**3 РАЗРАБОТКА ПРОГРАММНЫХ СРЕДСТВ ДЛЯ РЕШЕНИЯ ЗАДАЧИ**

**3.1 Постановка задачи**

В процессе прохождения преддипломной практики бы ли поставлены следующие задачи:

* изучение требований, предъявляемых к безопасной и комфортной работе инженеров-программистов;
* знакомство со спецификой этапов жизненного цикла программного продукта;
* изучение принципов работы программ для имитации речевого поведения человека при общении с одним или многими собеседниками;
* изучение языка разметки искусственного интеллекта  *AIML* с целью анализа его возможностей для построения интеллектуального чат-бота;
* изучение существующих фреймворков (*API*) для построения чат-ботов на базе распространенных текстовых менеджеров  (*Telegram*, *Facebook*, *Skype* и др);
* проектирование структуры программного комплекса и базы данных для хранения информации в соответствии с темой дипломной работы;
* разработка программного модуля для запроса информации из базы данных заданной предметной области на основе текстового пользовательского запроса;
* формирование информационной базы для выполнения дипломной работы и тестирования предполагаемых к созданию программных продуктов;
* создание прототипа программного продукта в соответствии с темой дипломной работы.
  1. **Особенности разработки приложения**

Общий ресурс, или общий сетевой ресурс – в информатике, это устройство или часть информации, к которой может быть осуществлён удалённый доступ с другого компьютера, обычно через локальную компьютерную сеть или посредством корпоративного интернета, как если бы ресурс находился на локальной машине. Примерами такого могут служить общий доступ к файлам (также известный как общий доступ к диску и общий доступ к папкам), общий доступ к принтеру (совместный доступ к принтеру), сканеру и т. п.

Веб-приложения создаются в различных вариантах архитектуры клиент-сервер. В таком случае клиентом выступает браузер, а сервером – веб-сервер.

Логика веб-приложения распределена между сервером и клиентом, хранение и обработка данных осуществляется, преимущественно, на сервере, её представление в удобном для пользователя виде – в браузере. Обмен информацией происходит по сети. Общая структура веб-приложения приведена на рисунке 3.1.

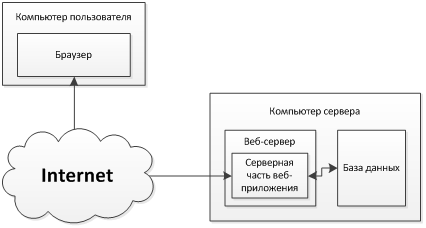


Рисунок 3.1 – Общая структура веб-приложения

Таким образом, чаще всего веб-приложения состоят как минимум из трёх основных компонентов.

Серверная часть веб-приложения – это программа или скрипт на сервере, обрабатывающая запросы пользователя (точнее, запросы браузера). Для программирования серверной части веб-приложения выбран язык *Java*. При каждом переходе пользователя по ссылке браузер отправляет запрос серверу. Сервер обрабатывает данный запрос, запуская некоторый скрипт, который формирует веб-страницу, описанную на языке *HTML*, и отсылает клиенту по сети. Браузер тут же отображает полученный в виде очередной веб-страницы результат [9].

Серверная часть веб-приложения обращается к базе, извлекая данные, которые необходимы для формирования страницы, запрошенной пользователем. В данном приложении база знаний представляет собой файлы формата *XML*.

*XML*-документ представляет собой обычный текстовый файл, в котором при помощи специальных маркеров создаются элементы данных, последовательность и вложенность которых определяет структуру документа и его содержание. Основным достоинством *XML* документов является то, что при относительно простом способе создания и обработки (обычный текст может редактироваться любым тестовым процессором и обрабатываться стандартными *XML* анализаторами), они позволяют создавать структурированную информацию, которую хорошо «понимают» компьютеры.

Пример формата исходного файла можно увидеть на рисунке 2.2.

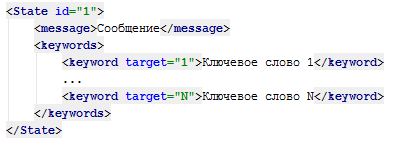


Рисунок 3.2 – Структура хранимых данных

Клиентская часть веб-приложения – это отображаемый в браузере графический интерфейс. Это то, что пользователь видит на странице. Через данный интерфейс пользователь взаимодействует с приложением, вводя информацию, нажимая на ссылки и кнопки.

Основной язык, которым описывается графический интерфейс веб-приложения – это *HTML*. Данный язык описывает структуру веб-страницы, размещение на ней компонентов. Оформление веб-страниц, их стиль и цветовая схема описываются в таблицах стилей – *CSS*. Для динамичности графического интерфейса, используются скрипты *JavaScript*. Программа, написанная на *JavaScript* и выполняющаяся на веб-странице, может управлять встроенными в страницу компонентами, тем самым реализуя пользовательский интерфейс с богатыми возможностями.

Отсутствие необходимости полностью перезагружать страницу после каждого получения данных от сервера может существенно ускорить работу веб-приложения. Такая концепция имеет название *Asynchronous* *JavaScript* *and* *XML* (асинхронный *JavaScript* и *XML*, *Ajax*). При использовании данного подхода динамические запросы к серверу происходят без видимой перезагрузки веб-страницы: пользователь не замечает, когда его браузер запрашивает данные.

**3.3 Описание разработанного приложения**

Программный продукт представляет собой веб-приложение, разработанное на языке *Java* в среде разработки *IntelliJ IDEA*. Данная среда разработки позволяет быстро, эффективно и просто создавать полноценное и многофункциональное приложение.

***3.3.1*** В качестве архитектурного фреймворка был выбран *Spring*. Этот фреймворк, используется для обеспечения лучшей масштабируемости приложения, возможности более простого тестирования и более простой интеграции с другими фреймворками. Благодаря этому писать большие приложения становится проще - разработчики просто избегают ряда проблем, связанных с созданием больших приложений, вместо того, чтобы их решать.

*Spring* – достаточно крупный фреймворк. Потому что создатели этого его ухитрились охватить практически все аспекты программирования промышленных *Java*-приложений. Соответственно, и составных частей у *Spring Framework* немало. На иллюстрации 3.3 отображена схема фреймворка.

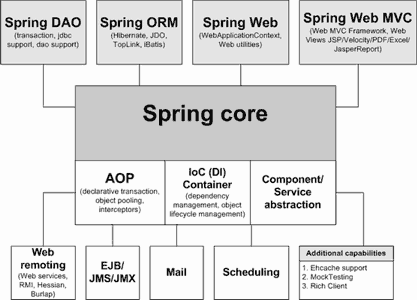


Рисунок 3.3 – Структура *Spring Framework*

*Inversion of Control* – на русский язык это название часто переводят как «инверсия управления». Довольно часто, говоря о *Spring Framework*, используют не этот термин, а «*Dependency Injection*».

Суть инверсии управления можно рассмотреть на небольшом примере. Например, у нас есть классы A и B. Причём класс A зависит от класса B (то есть, вызывает какие-то его методы). Для того, чтобы разорвать эту зависимость, вводят третье звено в этой цепочке - интерфейс Б. Он должен содержать те методы B, которые нужны классу A, и быть реализованным с помощью класса B. Это и называется инверсией управления. А *IoC*-контейнер – означает что все объекты хранятся в одном контейнере. Когда нам нужен экземпляр какого-то класса, то запрашивается контейнер, который извлекает или создаёт необходимый нам экземпляр и все его зависимости. Собственно говоря, ослабление связей между компонентами приложения и есть основная задача *Spring Framework*. Именно это обеспечивает хорошую масштабируемость приложений, сравнительную простоту их тестирования и другие особенности приложений на основе *Spring*.

*Spring MVC* основан на запросах. Разработчики работают со специальными «стратегическими» интерфейсами, однако у них есть доступ и к *Servlet API*, если вдруг они захотят спуститься на более низкий уровень абстракции. Далее перечисляются наиболее важные из этих интерфейсов. *HandlerMapping* ответственен за выбор обработчиков внешних запросов к приложению, *HandlerAdapter* – за вызов и исполнение этих обработчиков, *Controller* – за управление запросами и перенаправление на нужные ответы, *View* - за отсылку нужных ответов клиенту, *ViewResolver* – за выбор нужного в соответствии с ситуацией *View*, *HandlerInterceptor* – за перехват и фильтрацию приходящих запросов, *LocaleResolver* – за определение и сохранение локали пользователя, *MultipartResolver* – за содействие загрузке файлов.

***3.3.2*** Сам шаблон *MVC*, подразумевает взаимодействие трех компонентов: контроллера (*controller*), модели (*model*) и представления (*view*).

Контроллер (*controller*) представляет класс, с которого собственно и начинается работа приложения. Этот класс обеспечивает связь между моделью и представлением. Получая вводимые пользователем данные, контроллер исходя из внутренней логики при необходимости обращается к модели и генерирует соответствующее представление.

Представление (*view*) – это собственно визуальная часть или пользовательский интерфейс приложения - например, *html*-страница, через которую пользователь, зашедший на сайт, взаимодействует с веб-приложением.

Модель (*model*) представляет набор классов, описывающих логику используемых данных. [10].

Общую схему взаимодействия упрощенно представлена на рисунке 3.4.

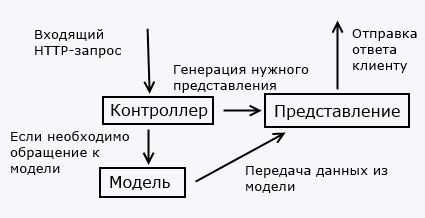


Рисунок 3.4 – Структура паттерна *MVC*

***3.3.3*** В разрабатываемом приложение роль представления выполняют *jsp* страницы, а так же интеграция с одним из самых прогрессивный и функциональный мессенджер на сегодняшний день –*Telegram*.

Технология проектирования *Java Server Pages (JSP)* – это одна из технологий *J2EE*, которая представляет собой расширение технологии сервлетов для упрощения работы с *Web*-содержимым. Страницы *JSP* позволяет легко разделить *Web*-содержимое на статическую и динамическую часть, допускающую многократное использование ранее определенных компонентов. Разработчики *Java Server Pages* могут использовать компоненты *JavaBeans* и создавать собственные библиотеки нестандартных тегов, которые инкапсулируют сложные динамические функциональные средства.

Спецификация *Java Server Pages* наследует и расширяет спецификацию сервлетов. Как и сервлеты, компоненты *JSP* относятся к компонентам *Web* и располагаются в *Web*-контейнере. Страницы *JSP* не зависят от конкретной реализации *Web*-контейнера, что обеспечивает возможность их повторного использования.

***3.3.4*** *Telegram* – бесплатный кроссплатформенный мессенджер для смартфонов и других устройств, позволяющий обмениваться текстовыми сообщениями и медиафайлами различных форматов. Используются проприетарная серверная часть c закрытым кодом и несколько клиентов с открытым исходным кодом, в том числе под *GNU GPL*.

Учётные записи пользователей привязываются к телефонным номерам. При регистрации в сервисе и последующих авторизациях новых устройств, производится проверка телефонного номера через отправку *SMS*-сообщения с или телефонный вызов.

При помощи специального *API* сторонние разработчики могут создавать «ботов», специальные аккаунты, управляемые программами. Типичные боты отвечают на специальные команды в персональных и групповых чатах, также они могут осуществлять поиск в интернете или выполнять иные задачи, применяются в развлекательных целях или в бизнесе.

Для интеграции бота в Telegram, было создано отдельное приложение использующее библиотеку *TelegramBots*. Это простая в использовании библиотека для создания ботов в *Java* для *Telegram*, поддерживающая встроенный режим.

***3.3.5*** Для проверки и исправления приходящих от пользователя сообщений используется сервис проверки правописания Яндекс.Спеллер (коротко Спеллер). Он предлагает веб-разработчикам использовать возможность интерактивной проверки орфографии на страницах своих сайтов.

Спеллер позволяет выявлять и исправлять орфографические ошибки в текстах, набираемых посетителями сайтов в различных *HTML*-формах.

Спеллер анализирует слова, основываясь на правилах орфографии и лексике современного языка. В качестве словарного источника используется орфографический словарь, содержащий правильные написания большинства наиболее употребимых слов.

Сервис в настоящее время поддерживает три языка:

1. русский – словарь содержит 3.6 миллиона словоформ (разработка компании «Информатик»);
2. украинский – словарь содержит 1.8 миллиона словоформ (разработка компании «Информатик»);
3. английский – словарь содержит 150000 словоформ (разработка компании «Яндекс»).

Яндекс.Спеллер выявляет и исправляет только орфографические ошибки, то есть ошибки в отдельных словах, такие как:

* неправильные, пропущенные или лишние буквы, например, «карова», «велоспед», «караввай»;
* ошибки капитализации (неправильное употребление прописных и строчных букв), например, «мОСКВА»;
* повторы слов, например, «поезд поехал поехал быстрее».

Пунктуационные, грамматические (ошибки согласования слов) и стилистические ошибки не исправляются.

Яндекс.Спеллер является *HTTP*-службой и предоставляет *Web Service API* для обработки клиентских запросов на проверку орфографии. Яндекс.Спеллер поддерживает *HTTP* *GET*- и *POST*-запросы и работает в кодировке *UTF-8*.

Для доступа к Яндекс.Спеллеру по *HTTP* предлагаются *XML*-, *SOAP*-, *JSON*- и *JSONP*-интерфейсы. Все интерфейсы обеспечивают одинаковую функциональность и используют одни и те же входные параметры.

*XML*- и *SOAP*- интерфейсы возвращают ответ в виде *XML*-документа, *JSON*-интерфейс вместо *XML*-элементов возвращает *JavaScript*-объекты с теми же именами и семантикой, а *JSONP*-интерфейс возвращает те же самые *JavaScript*-объекты, но в виде вызова *callback*-функции с заданным именем.

Доступ к *API* предоставляется по следующим *URL*:

1. *XML*- и *SOAP*-интерфейс *http://speller.yandex.net/services/spellservice*;
2. *WSDL*-документ для *SOAP* доступен по адресу *http://speller.yandex.net/services/spellservice?WSDL*;
3. *JSON*- и *JSONP*-интерфейс *http://speller.yandex.net/services/spellservice.json*

Для включения *JSONP*-интерфейса требуется передать дополнительный параметр callback с именем функции обратного вызова.

*Web Service API* включает в себя два метода, которые позволяют проверять правописание в одном или нескольких фрагментах текстов: *checkText* и *checkTexts.*

***3.3.6*** Для автоматизации сборки проекта использовался фреймворк *Maven*. Это инструмент для сборки *Java* проекта: компиляции, создания *jar*(*war*), создания дистрибутива программы, генерации документации. Простые проекты можно собрать в командной строке. Если собирать большие проекты с командной строки, то команда для сборки будет очень длинной, поэтому её иногда записывают в *bat/sh* скрипт. Но такие скрипты зависят от платформы. Для того чтобы избавиться от этой зависимостии и упростить написание скрипта используют инструменты для сборки проекта.

Для платформы *Java* существуют два основных инструмента для сборки: *Ant* и *Maven*.

*Maven* не зависит от *OS*. Сборка проекта происходит в любой операционной системе. Файл проекта один и тот же. Он позволяет управлять зависимостями. Редко какие проекты пишутся без использования сторонних библиотек. Эти сторонние библиотеки зачастую тоже в свою очередь используют библиотеки разных версий. *Maven* позволяет управлять такими сложными зависимостями. Что позволяет разрешать конфликты версий и в случае необходимости легко переходить на новые версии библиотек.

Возможна сборка из командной строки. Такое часто необходимо для автоматической сборки проекта на сервере (*Continuous Integration*).

Имеет хорошая интеграцию со средами разработки. Основные среды разработки на *Java* легко открывают проекты которые собираются c помощью *Maven*. При этом зачастую проект настраивать не нужно – он сразу готов к дальнейшей разработке.

Как следствие – если с проектом работают в разных средах разработки, то *Maven* удобный способ хранения настроек. Настроечный файл среды разработки и для сборки один и тот же – меньше дублирования данных и соответственно ошибок.

*Maven* обеспечивает декларативную, а не императивную (в отличие от средства автоматизации сборки *Apache* *Ant*) сборку проекта. В файлах описания проекта содержится его спецификация, а не отдельные команды выполнения. Все задачи по обработке файлов, описанные в спецификации, *Maven* выполняет посредством их обработки последовательностью встроенных и внешних плагинов.

Информация для сборки проекта, поддерживаемого *Apache* *Maven*, содержится в *XML*-файле с названием *pom*.*xml*. При запуске *Maven* проверяет, содержит ли конфигурационный файл все необходимые данные и все ли данные синтаксически правильно записаны. Минимальная конфигурация включает версию конфигурационного файла, имя проекта, его автора и версию. С помощью *pom.xml* конфигурируются зависимости от других проектов, индивидуальные фазы процесса построения проекта (*build process*), список плагинов, реализующих порядок сборки.

## **3.4 Описание основной структуры приложения**

Приложение можно условно разделить на серверную и клиентскую части. Серверная часть представляет собой набор *java* классов. Структура этихклассов представлена на рисунке 3.5.

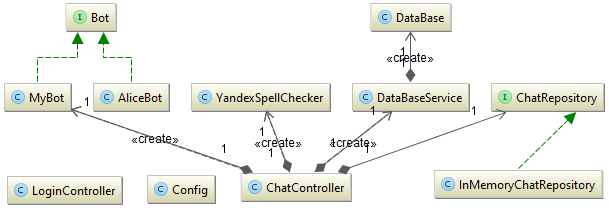


Рисунок 3.5 – *Java* классы приложения

Отображенные на рисунке выше классы, отвечают за работу приложения на сервере. Подробная структура основных классов изображена на рисунках 3.6 – 3.10. *Java* код классов находится в приложении А.

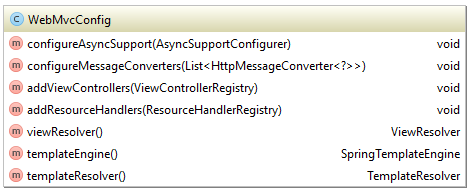


Рисунок 3.6 – Методы класса *WebMvcConfig*

Класс *WebMvcConfig* является единой точкой всех конфигураций.Отвечает за конфигурирование маппингов приложения и определение *spring* конфигов.

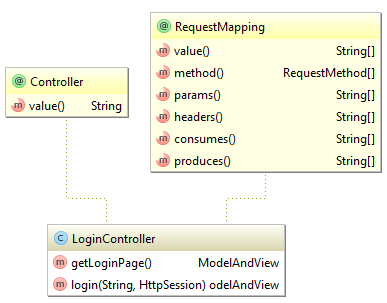


Рисунок 3.7 – Зависимости и методы класса *LoginController*

*LoginController* – это класс, который первым начинает работу с пользователем на стороне сервера. Отвечает за отображение логин страницы и аутентификацию, отправившего запрос, пользователя.

Аутентификация – проверка подлинности предъявленного пользователем идентификатора. Аутентификация требуется при доступе к таким интернет-сервисам как: электронная почта, форум, социальные сети и т.п. Положительным результатом аутентификации (кроме установления доверительных отношений и выработки сессионного ключа) является авторизация пользователя, то есть предоставление ему прав доступа к ресурсам, определенным для выполнения его задач.

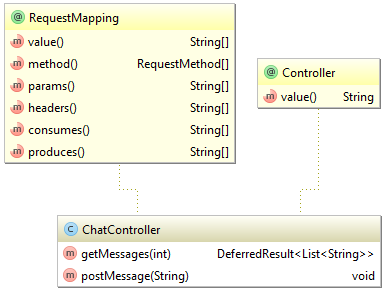


Рисунок 3.8 – Зависимости и методы класса *ChatController*

За основную работу с пользователем отвечает класс *ChatController.* Он представляет собой спринговый сервлет. И предоставляет методы получения запросов от пользователя и отправки ему ответов.

Метод *postMessage* получает вопрос пользователя. Для проверки и исправления ошибок в введенном пользователем сообщении, оно передается в метод *check* класса *YandexSpellChecker*. Затем корректное сообщение отправляется в класс *Bot* для получения ответа. Полученный ответ предается методу *fillResponse* класса *DataBaseService* для дополнения ответа информацией из базы данных. Вернувшаяся строка добавляется в ответ и ответ отправляет пользователю.

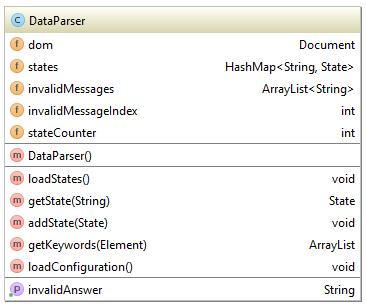


Рисунок 3.9 – Методы класса *Parser*

Класс *Parser* используется для работы с исходными файлами данных приложения. Содержит методы парсинга используемого формата файлов.

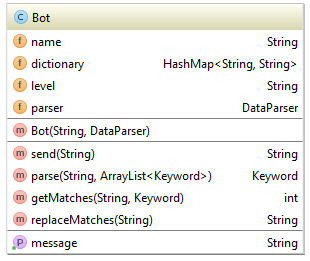


Рисунок 3.10 – Методы класса *Bot*

Класс *Bot* отвечает за внутреннюю работу с сообщениями. Содержит методы для определения ответа на текущее сообщение пользователя.

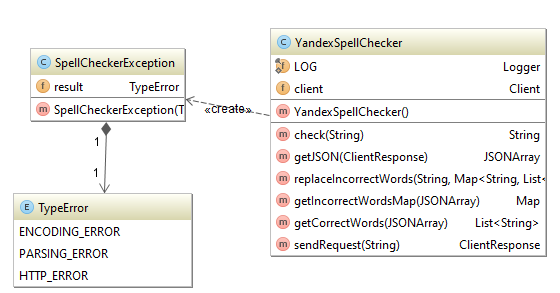


Рисунок 3.11 – Методы класса *YandexSpellerCheck*

Класс *YandexSpellerCheck* представляет собой интерфейс для работы с сервисом проверки правописания *Yandex* *Speller*.

Метод *sendRequest* получает строку и отвечает за формирование *http* cсоединения и отправку запроса к серверу *Yandex* *Speller*. И возвращает полученный *response*.

Метод *getJSON* входным параметром получает *response*. Выполняет парсинг ответа в объект типа *JSONArray*.

Метод *getIncorrectWordsMap* из объекта типа *JSONArray* формирует объект *Map*, в котором ключом являются слова, в которых допущена ошибка, а значением набор правильных вариантов написания этого слова.

Метод *replaceIncorrectWords* заменяет неправильные слова в исходном сообщении их корректными версиями на основе переданного объекта *Map*.

Метод *check* в качестве входного параметра получает сообщение пользователя, вызовом методов описанных выше, проверяет это сообщение и исправляет ошибки.

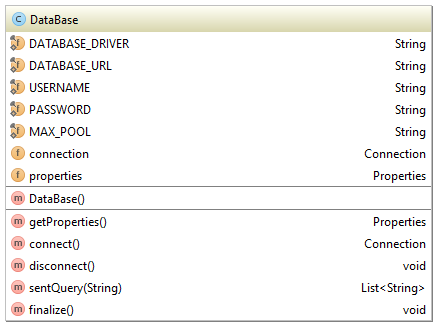


Рисунок 3.12 – Методы класса *DataBase*

Класс *DataBase* отвечает за работу с базой данных. Содержит методы установление соединения с базой, закрытие соединения и отправку запроса к базе и получения.

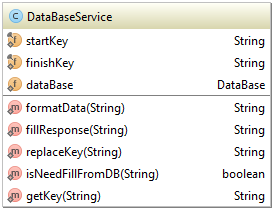


Рисунок 3.13 – Методы класса *DataBaseService*

Класс *DataBaseService* является посредником между контроллером и классом *DataBase*. Позволяет динамически добавлять в сообщения бота персистентную информацию, хранящуюся в базе данных.

Место инжекции дополнительной информации определяется фигурными скобками. Метод *getKey* выделяет маркер персистентной информации. А метод *replaceKey* заменяет маркер на полученную из базы данных информацию, предварительно форматированную в удобочитаемый пользователю формат, методом *formatData*.

Клиентская часть приложения представлена файлами форматов: *jsp*, *html*, *css*, *js*. Содержимое основных файлов представлено в приложении А.

Файл *login.jsp* содержит форму для логирования пользователя в системе и предоставляет ему возможность присоединиться к чату.

Файл *chat.html* предназначен для отображения пользователю основной страницы веб-приложения. Содержит форму отображения сообщений, поле ввода нового сообщения пользователя и кнопку отправки.

Файл *chat.js* содержит *javascript* код выполняющий всю работу по взаимодействию пользователя и серверной части приложения.

Приложение *TelegramBot* – консольное *Java* приложение, представляющее собой обертку для разрабатываемого чат-бота и отвечающее за интеграцию этого бота в *Telegram*. Данное приложение написано с использованием *API* *TelegramBots.* Для работы приложения предварительно был создан и зарегистрирован бот в мессенджере *Telegram*.

При регистрации нового бота в *Telegram*, необходимо было указать имя бота, после чего был выдан *token* для подключения к каналу созданного бота. Эти настройки хранятся в классе *BotConfig*.

Класс *TelegramBot* отвечает за запуск приложения и подключение его к серверам *Telegram*. Структура классов представлена на рисунке 3.14.

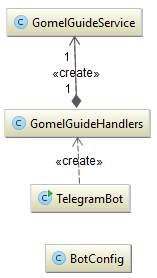


Рисунок 3.14 – Структура классов *TelegramBot*

Класс *GomelGuideHandlers* получает сообщения пользователя и отправляет ответ. Содержит методы *onUpdateReceived* и *sendMsg*.

Метод *onUpdateReceived* вызывается при получении сообщения от пользователя, передает сообщение пользователя в метод *getAnswer* класса *GomelGuideService* и вызывает метод *sendMsg*.

Метод *sendMsg* формирует ответное сообщение и отправляет его пользователю.

Передает сообщение пользователя в метод *getAnswer* класса *GomelGuideService.* Формирует ответное сообщение и отравляет.

Класс *GomelGuideService* содержит метод *getAnswer* принимающий в качестве входного параметра строку *message*. Этот метод создает *http* соединение к созданному веб-приложению (чат-боту). Отправляет *post* запрос с параметром *message,* в рамках этого соединения и возвращает полученный ответ.

**3.5 Описание интерфейса пользователя**

Для начала работы с приложением необходимо ввести *url*-адрес сайта в браузере. После чего откроется страница с предложением ввести свое имя и присоединиться к чату. Пример можно увидеть на рисунке 3.11.



Рисунок 3.11 – Стартовая страница приложения

После нажатия кнопки «*Join Chat*»откроется страница с полем сообщений и формой ввода нового сообщения.

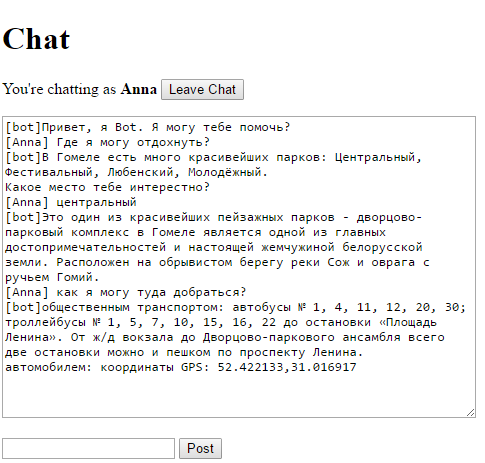


Рисунок 3.12 – Окно диалога

Для отправки сообщения необходимо ввести текст в пустое поле и нажать кнопку «*Post*». После чего отправленное сообщение и ответ на него появятся в поле чата.

По нажатию кнопки «*Leave Chat*» снова откроется стартовая страница. Если войти в чат под предыдущим именем, то история диалога будет сохранена и отображена на странице чата.

Для окончания работы с приложением достаточно закрыть окно браузера.

Для работы с ботом через мессенджер *Telegram*, необходимо открыть соответствующее приложение и найти в поиске бота *GomelGuide*. После чего откроется страница диалога с ботом. Пример переписки с ботом изображен на рисунке 3.13.

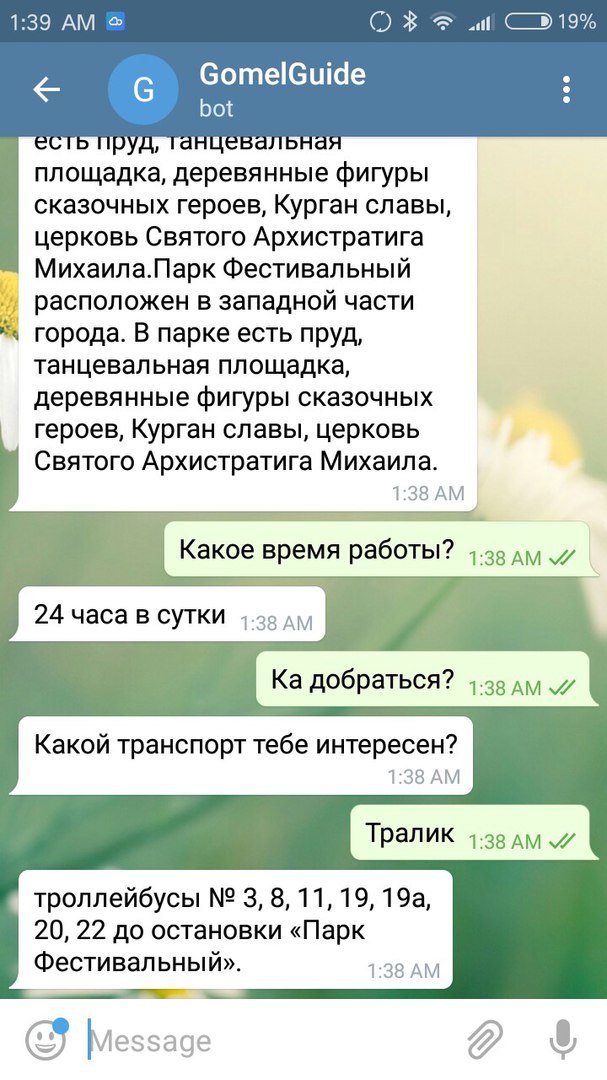
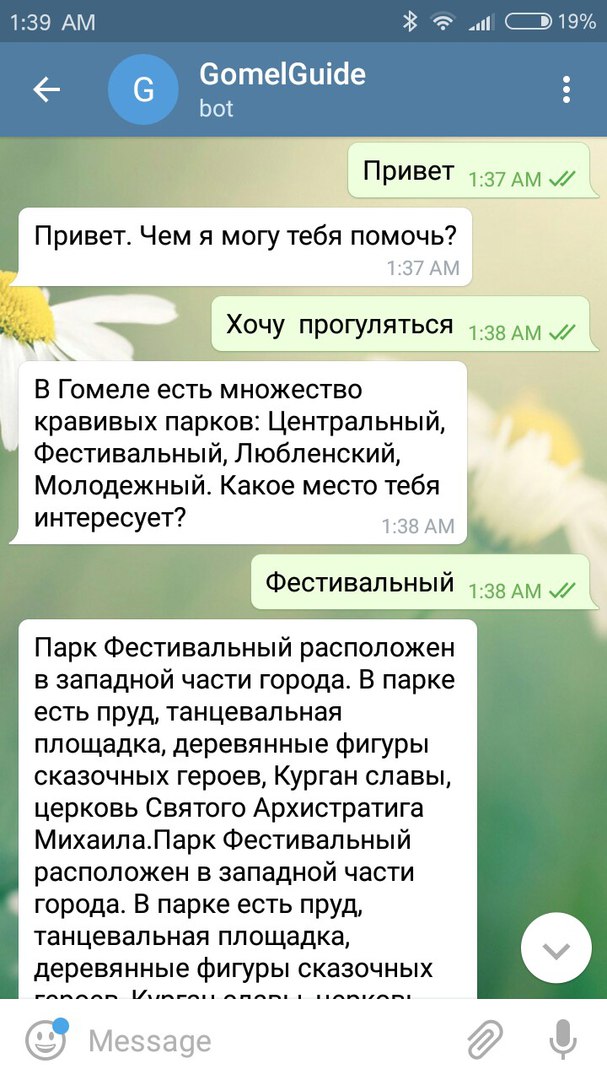


Рисунок 3.13 – Окно диалога в *Telegram*

Отправка сообщений в данном мессенджере происходит посредством ввода текста в соответствующее поле и нажатием клавиши ввода на экранной клавиатуре. Для инициации разговора с ботом необходимо отправить ему любое сообщение.

Для прекращения разговора достаточно выйти из окна диалога с ботом, либо закрыть приложение.