Маршрут
```

```
app.get('/', function (req, res) {
    res.send('Hello world, I am the TV Guide.')
})
```

5. Нам нужна другая библиотека для обработки даты и времени, `moment`. Установите ее с помощью команды `npm install` и добавьте показанный ниже код в файл *index.js*. Он необходим, чтобы получить сегодняшнюю дату для извлечения сегодняшней телепрограммы:

```
const moment = require('moment')

var todaysSchedule = {};
const now = new Date();
var date = now.getDate();
if (date < 10) { date = '0' + date; }
var month = now.getMonth() + 1;
if (month < 10) { month = '0' + month; }
const todaysDate = now.getFullYear() + '-' + month + '-' + date;

loadTodaysSchedule(todaysDate);
```

6. Создайте функцию `loadTodaysSchedule()`, чтобы получить сегодняшнюю телепрограмму. Ее можно сохранить локально и запрашивать, когда требуется ответить на вопросы пользователей. Обратите внимание, что программа не будет меняться каждый день автоматически.

```
function loadTodaysSchedule(todaysDate){
    var url = 'http://api.tvmaze.com/schedule?country=GB&date='
              + todaysDate;
    console.log('URL:' + url);
    request({
        url: url,
        method: 'GET'
    },
    function (error, response, body) {
        // Ответ бота
        if (!error && response.statusCode == 200) {
            // Вывод тела ответа
            todaysSchedule = JSON.parse(body);
            console.log('Todays Schedule: ' +
                        todaysSchedule.length);
```

```
            console.log(todaysSchedule);
        } else {
            console.log('Error: ' + error)
            console.log('Statuscode: ' + response.statusCode)
        }
    });
}
```

7. Ниже показан шаблон для конечной точки ответа:

```
app.post('/alexa', function (req, res) {
    console.log('Request from Alexa:');
    console.log(req.body);
    // 1. Добавьте здесь контекстный код.

    // 2. Получите userIntent из запроса.

    // 3. Установите объект возврата json по умолчанию.
    // 4. Код управления диалогом, чтобы решить, что сказать в ответ на цели
пользователя.
})
// Включение сервера
app.listen(app.get('port'), function() {
    console.log('running on port', app.get('port'))
})
```

8. Сохраним контекст и загрузим его по аналогии с навыком цитат. Мы будем сохранять информацию о передаче, которую запросил пользователь, чтобы ее можно было описать в следующем порядке:

```
var sessionContext = {
                        'show': null
                      };
var sessionId = req.body.session.sessionId;
if (!usercontexts.get(sessionId)){
    usercontexts.set(sessionId, sessionContext);
} else {
    sessionContext = usercontexts.get(sessionId);
}
```

9. Извлечем цель пользователя из запроса и приготовим ответ по умолчанию в формате JSON:

```
var userIntent = '';
    if (req.body.request.type === 'SessionEndedRequest'){
        userIntent = 'AMAZON.StopIntent';
    } else {
        userIntent = req.body.request.intent.name;
    }
    console.log('UserIntent:' + userIntent);
    var rjson = {
        "version": "1.0",
        "response": {
                "shouldEndSession": false,
            "outputSpeech": {
                "type": "SSML",
                    "ssml": "<speak>Hello, I am your TV Guide!</speak>"
        }
    }
};
```

10. Ниже показан код для ответа на цели пользователя. Ответ пользователя в формате JSON сбрасывается с информацией, полученной из телепрограммы:

```
if (userIntent === 'AMAZON.StopIntent'){
    rjson.response.shouldEndSession = true;
    rjson.response.outputSpeech.ssml =
          '<speak> Ok. Have a good day! </speak>';
    console.log(rjson);
    res.json(rjson);
}
else {
    if (userIntent === 'GetNextProgramme'){
        var network =
        req.body.request.intent.slots.network.value;
        var e = getNextProgrammeInNetwork(network);
        sessionContext.show = e;
        usercontexts.set(sessionId, sessionContext);
        rjson.response.shouldEndSession = false;
        rjson.response.outputSpeech.ssml =
          '<speak> The next show on ' + network + ' is ' +
          e.show.name + ' at ' + e.airtime + '! </speak>';
        console.log(rjson);
        res.json(rjson);
    }
    else if (userIntent === 'GetProgrammeAtTime'){
```

```
        var network =
        req.body.request.intent.slots.network.value;
        var time = req.body.request.intent.slots.time.value;
        var e = getProgrammeAtTime(network, time);
        sessionContext.show = e;
        usercontexts.set(sessionId, sessionContext);
        rjson.response.shouldEndSession = false;
        rjson.response.outputSpeech.ssml = '<speak> On '
        + network +
          ' at ' + e.airtime +
          ' is ' + e.show.name +
          '! </speak>';
        console.log(rjson);
        res.json(rjson);
    }
    else if (userIntent === 'GetDescription'){
        var e = sessionContext.show;
        var desc = getDescription(e);
        rjson.response.shouldEndSession = true;
        rjson.response.outputSpeech.ssml = '<speak> ' + desc +
          ' </speak>';
        console.log(rjson);
        res.json(rjson);
    }
    else if (userIntent === 'Thank'){
        rjson.response.shouldEndSession = true;

        rjson.response.outputSpeech.ssml = '<speak>
        You are most welcome. Goodbye! </speak>';
        console.log(rjson);
        res.json(rjson);
    }
    else {
        rjson.response.shouldEndSession = true;
        //rjson.response.outputSpeech.ssml = '<speak> Ok.
        //Have a good day! </speak>';
        console.log(rjson);
        res.json(rjson);
    }
}
```

11. Наконец, часть кода, которая извлекает необходимую информацию из расписания:

```
function getNextProgrammeInNetwork(networkName){
    console.log('Getting next programme in : ' + networkName);
    for (var i=0; i < todaysSchedule.length; i++){
        var e = todaysSchedule[i];
        var showtime = moment(e.airdate + ' ' + e.airtime);
        moment().format();

        var a = moment();
        var b = moment(showtime);
        if (a.diff(b, 'minutes') < 0){
            if (e.show.network.name === networkName){
                console.log(e.show.name + ' on ' +
                        e.show.network.name + ' at ' + e.airtime);
                return e;
            }
        }
    }
}

function getProgrammeAtTime(networkName, time){
    console.log('Getting next programme in : ' + networkName + '
                at ' + time);
    for (var i=0; i < todaysSchedule.length; i++){
        var e = todaysSchedule[i];
        if (e.show.network.name === networkName){
            var showtime = moment(e.airdate + ' ' + e.airtime);
            var requestedtime = moment(e.airdate + ' ' + time);
            moment().format();

            var a = moment(requestedtime);
            var b = moment(showtime);
            if (a.diff(b, 'minutes') > -30 && a.diff(b, 'minutes')
                < 30 ){
                console.log(e.show.name + ' on ' +
                    e.show.network.name + ' at ' + e.airtime);
                return e;
            }
        }
    }
}

function getDescription(episode){
    console.log('Getting episode info:' + episode.show.name);
    console.log(episode.show.summary);
```

```
    return episode.show.summary;
}
```

12. Теперь разместим это веб-приложение в облаке:

```
> git init
> git add .
> git commit -m alexa-tv-bot
> heroku create my-alexa-tv-bot
> git push heroku master
```

После того как вы опубликовали веб-приложения в облаке, выполнив команды, показанные выше, у вас будет готов менеджер диалогов, работающий на сайте **my-alexa-tvbot.herokuapp.com/alexa**. Именно сюда программное обеспечение Alexa отправит запрос `POST`. Ответ будет проанализирован, и для пользователя будет синтезировано SSMS-высказывание.

Тестирование навыка телегида

Вернемся к панели **Test** (Тест) и протестируем некоторые примеры высказываний, чтобы увидеть, как они работают.

1. Введите текст `whats next on bbc one` и нажмите кнопку **Ask My TV** (Спросить My TV).

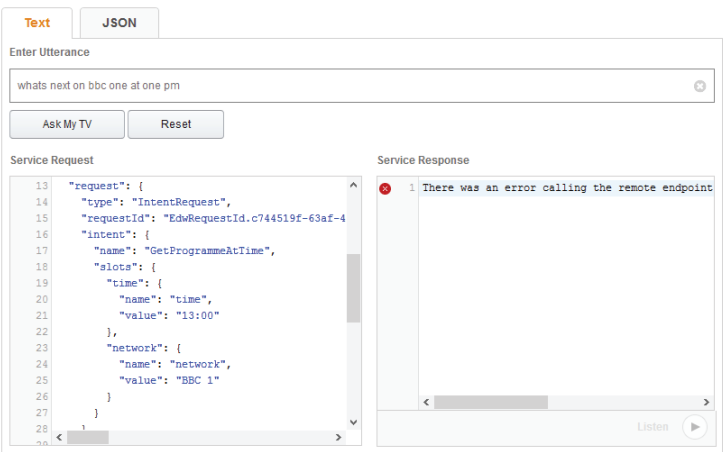


Рис. 8.32

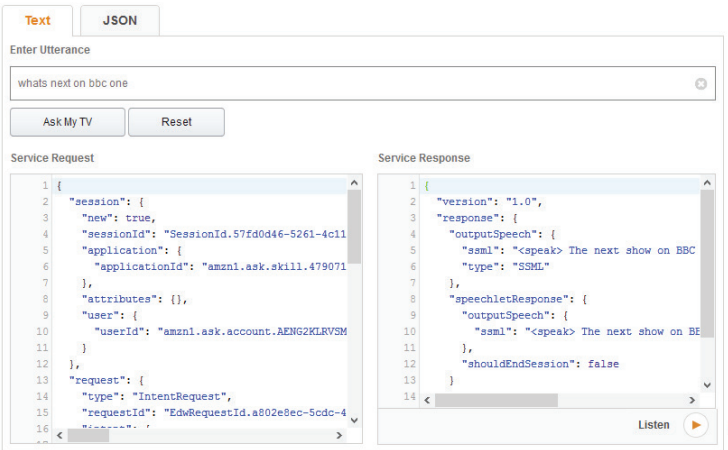


Рис. 8.33

2. Введите текст `whats on bbc two at 4pm` и нажмите кнопку **Ask My TV** (Спросить My TV).

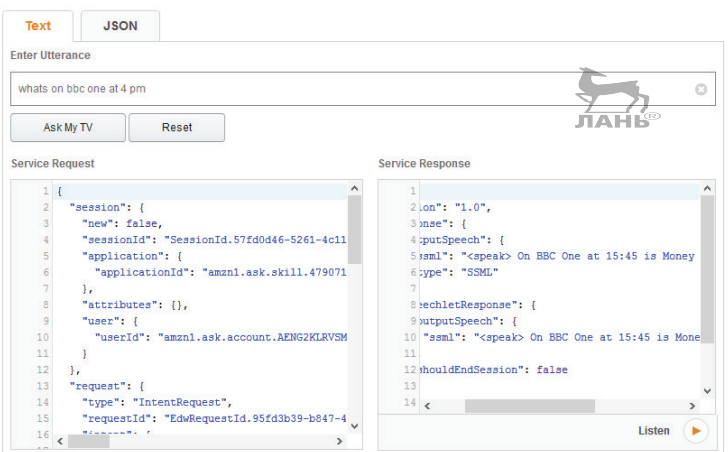


Рис. 8.34

3. Наконец, попробуем получить дополнительную информацию о телепередаче с помощью запроса, показанного ниже. Помните, что не нужно нажимать кнопку **Reset** (Сброс) перед выполнением запроса – так вы можете сбросить контекст диалога.

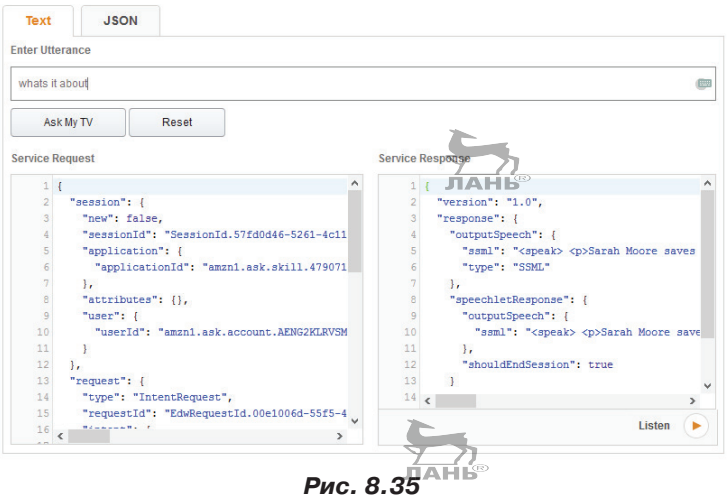


Рис. 8.35

Ответ, который вы получите, будет кратким описанием телепередачи, которое программное обеспечение Alexa извлекло ранее. Ниже представлен фрагмент вывода в формате JSON:

```
“outputSpeech”: {
        “ssml”: “<speak> <p>Sarah Moore saves three items from the tip in each
episode
        and transforms them into a much more valuable piece, before returning the
profits
        back to the original owners of the item.</p> </speak>”,
        ”type”: ”SSML”
    },
```

4. Проверив навык в эмуляторе, проверьте его на реальном устройстве Echo. Помните, что перед запросом вам нужно указать имя вызова. Например, `Alexa, ask my TV, whats next on BBC One`.

Встроенные цели и слоты

Программное обеспечение Alexa содержит библиотеку встроенных целей и слотов, которые могут использоваться в любых создаваемых вами навыках. Встроенные цели имеют свои преимущества. С точки зрения разработчика, вам не нужно предоставлять примерные высказывания, и всякий раз, когда Alexa получает

обновление с типовыми высказываниями, ваш навык расширяется. Кроме того, так пользователям проще запомнить простой набор фраз на основе навыков, чтобы эффективно взаимодействовать с динамиком Echo.

В таблице показан набор стандартных встроенных целей, которые вы можете использовать:

Встроенные цели	Описание
`Amazon.YesIntent`	*Если пользователь говорит* `yes`, `yeah` *или что-то подобное*
`Amazon.NoIntent`	*Если пользователь говорит* `no`, `no thanks` *или что-то подобное*
`Amazon.CancelIntent`	*Если пользователь просит отменить текущую задачу, например* `cancel`, `never mind` *или* `forget it`
`Amazon.HelpIntent`	*Если пользователь просит о помощи (*`help`, `can you help me`*)*
`Amazon.StopIntent`	*Если пользователь говорит* `stop`, `off` *или* `shut up`

Программное обеспечение Alexa также включает несколько стандартных слотов, которые вы можете использовать:

Встроенные слоты	Описание
`AMAZON.DATE`	*Любое описание даты может быть выбрано с использованием этого слота. Например,* `today (2017-10-10)`, `now`, `this weekend (2017-W40-WE)`, `next year (2018)` *или* `this winter`. *Эти фразы будут преобразованы в формат даты ISO 8601*
`AMAZON.TIME`	*Любая отсылка ко времени, например* `two o clock`, `tomorrow morning` *или* `noon`, *будет выбрана и сохранена в формате времени ISO 8601*
`AMAZON.DURATION`	*Длительность, например* `ten minutes`, `six hours` *или* `seven years`. *Преобразованная информация будет представлена в формате длительности ISO 8601*
`AMAZON.NUMBER`	*Любое число, например* `one`, `one hundred` *или* `two three four`
`AMAZON.FOUR_DIGIT_NUMBER`	*Любое четырехзначное число*

Существуют другие слоты, которые можно использовать, например `AMAZON.AggregateRating` (для таких рейтингов, как `best`, `five star`, `top ten`), `AMAZON.Animal` (для животных), `AMAZON.AT_CITY` (для городов по всему миру), `AMAZON.Color` (для цвета) и `AMAZON.Country` (для стран).

Изучите полный список слотов на странице **developer.amazon.com/docs/custom-skills/slot-type-reference.html**.

Заключение

Прекрасно! Я верю, что в этой главе вам понравилось конструировать на платформе Alexa. Мы изучили основы построения целей – навыки и слоты. Мы исследовали процесс конструирования модели взаимодействия на платформе разработки Alexa, а также создали веб-приложение менеджера диалогов в облаке. Мы создали два навыка: один для цитат, а другой для телепрограммы, – предоставляя широкие возможности для изучения и понимания различных элементов развития навыков Alexa. Я бы рекомендовал вам рассматривать эту главу в качестве начала и продолжить изучение. В магазине Alexa насчитывается около 10 тыс. навыков, и это количество продолжает увеличиваться. Благодаря использованию умных колонок в домах, которые, по прогнозам, будут только распространяться, в ближайшие годы будет огромная потребность в пользовательских навыках для платформы Alexa. В следующей главе мы рассмотрим, как конструировать действия для Ассистента Google и вызывать их в Google Home.

Ссылки

- Документация по Amazon Alexa: **developer.amazon.com/docs/askoverviews/buildskills-with-the-alexa-skills-kit.html**.

Глава 9. Мой верный слуга

Теперь, добравшись до последней главы книги, давайте сделаем что-то реально интересное. Как насчет того, чтобы создать бота – верного слугу? Допустим, что бот должен выполнять повседневные задачи, такие как организация списка дел, напоминания о задачах и встречах и т. д. В предыдущей главе мы разработали навык Alexa для просмотра телепрограммы. В этой главе мы рассмотрим процесс создания навыков для платформы Google Assistant. Ассистент Google – нечто сродни Apple Siri и Amazon Alexa, только компании Google. Вы можете взаимодействовать с Ассистентом Google различными способами, например через мобильное приложение Allo или его веб-версию, а также через динамик Google Home.

Google Home – это умный динамик, как и устройство Amazon Echo, только оснащенный Ассистентом Google. Другими словами, Google Home – это физическое воплощение Ассистента Google. Очень похоже на отношение Alexa к Echo и Siri к iPhone/iPad. Google Home – непосредственный конкурент Amazon Echo и в настоящее время захватил 24% рынка. Как и Alexa, навыки (или действия, как их называют в Google) ассистента могут быть развиты.

В этой главе мы создадим действие для Ассистента Google. Разрабатывая верного слугу, начнем с интеграции в него онлайн-службы, которую можно использовать для ежедневного управления вашими повседневными задачами. Мы рассмотрим, как действие может быть реализовано для взаимодействия с этой службой для добавления и извлечения задач. Во-первых, рассмотрим онлайн-сервис под названием Todoist и то, как добавлять и извлекать задания с помощью REST API. Затем мы создадим агента с помощью набора инструментов Dialogflow и интегрируем его в Ассистента Google в виде действия. Наконец, мы интегрируем в действие службу Todoist, завершив работу.

В ходе этой главы вы:

- сконструируете агента Dialogflow для обработки списков дел;
- интегрируете службу Todoist с помощью вебхуков;

- создадите действия Google Actions и интегрируете их в Ассистента Google;
- проверите действие на динамике Google Home.

Служба Todoist

Todoist – это онлайн-сервис для хранения и организации списков действий, которые вы планируете выполнить в ближайшем будущем. Пользователи могут добавлять задачи в свою учетную запись, организовывать их по проектам и назначать сроки и напоминания о своих задачах. Учетную запись можно получить на своем веб-сайте, а также в приложении на устройствах Android и iOS. Доступ к сервису Todoist можно получить с помощью API-интерфейса разработчика.

Рассмотрим две основные службы: добавление задач и извлечение списка дел на день.

Получение ключа

Для доступа к Todoist нам нужен ключ API. Чтобы получить его, выполните следующие действия.

1. Перейдите на страницу **todoist.com**.
2. Зарегистрируйте учетную запись.
3. Войдите в учетную запись и взгляните на панель.
4. Добавьте задачу, нажав кнопку **Add Task** (Добавить задачу).

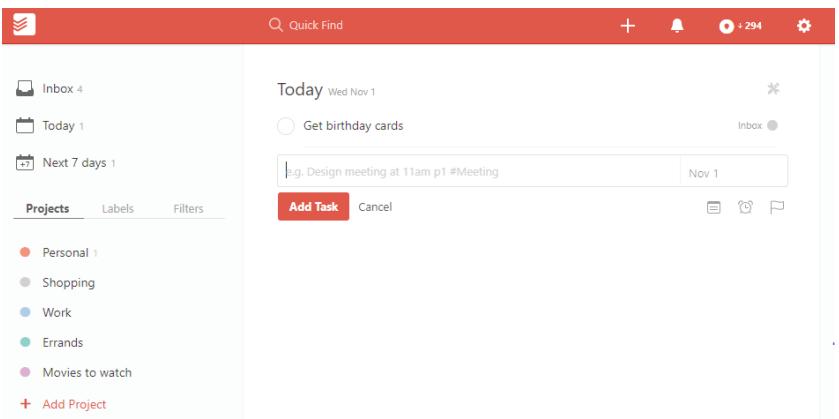


Рис. 9.1

Обратите внимание, что задачи могут быть организованы в проекты. По желанию вы можете создать новый проект. По умолчанию задачи будут добавлены в каталог **Inbox** (Входящие) и будут содержать дату создания.

5. Выберите задачу, которую вы только что создали, и изучите параметры:

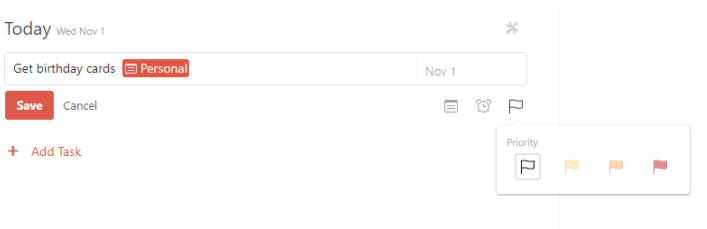


Рис. 9.2

Вы можете изменить сроки, проект, к которому принадлежит задача, ее приоритет и задать напоминания.

6. Теперь, когда мы изучили возможности служебной платформы, добавим ключ API в учетную запись. Щелкните мышью по значку настроек в правом верхнем углу:

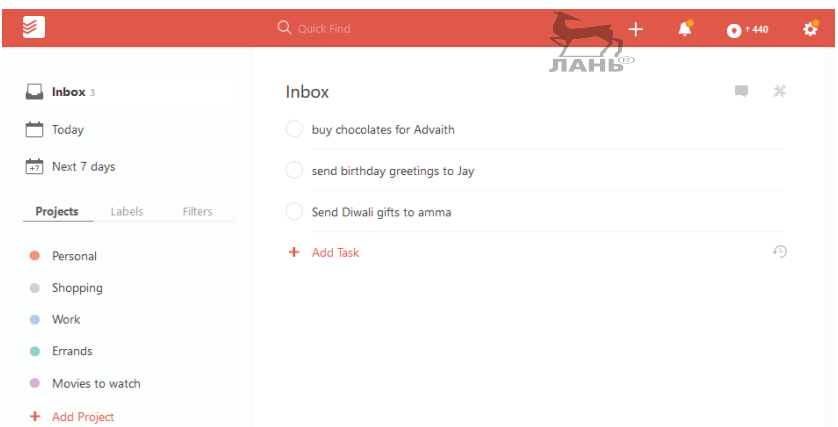


Рис. 9.3

7. Щелкните мышью по ссылке **Settings** (Настройки).
8. В боковом меню щелкните мышью по ссылке **Integrations** (Интеграция).

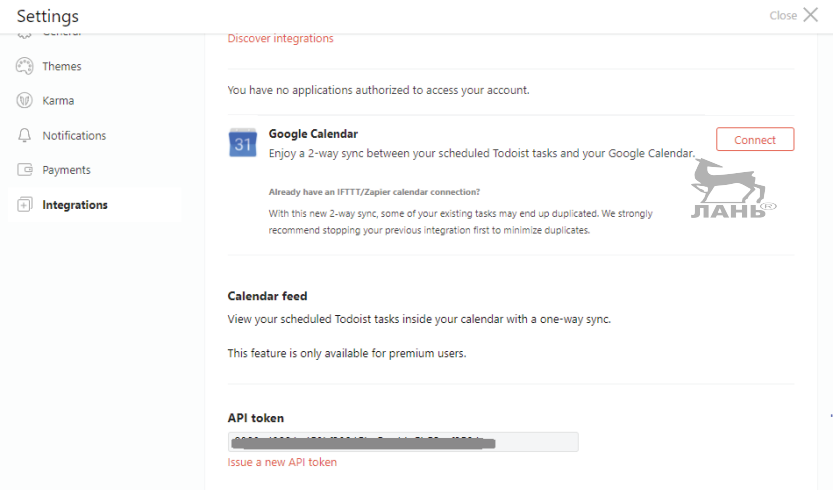


Рис. 9.4

9. Прокрутите страницу вниз до раздела **API token** (Токен API) – рис. 9.4.

   Скопируйте свой токен и сохраните его.

Теперь напишем программу Node.js для добавления элементов и получения списков.

Добавление задачи

Используя ранее полученный ключ, мы можем добавлять и извлекать список задач. Ниже показан код Node.js для добавления задачи в список дел:

```
var task = {'type': 'item_add',
    'temp_id': uuid.v4(),
    'uuid': uuid.v4(),
    'args': {
        'content': taskDescription
    }};
var url = 'https://todoist.com/api/v7/sync?token=' +
           todoist_token +
           '&sync_token=*&resource_types=["items"]&commands=['
           + JSON.stringify(task) +']';
           console.log('URL:' + url);
request({
```

```
    url: url,
    method: 'GET'
},
function (error, response, body) {
    //ответ бота
    if (!error && response.statusCode == 200) {
        console.log(JSON.parse(body));
    } else {
        console.log('Error: ' + error)
        console.log('Statuscode: ' + response.statusCode)
    }
});
```

Получение всех задач на день

Чтобы получить все задачи в списке дел за день, используйте следующий код Node.js:

```
var url = 'https://todoist.com/api/v7/sync?token='
         + todoist_token +
           '&sync_token=*' +
           '&resource_types=["items"]';
console.log('URL:' + url);
var resp = syncrequest('GET', url);
var allItems = JSON.parse(resp.getBody('utf8')).items;
var allItemsSummary = '';
for (var i = 0; i < allItems.length; i++){
    allItemsSummary += allItems[i].content;
    if (i < allItems.length - 2){
        allItemsSummary += ', ';
    }
    else if (i == allItems.length - 2) {
        allItemsSummary += ' and ';
    }
    else {
        allItemsSummary += '.';
    }
}

response = 'You have ' + allItems.length + ' in your list. ' + allItemsSummary;
console.log(response);
```

Показанный код позже будет встроен в веб-службу, вызываемую из платформы Dialogflow для получения ответного высказывания,

сгенерированного динамически вебхуком. Чтобы изучить весь спектр возможных действий в Todoist, обратитесь к документации по адресу **developer.todoist.com.**

Создание действия

Чтобы создать действие для Ассистента Google, нам нужно начать с Dialogflow. Общие шаги следующие.

1. Создайте агента Dialogflow.
2. Создайте веб-приложение, которое обращается к службе Todoist.
3. Интегрируйте агента в действие Google.
4. Протестируйте результат в эмуляторе.
5. Проверьте результат на устройстве Google Home.

Создание агента Dialogflow

Чтобы создать агента Dialogflow, выполните следующие действия.

1. Перейдите на страницу **dialogflow.com** и авторизуйтесь в системе, используя свою учетную запись Google.
2. Откройте раскрывающийся список слева и выберите пункт **Create new agent** (Создать нового агента).

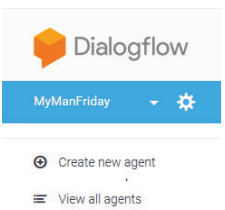


Рис. 9.5

3. Заполните форму и щелкните мышью по кнопке **Save** (Сохранить) – рис. 9.6.
4. Будет создан новый проект Google, связанный с агентом. Вы можете просмотреть настройки, щелкнув мышью по значку **Settings** (Настройки) рядом с именем агента в раскрывающемся списке – рис. 9.7.
5. Добавим четыре цели для агента:
   - `greet`
   - `add_task`

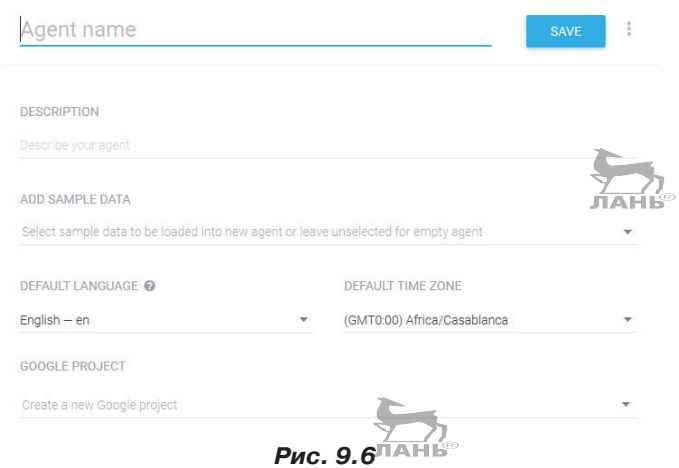


Рис. 9.6

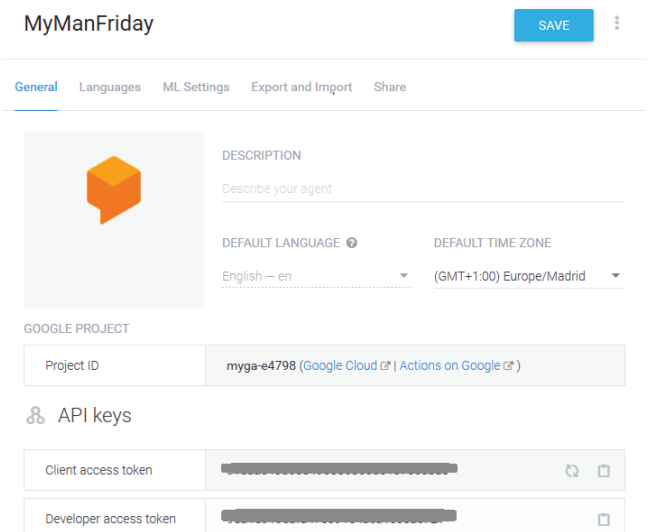


Рис. 9.7

- `inform_task_description`
- `list_all_tasks`

Предположим, что пользователь начнет разговор с приветствия (цель `greet`), за которым следует запрос либо добавить задачу

(add_task), либо перечислить все задачи на день (list_all_tasks). В следующем запросе на добавление задачи агент просит пользователя описать задачу, на что пользователь ответит описанием задачи (inform_task_description).

6. Добавьте несколько примеров высказываний к цели greet:
   - hi
   - hello there
7. Теперь перейдем к цели add_task. Эта цель касается добавления задачи в список. Добавьте несколько высказываний:
   - i want to add a task
   - add new task
   - add a task to my list

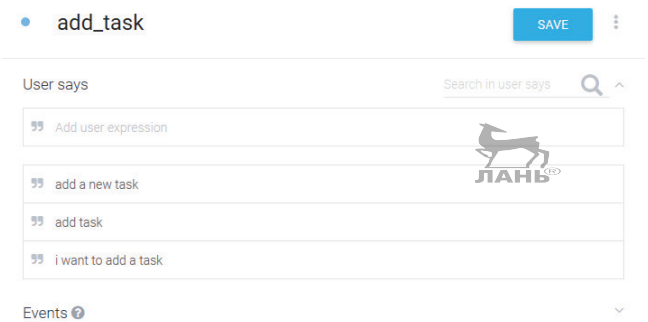


Рис. 9.8

8. Для достижения цели нужно запросить описание задачи, которая должна быть добавлена. Давайте так и сделаем, а заодно зададим контекст asked-for-task – рис. 9.9.
9. Пользователь, как ожидается, ответит описанием задачи. Нам нужно записать это описание и передать его в веб-приложение, которое разместит его в списке Todoist. Выполним это с целью inform_task_description в контексте asked-for-task – рис. 9.10.
10. Все высказывания в цели inform_task_descriptions можно рассматривать как описание задачи, и поэтому они передаются в список Todoist как есть. Чтобы захватить высказывание в параметре целиком (например, task), выберите высказывание целиком и укажите пункт @sys.any в раскрывающемся списке, как показано на рисунке – рис. 9.11.

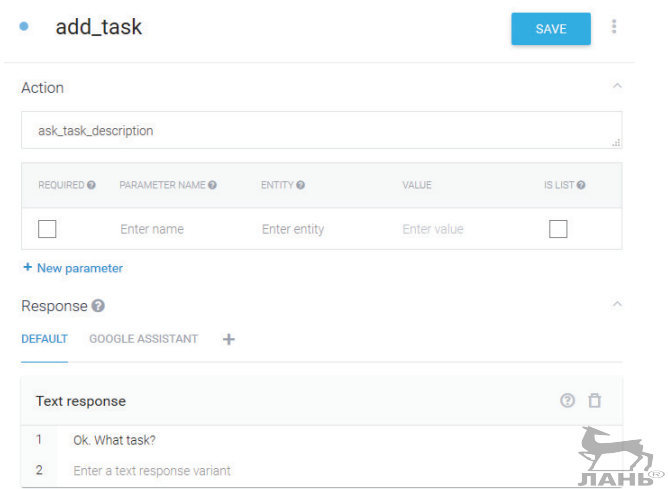


Рис. 9.9

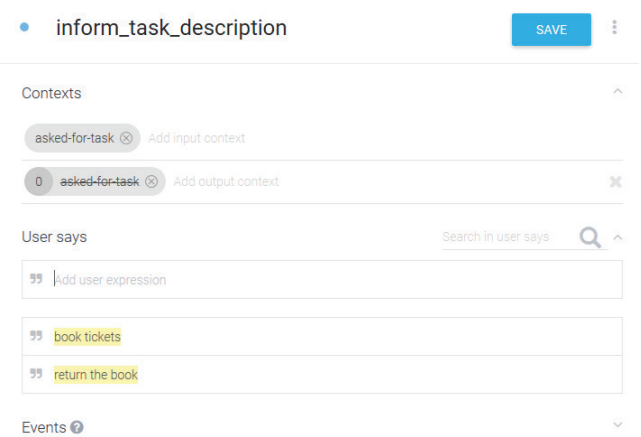


Рис. 9.10

11. Так будет создан параметр, называемый `any`, связанный с целью. Давайте переименуем его. Щелкните мышью по пункту `any` и измените название на `task` – рис. 9.12.
12. Сделайте то же самое для нескольких других примеров высказываний. Ниже приведены описания задачи:

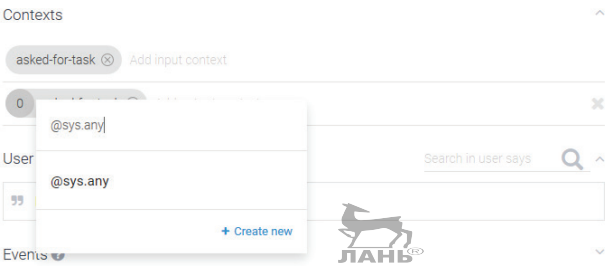


Рис. 9.11

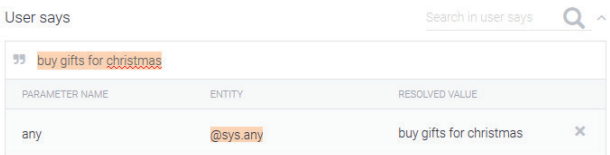


Рис. 9.12

- `buy gifts for christmas`
- `send a birthday card to Jones`
- `have tablet after lunch`

13. Добавьте текст `Sorry. I could not add a new task.` в виде ответа к цели. Этот ответ будет отправлен, только если агент не сможет добавить новую задачу с помощью веб-приложения. Вскоре мы включим вебхук для достижения этой цели. Сохраните цель, нажав кнопку **Save** (Сохранить).
14. Создайте новую цель с именем `list_all_tasks`, отображающую все задачи в списке дел пользователя. Добавьте следующие высказывания:
    - `what tasks do i have still`
    - `list all my tasks`
    - `get me the list of all tasks`

    Присвойте ответному действию имя `inform_all_tasks` и установите ответ по умолчанию с текстом `Sorry. I cannot get the list of tasks just now`. Мы также включим вебхук для достижения этой цели позже.
15. Теперь настроим веб-службу, чтобы вызвать функции `list_all_tasks` и `add_task`. Щелкните мышью по пункту **Fulfillment** (Выполнение) в меню слева.

16. Задействуйте вебхук и введите URL-адрес веб-приложения для вызова. Поскольку у нас его еще нет, укажите заполнитель для него (**https://my-homebot.herokuapp.com/home**).

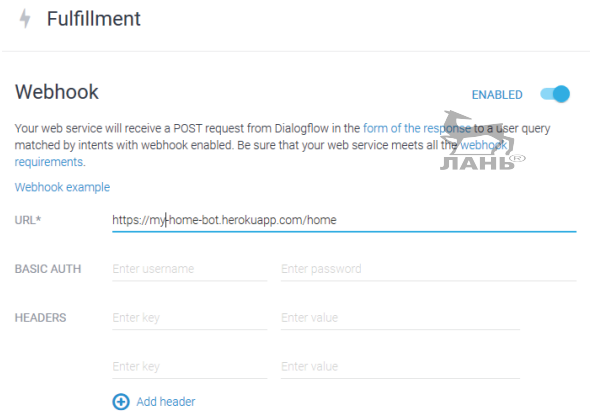


Рис. 9.13

17. Прокрутите страницу до конца и нажмите кнопку **Save** (Сохранить).
18. Вернитесь к объектам `list_all_tasks` и add_task. Для каждой из этих целей прокрутите страницу вниз и щелкните мышью по ссылке **Fulfillment** (Выполнение). Так будет гарантия, что цель и параметры передаются в веб-приложение, которое генерирует соответствующие ответные высказывания.

Создание веб-приложения

Теперь, когда у нас есть агент Dialogflow для распознавания пользовательских высказываний, нам нужно создать веб-приложение, с которым он может взаимодействовать. Агент передаст пользовательские цели и параметры в веб-приложение, чтобы можно было получить соответствующие ответы. Например, если пользователь хочет добавить новую задачу, в веб-приложение нужно отправить описание задачи, которое, в свою очередь, добавит новую задачу в ваш список дел с помощью API-интерфейса Todoist. Чтобы создать веб-приложение, выполните следующие действия:

1. Создайте новый проект Node.js, выполнив команду `npm`, и создайте проект бота-слуги:

```
> npm init
```

2. Установите необходимые библиотеки:

```
> npm install request --save
> npm install sync-request --save
> npm install express --save
> npm install body-parser --save
> npm install moment --save
> npm install node-uuid -save
```

`request` и `sync-request` – это библиотеки, которые будут использоваться для взаимодействия со службой Todoist. Библиотеки `express` и `body-parser` будут использоваться для разворачивания веб-сервера, `moment` – для вычисления времени и даты. Наконец, библиотека `node-uuid` будет использоваться для генерации UUID задач.

3. Создайте файл с именем *index.js*. Добавьте инструкции импорта, как показано ниже:

```
'use strict'

/*
* Бот Google Home
*/

const request = require('request')
const syncrequest = require('sync-request')
const express = require('express')
const bodyParser = require('body-parser')
const moment = require('moment');
const uuid = require('node-uuid');
```

4. Чтобы задействовать сервер, нам нужно добавить следующий код:

```
const app = express()

app.set('port', (process.env.PORT || 5000))

// Процесс application/x-www-form-urlencoded
app.use(bodyParser.urlencoded({extended: false}))
```

```
// Процесс application/json
app.use(bodyParser.json())

app.use(express.static('public'))

// Маршрут
app.get('/', function (req, res) {
    res.send('Hello world, I am your man Friday.')
})

// Работа с целями
app.post('/home', function (req, res) {
    // Обработка кода целей
}

// Включение сервера
app.listen(app.get('port'), function() {
    console.log('running on port', app.get('port'))
})
```

5. Добавьте следующий код в POST-обработчик /home для обработки целей:

```
console.log('Request from DialogFlow:');
console.log(req.body);
var intent = req.body.result.metadata.intentName;
var botSpeech = 'hello world';
if (intent === 'inform_task_description'){
    var taskDescription = req.body.result.parameters.task;
    addTask(taskDescription);
    botSpeech = 'Adding new task.';
}
else if (intent === 'list_all_tasks'){
    botSpeech = getAllTasks();
}
var out = {speech: botSpeech,
        displayText: botSpeech,
        data: null};
var outString = JSON.stringify(out);
console.log('Out:' + outString);
res.send(outString);
```

6. Наконец, нам нужно создать две функции, вызываемые из обработчика /home. Чтобы добавить новую задачу, сначала укажем следующий код:

```
function addTask(taskDescription){
    var todoist_token = 'YOUR-TODOIST-TOKEN';
    var task = {'type': 'item_add',
        'temp_id': uuid.v4(),
        'uuid': uuid.v4(),
        'args': {
            'content': taskDescription
    }};
    var url = 'https://todoist.com/api/v7/sync?token='
               + todoist_token +
                '&sync_token=*&resource_types=["items"]&commands=['
                + JSON.stringify(task) +']';
    console.log('URL:' + url);
    request({
        url: url,
        method: 'GET'
    },
    function (error, response, body) {
        // Ответ бота
        if (!error && response.statusCode == 200) {
            console.log(JSON.parse(body));
        } else {
            console.log('Error: ' + error)
            console.log('Statuscode: ' + response.statusCode)
        }
    });
}
```

А теперь, чтобы получить сводку всех задач, добавим следующий код:

```
function getAllTasks(){
    var todoist_token = 'YOUR-TODOIST-TOKEN';
    var url = 'https://todoist.com/api/v7/sync?token='
              + todoist_token +
                     '&sync_token=*' +
                     '&resource_types=["items"]';
        console.log('URL:' + url);
        var resp = syncrequest('GET', url);
        var allItems = JSON.parse(resp.getBody('utf8')).items;
        var allItemsSummary = '';
        for (var i = 0; i < allItems.length; i++){
            allItemsSummary += allItems[i].content;
            if (i < allItems.length - 2){
```

```
                allItemsSummary += ‘, ‘;
            }
            else if (i == allItems.length - 2) {
                allItemsSummary += ‘ and ‘;
            }
            else {
                allItemsSummary += ‘.’;
            }
        }
        var alltasks = ‘You have ‘ + allItems.length + ‘
                       in your list. ‘
                        + allItemsSummary;
        return alltasks;
}
```

7. Разместим код в облаке, выполнив следующие команды:

```
> git init
> git add .
> git commit -m initial-commit
> heroku create my-home-bot
> git push heroku master
```

После размещения веб-приложение готово к общению с нашим агентом Dialogflow.

Тестирование агента на Dialogflow

Теперь мы готовы протестировать нашего верного слугу на платформе Dialogflow. Вы можете протестировать своего агента с помощью эмулятора. Если в одном из предыдущих разделов вы уже создавали агента Dialogflow (ранее API.AI), то знаете, как это сделать. Если нет, выполните следующие действия.

1. Перейдите на сайт **dialogflow.com**.
2. Справа вы увидите фрейм с текстовым полем **Try it now** (Попробовать сейчас), введите приветствие (`hi`) и нажмите клавишу **Enter**. Здесь мы будем вводить данные для передачи агенту. Агент возвращает текстовый ответ, а также цель пользователя и действие системы – рис. 9.14.
3. Теперь добавьте задачу (`add a new task`). Агент запросит описание задачи. Обратите внимание, что для заданной задачи (`asked-for-task`) обновлен контекст, – рис. 9.15.

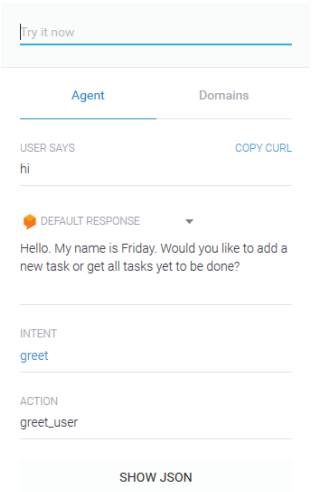


Рис. 9.14

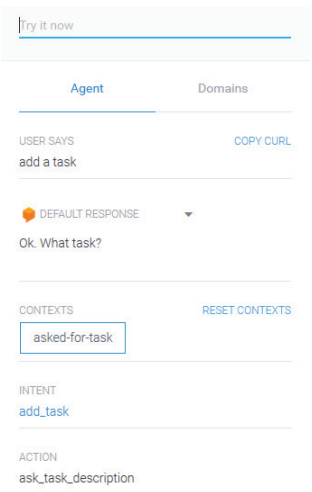


Рис. 9.15

4. Укажите описание задачи. Агент добавит новую задачу в список дел и отправит подтверждение – рис. 9.16.

   Если по какой-либо причине агент не может связаться с сервисом Todoist, он вернет ответное сообщение по умолчанию.

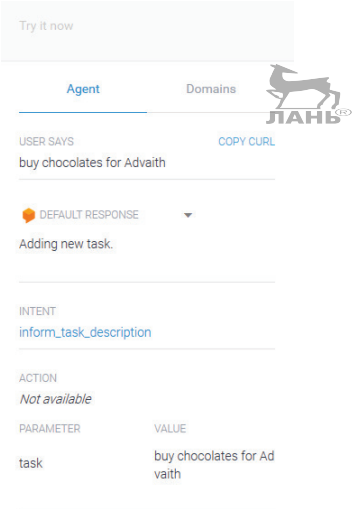


Рис. 9.16

5. Добавьте еще несколько задач подобным образом.
6. Теперь мы можем проверить задачи, добавленные в список дел, авторизовавшись на сайте **todoist.com** или используя приложение Todoist.

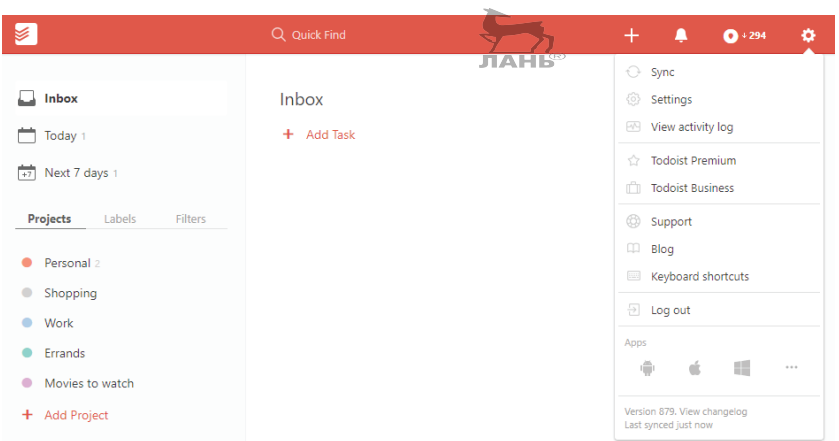


Рис. 9.17

7. Попробуйте вывести все задачи (`get all tasks`).

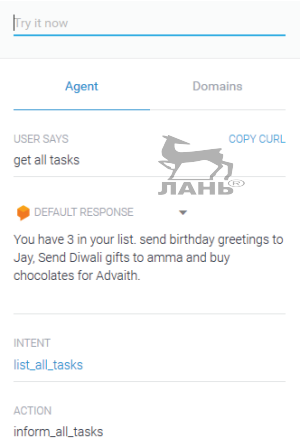


Рис. 9.18

Ура! Вы успешно интегрировали в агента Dialogflow службу Todoist.

Интеграция в Ассистента Google

Теперь, когда агент готов, следующим шагом будет интеграция в Ассистента Google. Выполните следующие шаги для выполнения этой интеграции.

> **Совет.** Чтобы протестировать проект в Интернете без аппаратного устройства, вам может потребоваться установить в активное положение переключатели **Web & App Activity** (История приложений и веб-поиска), **Device Information** (Информация с устройств) и **Voice & Audio Activity** (История голосового управления) на странице **Activity Controls** (Отслеживание действий) (**myaccount.google.com/activitycontrols**) вашей учетной записи Google.

1. В меню на сайте Dialogflow выберите пункт **Integrations** (Интеграции) – рис. 9.19.
2. Щелкните мышью по ссылке **Google Assistant** (Ассистент Google) – рис. 9.20.
3. В окне **Google Assistant** (Ассистент Google) установите переключатель в верхнем правом углу в активное положение.

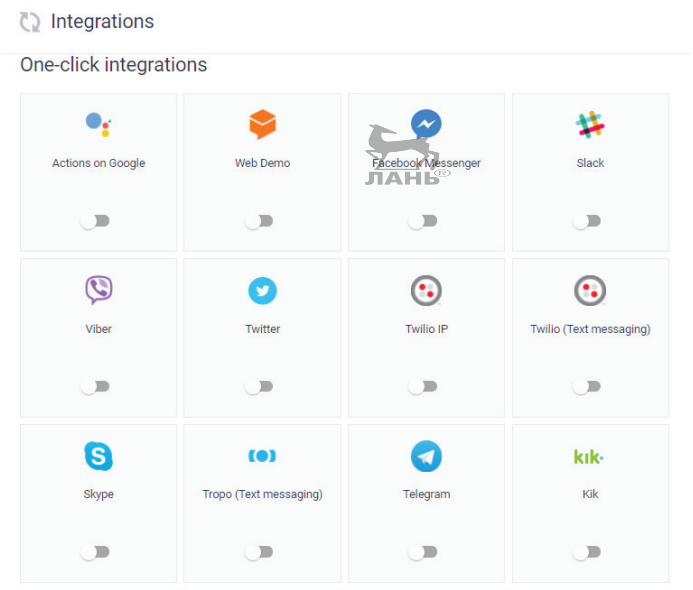


Рис. 9.19

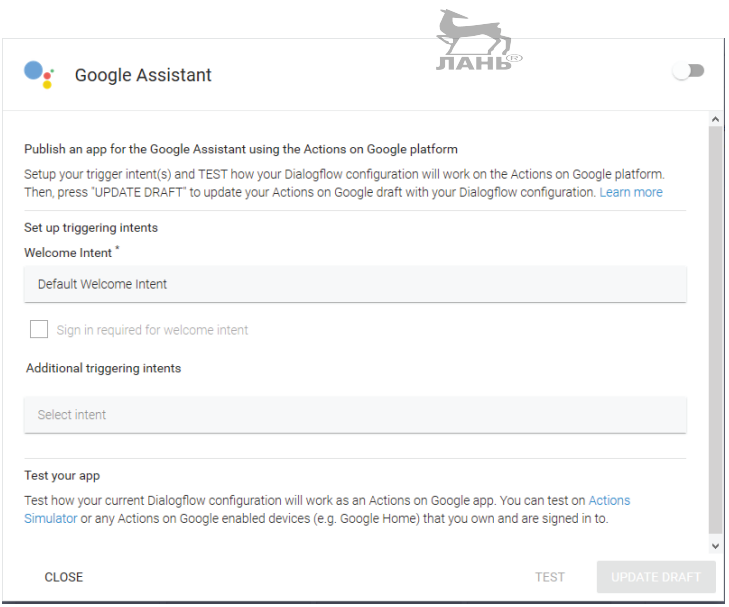


Рис. 9.20

4. Выберите цель `welcome`. Выберите цель `greet`, которую мы определили.
5. Нажмите кнопку **Test** (Тестировать), а затем **View** (Просмотреть), чтобы открыть страницу **Actions Simulator** (Эмулятор действий), – рис. 9.21.

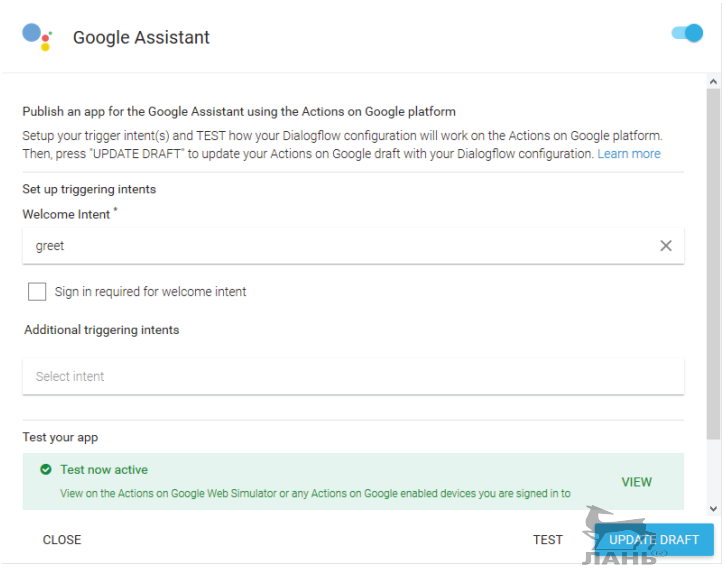


Рис. 9.21

На другой вкладке браузера откроется страница **console.actions.google.com**, где агента можно протестировать в режиме эмуляции действия для Ассистента Google, – рис. 9.22.

6. В разделе **Surface** (Устройство) щелкните мышью по значку смартфона и смените тип ввода на клавиатуру (**Keyboard**).
7. Введите текст `Talk to my test app` и нажмите клавишу **Enter**. Откроется тестовая версия действия, и вы сможете поговорить с агентом так же, как и на сайте Dialogflow. Все задачи будут добавлены в ваш список дел, который можно проверить, посетив сайт **todoist.com** или запросив список после добавления задач, – рис. 9.23.

Примечание. Обратите внимание, что текст `talk to my test app` является фразой инициации диалога по умолчанию. Мы определим фразы инициации диалога через минуту.

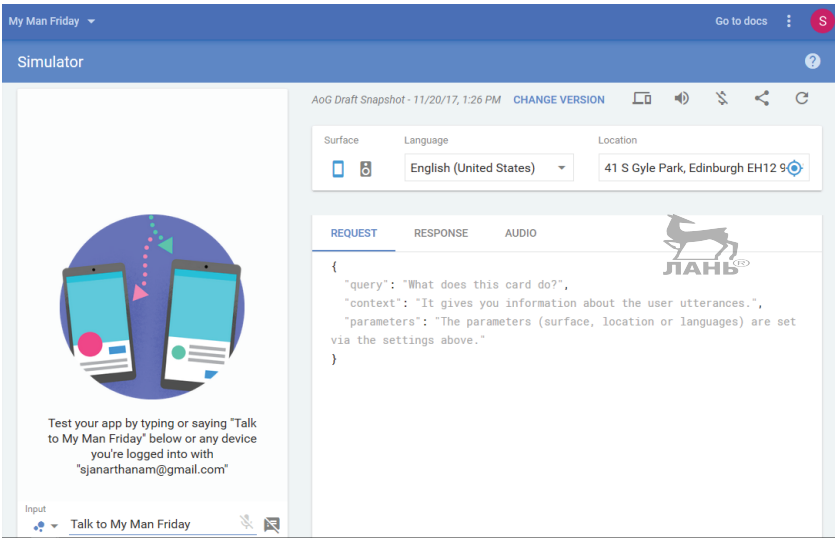


Рис. 9.22

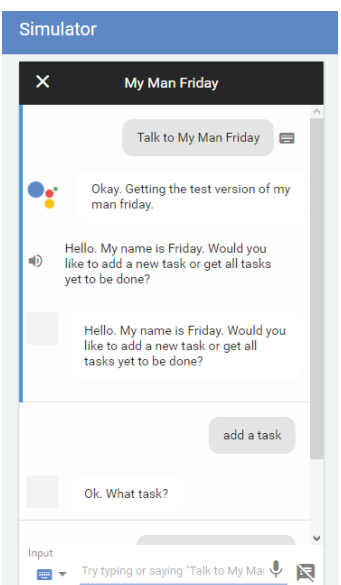


Рис. 9.23

8. После проверки агента вернитесь на сайт Dialogflow, щелкните мышью по ссылке **Update Draft** (Обновить черновик) и нажмите кнопку **Visit Console** (Открыть консоль) в открывшемся окне – рис. 9.24.

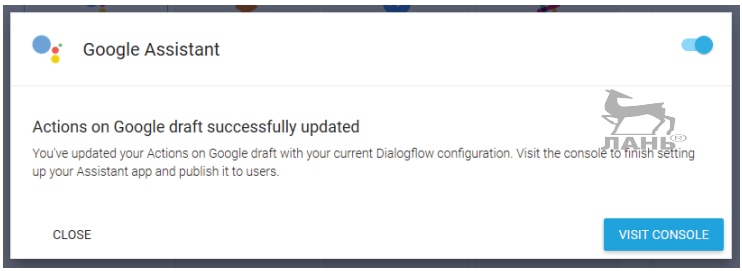


Рис. 9.24

9. Обратите внимание, что на странице **Overview** (Обзор) действия отсылаются к платформе Dialogflow. На следующем шаге нужно предоставить некоторую важную информацию о приложении.
10. Щелкните мышью по ссылке **Edit** (Изменить) – рис. 9.25.

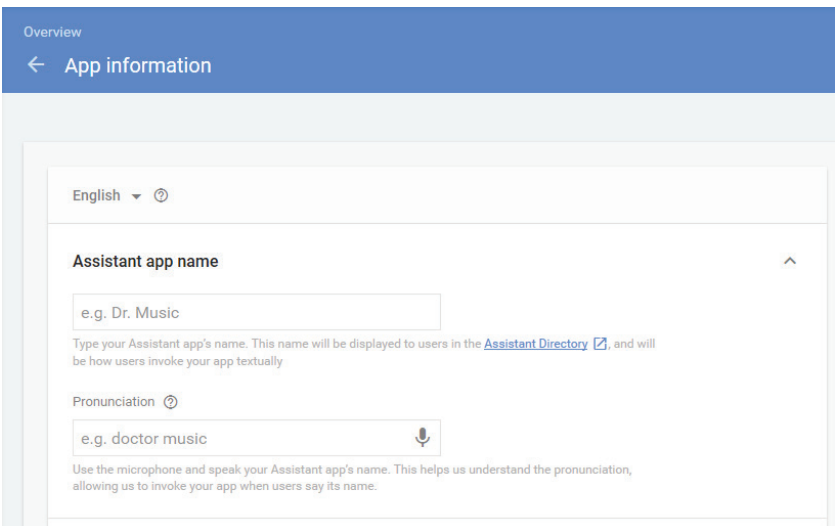


Рис. 9.25

11. Введите название, произношение и описание.

12. Прокрутите страницу вниз, чтобы просмотреть дефолтные вызовы для действия. При необходимости вы можете добавить собственные – рис. 9.26:

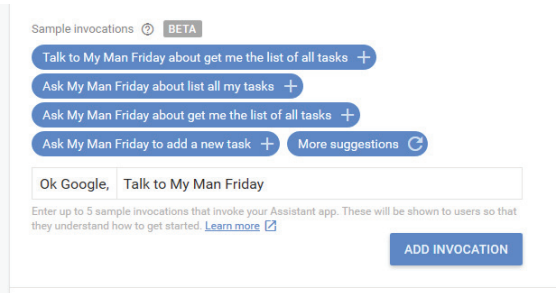


Рис. 9.26

13. Добавьте изображения для значка и баннера. На стадии разработки этот шаг можно пропустить.
14. Прокрутите страницу вниз и нажмите кнопку **Save** (Сохранить).

Теперь действие доступно для тестирования в учетной записи Google разработчика. Чтобы оно стало доступным для других пользователей, вы должны подать заявку на утверждение на сайте **developers.google.com/actions/distribute/**.

Проверка на устройстве Google Home

Теперь, когда действие настроено, вы можете протестировать его на своем устройстве Google Home. Необходимо использовать тот же аккаунт Google, что применялся для разработки агента и действия, поскольку действие все еще имеет статус «в разработке» и не опубликовано. Поэтому включите динамик Google Home, скажите: «Эй, Google» – и поговорите со своим верным слугой – рис. 9.27.

Теперь у вас должен получиться такой же диалог, что и в эмуляторе, только голосом и с помощью динамика Google Home. Хотя точность распознавания речи довольно высока, когда вы говорите простым английским текстом, система может не распознавать необычные имена и нетипичные конструкции. Диалог начнется с сообщения об отказе от ответственности, в котором говорится, что действие все еще тестируется. Попробуйте добавить задание и вывести

Рис. 9.27

список всех задач, которые вы планируете выполнить. Перейдите в приложение Todoist, чтобы удалить завершенные задачи, и вновь попробуйте просмотреть их на устройстве. Вы закончили разработку собственного действия для Ассистента Google!

Заключение

Прекрасно! Проделана замечательная работа по созданию действия для Ассистента Google, для управления списком планируемых дел. Я надеюсь, что вы сможете расширить функционал агента для организации задач по проектам, а также научитесь создавать и извлекать задачи в указанное время. Такие дополнения сделают агента намного более мощным. Представьте, что вы сможете управлять своими ежедневными задачами с помощью голосового ассистента дома или на работе. В отчете Global Market Insights за июнь 2017 года сообщается, что среднегодовой темп распространения ассистентов с голосовым управлением в умных динамиках предположительно составит 50% в период с 2017 по 2024 год. Эти ассистенты будут взаимодействовать не только с облачными услугами, но и с устройствами интернета вещей и прочими интеллектуальными домашними устройствами, такими как светильники, термостаты и узлы, контролируемые голосом. Наряду с компаниями Amazon, Google и Apple другие разработчики также стремятся отхватить свой кусок пирога. Ожидается, что в ближайшие годы количество действий Google и навыков Alexa будет увеличиваться для удовлетворения растущего спроса на рынке. Надеюсь, что, прочитав эту главу, вы начали свой путь к созданию интересных и инновационных возможностей для ассистентов с голосовым управлением.

Выводы

Хочется верить, что вам понравились все эти проекты и вы получили удовольствие, создавая захватывающих и интересных чат-ботов и голосовых ботов. Хорошо поработали! Но помните, мы едва коснулись этой темы, и много чего еще предстоит изучить. Доступно колоссальное количество возможностей по разработке чат-ботов и пользовательских интерфейсов, помимо простого осваивания инструментов, служб и источников данных. Необходимы дополнительная практика и более глубокое погружение в концепции отрасли, чтобы научиться разрабатывать ботов, предоставляющих пользователям бóльшие возможности. Пользователь должен достигать цели, поговорив с ботом, не задумываясь над построением фраз и улыбаясь. Отличный опыт в работе с диалоговыми интерфейсами – это тот, который пользователь получает при взаимодействии, к чему мы всегда должны стремиться. И только с практикой и тщательно продуманным дизайном мы можем достичь успеха.

Помните, что эти технологии развиваются быстрыми темпами, как и инструменты, услуги и знания базовых концепций. Чтобы быть в курсе последних событий, я бы рекомендовал вам подписаться на группы в социальных сетях (Facebook и LinkedIn), посвященные чат-ботам и ИИ, участвовать в локальных встречах программистов, подписываться на журналы и статьи и участвовать в хакатонах и конференциях (см. приложение). Многие компании по всему миру вводят чат-ботов как новые и удобные каналы для связи с общественностью. Следите за новостями и, по возможности, взаимодействуйте с чат-ботами, чтобы прочувствовать и понять принципы их проектирования.

Я верю, что эта книга предоставила вам фундаментальное понимание концепций, инструментов и техник, которые вы сможете использовать в будущем. Надеюсь, вам понравилось разрабатывать всех этих ботов, попутно вдохновляясь новыми идеями, чтобы создать крутые проекты в будущем. И напоследок я желаю вам успешных дней обучения, проектирования и разработки отличных диалоговых интерфейсов.

Ссылки

- Документация Google Actions: **developers.google.com/actions/extending-the-assistant**.

Приложение. Дополнительные ресурсы

Статьи

Представляю несколько статей, которые могут оказаться интересными и полезными.

- Срини Дванарсанам. Создание структуры чат-бота, **chatbotslife.com/chatbot-design-canvas-c3940685ca2c**.
- Йогеш Муржани. Конструирование чат-ботов, **uxdesign.cc/how-to-design-a-robust-chatbot-interaction-8bb6dfae34fb**.
- Ольга Давыдова. 25 платформ чат-ботов: сравнительная таблица, **chatbotsjournal.com/25-chatbot-platforms-a-comparative-table-aeefc932eaff**.
- Себыстьен Форолт. Исключительное руководство по разработке чат-бота, **chatbotsmagazine.com/the-ultimate-guide-to-designing-a-chatbot-tech-stack-333eceb431da**.
- Юнджи Сео. 19 лучших практик по построению чат-ботов, **chatbotsmag azine.com/19-best-practices-for-building-chatbots-3c46274501b2**.
- Рон Левинзон. Инструменты, которые должен знать каждый создатель ботов, **chatbotsmagazine.com/the-tools-every-bot-creator-must-know-c0e9dd685094**.

Список конференций

Ниже представлен список популярных конференций, на которых вы можете узнать больше о состоянии отрасли, а также сообществе дизайнеров, разработчиков и инвесторов.

- Chatbot Summit (**chatbotsummit.com**).
- ChatbotConf (**orat.io/chatbotconf**).
- Re-work AI Assistant Summit (**www.re-work.co/events/**).
- Intelligent Assistants Conferences (**opusresearch.net/wordpress/events/**).

Список журналов

Представляю список журналов и информационных бюллетеней, в которых публикуются подробные статьи и новости о событиях из мира чат-ботов:

- Chatbots Magazine (**chatbotsmagazine.com**).
- Chatbots Journal (**chatbotsjournal.com**).
- Chatbot's Life (**chatbotslife.com**).
- Chatbot News Daily (**chatbotnewsdaily.com**).
- Chatbots Weekly (**www.chatbotsweekly.com**).

Группы в социальных сетях

Ниже приведен список групп в социальных сетях, на которые можно подписаться, чтобы общаться с коллегами и быть в курсе последних событий в отрасли:

- UX for Bots (**www.facebook.com/groups/uxforbots/**).
- ChatBots (**www.facebook.com/groups/aichatbots/**).
- Chatbots Developers (Global) (**www.facebook.com/groups/chatbotsdevelopers/**).
- Messenger Platform Developer Community (www.facebook.com/groups/messengerplatform/).
- Chatbot Professionals (**www.linkedin.com/groups/7052578**).

Предметный указатель

А

Автоматическое распознавание речи, 18
Архитектура диалогового пользовательского интерфейса, *23*
Атрибуты, 40, 41

Б

Блоки, 40
  по умолчанию, 41, 57
Бот
  Запрос информации у пользователей, 201
  Отправка более одного сообщения одновременно, 200
  Развернутые сообщения, 204
  Развертывание, 198
  Реакция на пользовательские высказывания, 213
  Создание, 195
  цитат, 276
    Менеджер диалогов, 285, 287, 289
    Настройка навыков, 277
Бот-персонаж, 116

Г

Графические пользовательские интерфейсы, 19

Д

Джозеф Вайзенбаум, 20
Диалоговые пользовательские интерфейсы, 19, 21, 23, 25, 27
Дополнительные карты, 45, 47
  Аудио, 46
  Быстрые ответы, 47
  Видео, 47
  Галерея, 48
  Графика, 45
  Кнопки, 50
  Навигация, 49, 51
  Списки, 48

З

Завершение общения, 219

И

Импорт и экспорт агентов, 143
Интерфейсы естественного языка для баз данных, 21

К

Каналы, 28
Карты, 40, 41
Классификация диалоговых интерфейсов, 25
Кнопки, 40, 41
Конструирование диалога, 71
Контекст, 127
Контекстные переменные, 56

М

Модель взаимодействия, 278, 279, 281, 283
построение, 292, 293, 295, 297, 299, 301
Монетизация, 31

Н

Настраиваемые пользовательские атрибуты, 55
Настройка бота, 42, 43
Новостной бот, 239

О

Обработка на стороне сервера, 60, 61, 63, 65
Обработка произношения пользователей, 181
Осуществления, 149

П

Переключение контекста, 217
Переход к блоку, 51
Плагины, 40, 41
Погодный бот, 71
Подключение к Skype, 233, 235, 237
Получение данных о погоде, 72
Получение ключа API, 72
Пользовательский ввод, 53
Построение интерфейса на стороне сервера, 76
Поток диалога, 209
Проверка вашего ключа, 74
Публикаци
в Facebook Messenger, 93, 103
Публикация
в Facebook Messenger, 93, 95, 97, 99, 101, 105, 107
для общего доступа, 106
Путеводитель по городу, 35

Р

Развернутые ответы, 141
Развертывание чат-бота
Интеграция в Facebook Messenger, 123, 125, 127, 129, 131, 133, 135, 137, 139, 141, 143, 145, 147, 149, 151, 153, 155
Интеграция в веб-сайт, 121
Разговорные системы, 21
Распознавани
естественного языка, 30, 57
Распознавание
естественного языка, 18, 59
контекста, 219
Реализация чат-бота, 81, 83, 85, 87

С

Синтез текста в речь, 18
Слоты
встроенные, 305
Службы каталогов, 30
Создание
веб-приложения, 318, 319, 321
веб-службы чат-бота, 88, 89, 91
Состояние диалога, 221
Сохранение целостности контекста, 216
Структура диалога, 163
Сущности, 131
Системные, 139
Экспорт и импорт, 146

Т

Телегид, 291
Настройка сервера, 296
Тестирование навыка, 303
Тестирование чат-бота, 105
Добавление тестировщиков, 106
Типы содержимого
Вложения видео, 109
Вложения изображений, 108

Документы, 109
Звуковые вложения, 108

У

Учет контекста, 52, 53, 55

Ц

Цели
встроенные, 305
Экспорт и импорт, 146

Ч

Чат-бот
история, 20

Ш

Шаблоны
кнопки, 110
списка, 111
Универсальный, 112
Широковещание, 66

Э

Эмулятор канала, 195

Я

Язык разметки искусственного интеллекта, 20

A

AIML, 20, 29
Alexa, 22
Amazon Alexa, 275
Artificial Intelligence Markup Language, 20
ASR, 18

B

BAT-файл. См. Пакетный файл
BillyBot, 27

C

Chatfuel, 29, 35, 68
регистрация, 36
создание бота, 37
ChatScript, 20, 29
Cleo, 26
CMD-файл. См. Пакетный файл
ConcludeBot, 27
Cortana, 22

D

Dialogflow, 29, 116, 117, 119, 121
интерфейс Node.js, 184
Настройка, 117
Развертывание чат-бота, 121
Создание агента, 181, 313, 315, 317
Создание базового агента, 118
Тестирование агента, 322, 323
DocuBot, 27
DoNotPay, 26

E

Echo
тестирование бота, 291
Eliza, 20
Emily, 27
Ernest.ai, 26
Eugene Goostman, 20

F

Facebook
Messenger, 22, 70
Дополнительные форматы сообщений, 108, 109, 111, 113
Связывание приложений, 104

Создание
имени пользователя, 95
интерфейса для веб-приложения, 100
приложения, 96
страницы, 94

G

Google Assistant, 23
GUI, 19

H

HealthTap, 27
H&M, 26
Homepod, 23

L

LawBot, 27
LawDroid, 27
LegaliBot, 27
Lexi, 27
Lisa, 27
LunchTrain, 27

M

ManyChat, 29
Meekan, 27
MS Bot, 195

N

Natural Language Interfaces to Database Systems, 21
NewsAPI, 248
NLIDBS, 21
NLU, 18

O

Obie.ai, 27

P

PandoraBots, 29

R

RiveScript, 29

S

SDK Twitter, 246, 247
Обновление статуса, 246
Поиск твитов, 247
Ретвиты сообщений, 247
SDS, 21
SimplePoll, 27
Siri, 21
SMS-бот, 165, 167, 169, 171, 173
Интеграция агента Dialogflow, 187
Настройка панели управления, 166
Отправка простых сообщений, 167
Планирование задач, 173
Создание двухстороннего чат-бота, 175
Уведомления о поездах, 170
SPIXII, 26
Spoken Dialogue Systems, 21
Stoptober, 27

T

Todoist, 309
Добавление задачи, 311
Получение всех задач на день, 312
Получение ключа, 309
Создание действия, 313
Transport API, 157, 159, 161
Ближайшая станция, 163
Время прибытия, 164
Остановки рядом с вами, 159
Отправления в реальном времени, 160

Поезда в этом районе, 160
Расписание конкретного поезда, 162
Расписание на конкретной станции, 160
Следующий поезд, 164
Создание учетной записи разработчика, 158
TTS, 18
Twitter, 240, 241
Twitter-бот, 242, 243, 245, 252, 253
Доступ к базе интересов пользователей, 264
Информирование бота об интересах пользователей, 267
Персонализация, 255, 257, 259, 261, 263, 265, 267, 269, 271
Создание базы данных интересов пользователей, 259

W

Watson, 22
WorkBot, 27

Y

Your.MD, 27

Z

Zomato, 222, 223
Получение данных, 223
Служба для поиска ресторанов, 225, 227, 229, 231

Книги издательства «ДМК Пресс» можно заказать в торгово-издательском холдинге «Планета Альянс» наложенным платежом, выслав открытку или письмо по почтовому адресу: 115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.

При оформлении заказа следует указать адрес (полностью), по которому должны быть высланы книги; фамилию, имя и отчество получателя. Желательно также указать свой телефон и электронный адрес.

Эти книги вы можете заказать и в интернет-магазине: **www.alians-kniga.ru**.

Оптовые закупки: тел. (499)782-38-89.

Электронный адрес **books@alians-kniga.ru**.

Срини Джанарсанам

Практическое руководство по разработке чат-интерфейсова

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод с англ. *Райтман М.А.*
Корректор *Синяева Г. И.*
Верстка *Татаринов А. Ю.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16.
Гарнитура «Петербург». Печать офсетная.
Усл. печ. л. 21,25. Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com