Министерство науки и высшего образования Российской Федерации
Федеральное государственное бюджетное образовательное учреждение высшего образования
«Московский государственный технический университет имени Н. Э. Баумана
(национальный исследовательский университет)»
(МГТУ им. Н. Э. Баумана)

ФАКУЛЬТЕТ «Информатика и системы управления»

КАФЕДРА «Программное обеспечение ЭВМ и информационные технологии»

РАСЧЕТНО-ПОЯСНИТЕЛЬНАЯ ЗАПИСКА
К КУРСОВОЙ РАБОТЕ
НА ТЕМУ:
Веб-приложение для обмена данными через локальный почтовый сервис

Студент ИУ7-74Б _____ Керимов А. Ш.
(Группа) (Подпись, дата) (Фамилия И. О.)

Студент ИУ7-74Б _____ Новиков М. Р.
(Группа) (Подпись, дата) (Фамилия И. О.)

Руководитель курсовой работы _____ Рогозин Н. О.
(Подпись, дата) (Фамилия И. О.)

2020 г.

Содержание

Введение

Технология электронной почты появилась в 1965 году — сотрудники Массачусетского технологического института Ноэль Моррис и Том Ван Влек написали программу mail для операционной системы CTSS. Однако коммерческое использование электронной почты началось только в 1990-х с запуском сервиса Hotmail.

Общепринятым в мире протоколом обмена электронной почтой является протокол SMTP, который использует DNS для определения правил пересылки почты. Стандарт протокола был впервые описан в 1982 году (RFC 821), а затем дополнен в 2008 году (RFC 5321). Почтовые серверы используют SMTP для отправки и получения почтовых сообщений. Работающие на уровне пользователя клиентские почтовые приложения обычно используют SMTP только для отправки писем на почтовый сервер, а для получения сообщений применяют другие протоколы (POP, IMAP).

Курсовая работа посвящена разработке веб-приложения для обмена данными через локальный почтовый сервис и библиотек, реализующих SMTP-сервер и SMTP-клиент.

1 Аналитический раздел

1.1 Постановка задачи

В соответствии с техническим заданием на курсовую работу, необходимо разработать веб-приложения для обмена данными через локальный почтовый сервис.

Для решения поставленной задачи необходимо:

1. Провести анализ протокола SMTP.
2. Разработать алгоритмы, реализующие протокол SMTP.
3. Разработать библиотеку, реализующую SMTP-сервер.
4. Разработать библиотеку, реализующую SMTP-клиент.
5. Разработать веб-приложение для обмена данными через почтовый сервис.

1.2 Требования к разрабатываемому ПО

В соответствии с техническим заданием на курсовую работу, были установлены следующие требования к разрабатываемому программному обеспечению:

1. SMTP-сервер должен быть запущен локально.
2. Для получения сообщения сервер должен использовать разработанную библиотеку.
3. Для отправки сообщения веб-приложение должно использовать разработанную библиотеку.

4. Веб-приложение должно, помимо отправки сообщения, предусматривать авторизацию и регистрацию пользователей, просмотр входящих и исходящих сообщений, удаление сообщений.

1.3 Анализ протокола SMTP

SMTP (англ. Simple Mail Transfer Protocol — простой протокол передачи почты) — это сетевой протокол, предназначенный для передачи электронной почты между сервером отправителя и почтовым клиентом или сервером получателя через надёжный канал, в роли которого обычно выступает TCP-соединение. Взаимодействие в рамках SMTP строится по принципу двусторонней связи, которая устанавливается между отправителем и получателем почтового сообщения. При этом отправитель инициирует соединение и посылает запросы на обслуживание, а получатель отвечает на эти запросы.

Операция протокола включает в себя комбинацию, состоящую из следующих последовательностей команд и ответов:

- MAIL FROM — команда, устанавливающая обратный электронный адрес;
- RCPT TO — команда, определяющая получателя письма;
- DATA — команда, отвечающая за отправку текста электронного сообщения — это тело письма, которое включает в себя заголовок и текста письма, разделенные пустой строкой.

Помимо промежуточных ответов для DATA-команды, каждый ответ сервера может быть положительным или отрицательным. Последний, в свою очередь, может быть постоянным либо временным. Отказ SMTP-сервера в передаче сообщения — постоянная ошибка. В этом случае клиент должен отправить возвращённое письмо. После сброса, то есть положительного ответа, сообщение скорее всего будет отвержено. Кроме того сервер может сообщить о том, что ожидаются дополнительные данные от клиента.

Согласно секции 4.5.1 RFC 5321 помимо трёх перечисленных, любой SMTP сервер должен поддерживать команды:

- `EHLO`, `HELO` — команда-приветствие, обозначает начало сессии. Предпочтительнее использование команды `EHLO`.
- `RSET` — команда, указывающая, что текущая почтовая транзакция будет прервана.
- `NOOP` — команда, не влияющая на параметры или ранее введенные команды. Она не определяет никаких действий, кроме отправки получателем ответа 250 OK.
- `QUIT` — команда, указывающая, что получатель должен отправить ответ 221 OK, а затем закрыть канал передачи.
- `VRFY` — команда, запрашивающая у получателя подтверждение, что аргумент идентифицирует пользователя или почтовый ящик.

1.4 Обзор существующих решений

Рассмотрим некоторые существующие решения, на базе которых можно организовать почтовую связь в локальной сети.

Sendmail — агент передачи почты с открытым исходным кодом, разработанный Эриком Оллманом в 1983 году. Является кросс-платформенным, поддерживает множество способов аутентификации. К достоинством данного программного обеспечения можно отнести портативность и гибкость, а к недостаткам — сложность модификации, слабые механизмы безопасности.

Postfix — свободное программное обеспечение, которое разрабатывалось как альтернатива Sendmail. Данный агент передачи почты кросс-платформенный, ориентирован, в первую очередь, на безопасность, имеет исчерпывающую документацию, обеспечивает высокую скорость работы. Postfix полностью совместим с Sendmail. К недостаткам можно отнести сложность при настройке и поддержке сервиса.

Exim — это агент пересылки почты, который обычно используется в Unix-подобных операционных системах. Имеет монолитную архитектуру.

Достоинства сервиса: гибкость конфигурации, большое сообщество, совместимость с Sendmail. К недостатка можно отнести сложность работы в сравнении с другими агентами, монолитную архитектуру ПО.

1.5 Выводы

В результате анализа технического задания на курсовую работу была поставлена задача, были определены основные требования к разрабатываемому ПО.

В результате анализа протокола SMTP были определены принципы его работы, что позволяет перейти к проектированию библиотек, реализующих SMTP-сервер и SMTP-клиент.

Кроме того, были рассмотрены существующие решения, определены их достоинства и недостатки.

2 Конструкторский раздел

2.1 Проектирование библиотек

2.1.1 Библиотека, реализующая SMTP-сервер

Библиотека должна представлять интерфейс для реализации SMTP-сервера.

Необходимо реализовать перечисленные в аналитической части команды протокола SMTP согласно спецификации RFC 5321.

SMTP-сервер можно описать конечным автоматом состояний. Диаграмма состояний SMTP-сервера представлена на рисунке 2.1.

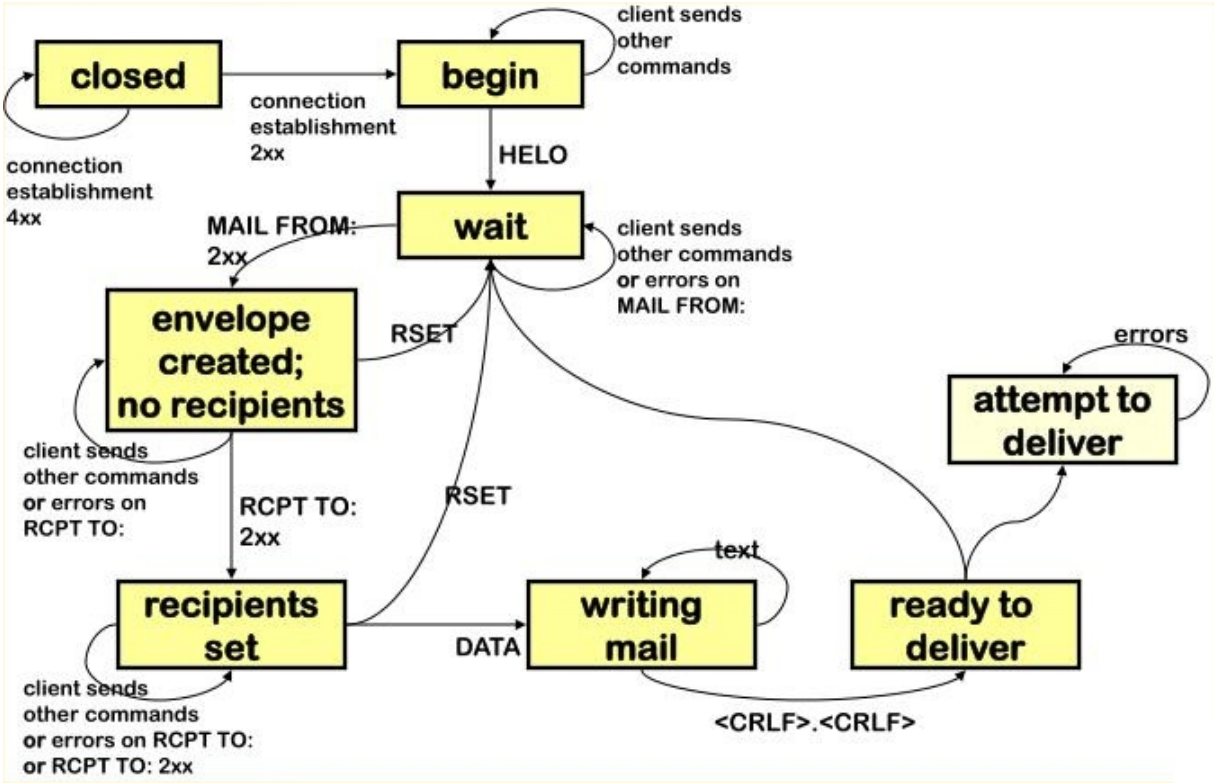

Рис. 2.1: Диаграмма состояний SMTP-сервера

Синтаксис команд представлен в листинге 2.1.

Листинг 2.1: Синтаксис поддерживаемых команд

```
EHLO <hostname>
HELO <hostname>
MAIL FROM: <address>
RCPT TO: <address>
DATA
RSET
NOOP
QUIT
```

Помимо вышеописанных команд, библиотека должна предоставлять интерфейс функции обработки сообщения, которая позволит произвести журналирование, запись в базу данных и прочее.

2.1.2 Библиотека, реализующая SMTP-клиент

Для работы через протокол SMTP клиент создаёт TCP соединение с сервером. Затем клиент и SMTP-сервер обмениваются информацией пока соединение не будет закрыто или прервано.

Библиотека должна поддерживать как интерфейс команд протокола SMTP отдельно, так и единую функцию, инкапсулирующую работу с сервером. Такая функция на вход будет принимать адрес электронной почты отправителя, список адресов электронных почт получателей, заголовок письма и его содержимое.

Схема режима работы SMTP-клиента представлена на рисунке 2.2.

2.2 Проектирование веб-приложения

2.2.1 База данных

Прежде всего необходимо спроектировать базу данных для хранения информации о сообщениях и пользователях почтового сервиса.

Каждый пользователь веб-приложения описывается именем, электронным адресом и паролем.

Каждое сообщение описывается электронным адресом отправителя и получателей, темой письма, временем отправки либо получения письма и содержанием письма.

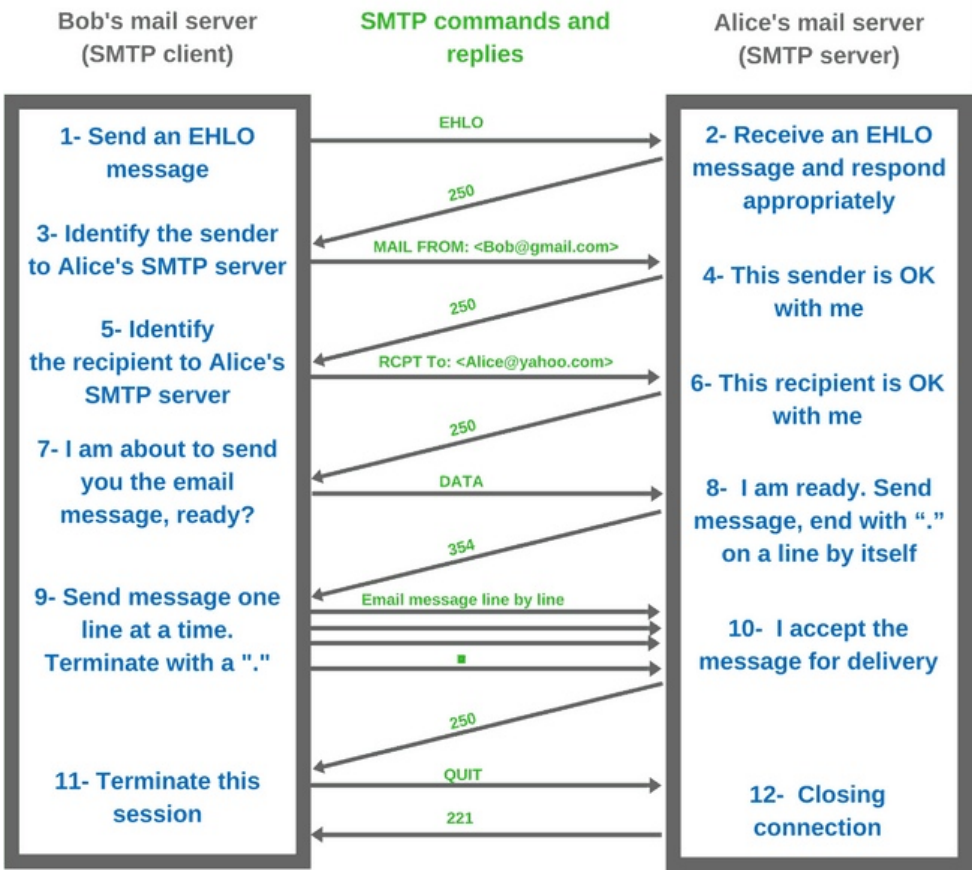

Рис. 2.2: Схема режима работы SMTP-клиента

Кроме того, необходимы дополнительные сущности, описывающие наличие письма во входящем или исходящем ящиках для обеспечения возможности удаления.

На рисунке 2.3 представлена схема базы данных.

2.2.2 SMTP-сервер

SMTP-сервер должен использовать разработанную библиотеку.

В перегруженной функции обработки сообщения необходимо произвести следующие действия:

1. сохранить сообщение в таблицу Message;
2. сохранить запись исходящей почты в таблице MessageFrom;
3. для каждого получателя сохранить запись входящей почты в таблице MessageTo;

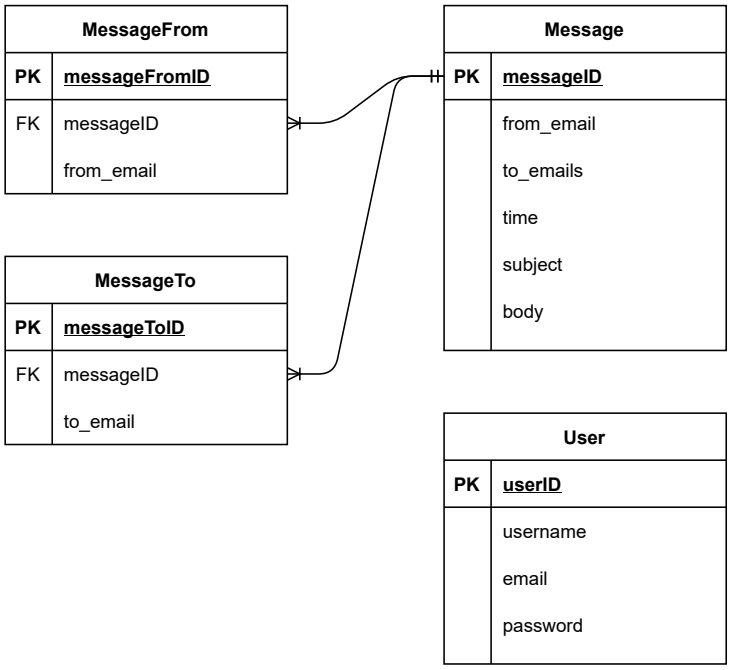

Рис. 2.3: Схема базы данных

2.2.3 Архитектура приложения

Для проектирования приложения применим архитектурый шаблон проектирования Model-View-Controller (далее — MVC).

Архитектурный шаблон MVC

MVC представляет из себя схему разделения данных и бизнес-логики приложения, пользовательского интерфейса и управляющей логики на три независимых компонента: модель, представление и контроллер. Такой подход позволяет изолировать данные и управляющую логику, независимо разрабатывать, тестировать, поддерживать и модифицировать компоненты. Схема шаблона MVC представлена на рисунке 2.4.

Модель представляет собой данные и методы для работы с данными. В модели выполняются запросы к базе данных, бизнес-логика. Этот компонент разрабатывается таким образом, чтобы отвечать на запросы контроллера, изменять свое внутреннее состояние и не зависеть от представлений.

Представление получает данные модели и отображает их пользователю. Представление не обрабатывает данные.

Контроллер является связующим компонентом — интерпретирует

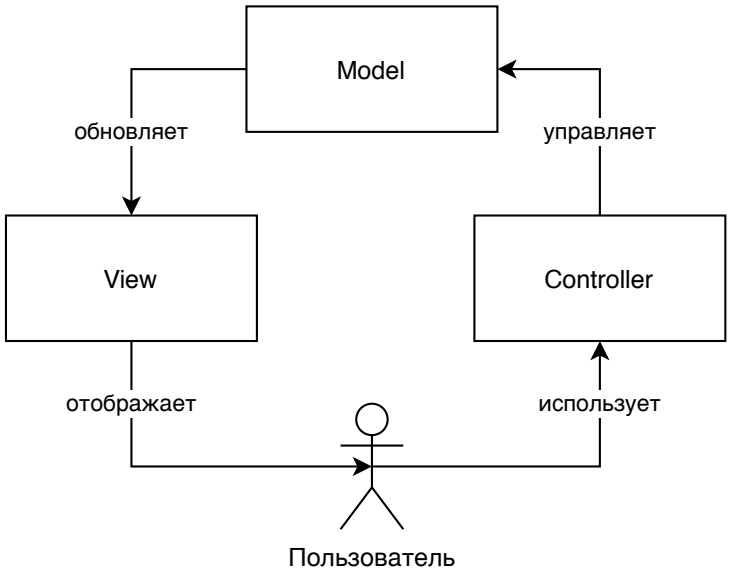

Рис. 2.4: Диаграмма шаблона MVC

действия пользователя, оповещая модель об изменениях, которые необходимо внести.

UML-диаграмма компонентов приложения

На рисунке 2.5 представлена UML-диаграмма компонентов веб-приложения.

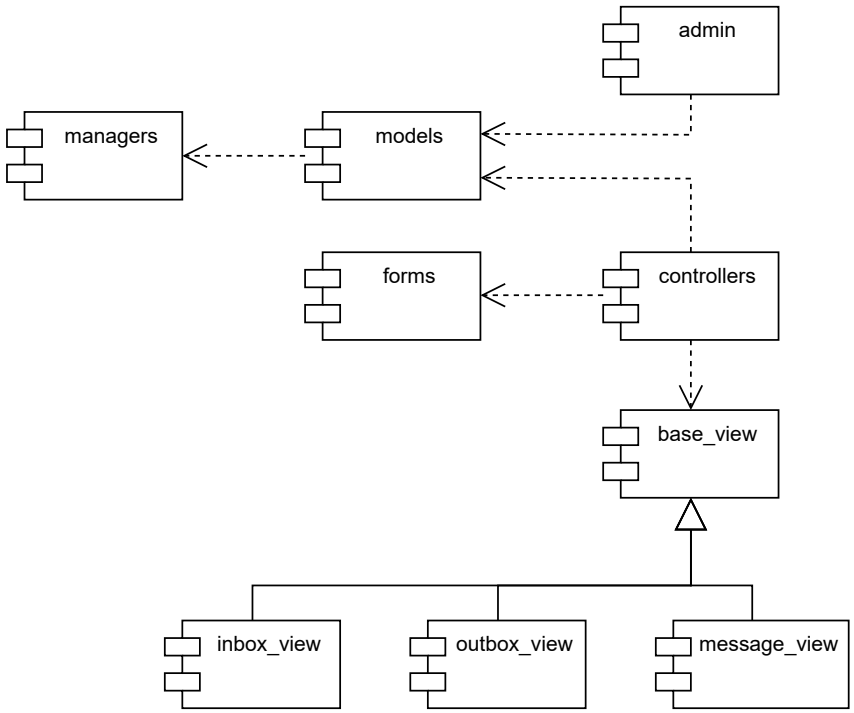

Рис. 2.5: UML-диаграмма компонентов приложения

2.3 Выводы

В конструкторском разделе были спроектированы библиотеки, реализующие протокол SMTP. Были спроектированы база данных и архитектура приложения, построена UML-диаграмма компонентов веб-приложения.

3 Технологический раздел

3.1 Выбор языка программирования и среды разработки

В качестве языка программирования был выбран язык Python.

В качестве фреймворка для разработки веб-приложения был выбран фреймворк Django. Django является одним из наиболее популярных фреймворков для разработки веб-приложений на Python. В качестве СУБД был выбран SQLite, который используется в Django по умолчанию.

В качестве среды разработки была выбрана IDE PyCharm. PyCharm содержит редактор кода, отладчик, средства для статического анализа кода, средства для сборки проекта, тесную интеграцию с фреймворком Django.

3.2 Реализация библиотек

3.2.1 Библиотека, реализующая SMTP-сервер

Для каждой команды релизован соответствующий обработчик. На листинге 3.1 представлен обработчки для команды EHLO.

Листинг 3.1: Обработчик для команды EHLO

```
def handle_EHLO(self, arg: Char) -> None:
        if not arg:
                return self.push(f'501 Syntax: {SmtpStream.syntax_EHLO()}')
        if self.__seen_greeting:
                return self.push('503 Bad sequence of commands: duplicate
                   HELO/EHLO')
        self.__set_reset_state()
```

```
        self.__seen_greeting = arg
        self.__extended_smtp = True
        self.push(f'250-{self.__fqdn}')
        if self.__data_size_limit:
                self.push(f'250-SIZE {self.__data_size_limit}')
                self.command_size_limits['MAIL'] += 26
        if not self.__decode_data:
                self.push('250-8BITMIME')
                self.push('250 HELP')
```

В методе `__found_terminator_in_command_state` (см. листинг 3.2) вызывается обработчик команды. Если обработчик не найден — ошибка 500.

Листинг 3.2: Вызов обработчика команды EHLO

```
...
method = getattr(self, 'handle_' + command, None)
if not method:
        return self.push(f'500 Command unrecognized: `{command}`')
method(arg)
...
```

3.2.2 Библиотека, реализующая SMTP-клиент

В библиотеке SMTP-клиента реализованы методы для отправки команд. На листинге 3.3 представлен метод, реализующий отправку коменды EHLO.

Листинг 3.3: Отправка команды EHLO

```
def ehlo(self, name=''):
        self.__supported_esmtp_features = {}
        self.put_command(self.ehlo_msg, name or self.__local_hostname)
        code, msg = self.getreply()
        if code == -1 and len(msg) == 0:
                self.close()
                raise SmtpServerDisconnected('Server not connected')
        self.__last_ehlo_response = msg
        if code != 250:
                return code, msg
        self.__is_server_supports_esmtp = 1
        assert isinstance(self.__last_ehlo_response, bytes),
           repr(self.__last_ehlo_response)
        resp = self.__last_ehlo_response.decode("latin-1").split('\n')
        del resp[0]
        for each in resp:
```

```
16                m = re.match(r'(?P<feature>[A-Za-z0-9][A-Za-z0-9\-]*) ?',
                      each)
17                if m:
18                        feature = m.group('feature').lower()
19                        params = m.string[m.end('feature'):].strip()
20                        self.__supported_esmtp_features[feature] = params
21        return code, msg
```

Метод `send_msg()` вызывает команду ehlo (или helo, если это необходимо), если эти команды не были вызваны ранее, затем выполняет транзакт. Данный метод представлен на листинге 3.4.

Листинг 3.4: Метод отправки сообщения

```
1         def send_msg(self, from_addr, to_addrs, msg, mail_options=(),
              rcpt_options=()):
2         self.ehlo_or_helo_if_needed()
3         esmtp_opts = []
4         if isinstance(msg, str):
5                 msg = fix_eols(msg).encode('ascii')
6         if self.__is_server_supports_esmtp:
7                 if self.has_extn('size'):
8                         esmtp_opts.append(f'size={len(msg)}')
9                 for option in mail_options:
10                        esmtp_opts.append(option)
11        code, resp = self.mail(from_addr, esmtp_opts)
12        if code != 250:
13                if code == 421:
14                        self.close()
15                else:
16                        self.__rset()
17                raise SmtpSenderRefused(code, resp, from_addr)
18        senders = {}
19        if isinstance(to_addrs, str):
20                to_addrs = [to_addrs]
21        for each in to_addrs:
22                code, resp = self.rcpt(each, rcpt_options)
23                if (code != 250) and (code != 251):
24                        senders[each] = (code, resp)
25                if code == 421:
26                        self.close()
27                raise SmtpRecipientsRefused(senders)
28        if len(senders) == len(to_addrs):
29                self.__rset()
30                raise SmtpRecipientsRefused(senders)
31        code, resp = self.data(msg)
32        if code != 250:
33                if code == 421:
34                        self.close()
```

```
                else:
                        self.__rset()
                raise SmtpDataError(code, resp)
        return senders
```

3.3 Реализация веб-приложения

3.3.1 Реализация SMTP-сервера

Классы `SmtpInbox` и `SmtpInboxServer` представлены на листинге 3.5

Листинг 3.5: Синтаксис поддерживаемых команд

```
class SmtpInboxServer(SmtpServer, object):
        def __init__(self, handler, *args, **kwargs):
                super().__init__(*args, **kwargs)
                self.__handler = handler

        def process_message(self, peer, mailfrom, rcpttos, data, **kwargs):
                log.info(f'Collating message from {mailfrom}')
                subject = Parser().parsestr(data)['subject']
                log.debug(dict(to=rcpttos, sender=mailfrom,
                    subject=subject, body=data))
                return self.__handler(to=rcpttos, sender=mailfrom,
                    subject=subject, body=data)


class SmtpInbox(object):
        def __init__(self, port=None, address=None):
                self.port = port
                self.address = address
                self.collator = None

        def collate(self, collator):
                self.collator = collator
                return collator

        def serve(self, port=None, address=None):
                port = port or self.port
                address = address or self.address

                log.info(f'Starting SmtpServer at {address}:{port}')

                server = SmtpInboxServer(self.collator, (address, port),
                    None, decode_data=True)
```

```
30
31                  try:
32                          asyncore.loop()
33                  except KeyboardInterrupt:
34                          log.info('Cleaning up')
35
36  if __name__ == '__main__':
37          inbox = SmtpInbox()
38
39          @inbox.collate
40          def handle(to, sender, subject, body):
41                  if subject is None:
42                  subject = ''
43                  try:
44                          time = datetime.datetime.now().strftime("%Y-%m-%d
                               %H:%M:%S")
45                          message_id = execute_statement(
46                                  f'INSERT INTO webmail_message '
47                                  f'(from_email, to_emails, time, subject,
                                       body) '
48                                  f'VALUES '
49                                  f'(\"{sender}\", \"{" ".join(to)}\",
                                       \"{time}\",        \"{subject}\",
                                       \"{body[10 + len(subject):]}\")'
50                          )
51                          execute_statement(
52                                  f'INSERT INTO webmail_messagefrom'
53                                  f'(from_email, message_id)'
54                                  f'VALUES'
55                                  f'(\"{sender}\", \"{message_id}\")'
56                          )
57                          for recipient in to:
58                          execute_statement(
59                                  f'INSERT INTO webmail_messageto'
60                                  f'(to_email, message_id)'
61                                  f'VALUES'
62                                  f'(\"{recipient}\", \"{message_id}\")'
63                          )
64                  except sqlite3.Error as e:
65                          print('Failed to insert data:', e)
66
67
68          inbox.serve(address='0.0.0.0', port=4467)
```

SMTP-сервер запускается по адресу 0.0.0.0 и использует порт 4467.

SMTP-сервер создает объект класса `SmtpInbox` и переопределяет функцию `handle()`, которая при получении сообщения выполняет

запись в таблицы Message, MessageTo и MessageFrom базы данных веб-приложения.

Функция `execute_statement()` открывает соединение с базой данных, выполняет переданную ей в качестве параметра SQL-операцию (`INSERT INTO`), сохраняет изменения и закрывает соединение.

3.3.2 Реализация приложения

Реализация шаблона проектирования

В основе Django лежит шаблон проектирования MVC, который во фреймворке называется Model-View-Template, где Model — модель, являющаяся фактически ORM-сущностью, View — контроллер, Template — представление. Бизнес-логику в Django принято выделять в отдельный компонент.

Компоненты приложения

Выделим три компонента: компонент доступа к данным, компонент графического интерфейса пользователя, компонент, связывающий данные и графический интерфейс пользователя.

Компонент доступа к данным представляет из себя классы `django.db.models.Models` (далее — модели), данные в которых соответствуют атрибутам таблиц в базе данных. Модели содержат методы для обработки данных на уровне строки, например метод `get_to_emails()`, возвращающий список адресатов сообщения. На листинге 3.6 представлена модель `Message`, соответствующая таблице Message в базе данных.

Листинг 3.6: Модель `Message`

```
class Message(models.Model):
    from_email = models.CharField(max_length=254)
    time = models.DateTimeField()
    subject = models.TextField()
    body = models.TextField()

    objects = MessageManager()

    def get_to_emails(self):
```

```
        to_emails = []
        for recipient in self.recipient.all():
            to_emails.append(recipient.to_email)
        return to_emails
```

Для работы с данными на уровне таблицы используются классы, называемые менеджерами. Эти классы наследуются от `django.db.models.Managers` и содержат методы для доступа к данным. На листинге 3.7 приведен пример менеджера.

Листинг 3.7: Менеджер `MessageManager`

```
class MessageManager(models.Manager):
    def get_inbox(self, email):
        return self.filter(recipient__to_email=email).order_by('-time')

    def get_outbox(self, email):
        return self.filter(from_email=email).order_by('-time')
```

Классы-контроллеры в Django наследуются от класса `django.views.generic.DefaultView`. При этом существуют классы для выполнения типичных задач представления данных: для отображения списка объектов — `ListView`, для отображения информации о конкретном объекте — `DetailView` и др. На листинге 3.8 представлен класс `MessageDetailView`.

Листинг 3.8: Контроллер `MessageDetailView`

```
@method_decorator(login_required, name='dispatch')
class MessageDetailView(DetailView):
    model = Message
    template_name = 'webmail/message.html'
    context_object_name = 'message'

    def get_object(self, *args, **kwargs):
        entity = super().get_object(*args, **kwargs)
        if self.request.user.email not in entity.get_to_emails() +
           [entity.from_email]:
            raise Http404
        return entity
```

Компонент интерфейса представляет из себя набор HTML-страниц, использующих шаблонизатор Django, который позволяет использовать шаблоны для генерации конечных страниц. Шаблонизатор позволяет, в частности, переиспользовать код и ускоряет верстку веб-приложения. Данные передаются из контроллера в качестве параметров на страницу.

3.3.3 Интерфейс

Интерфейс приложения представляет из себя страницу, в шапке которой расположено название приложение, навигационное меню, позволяющее перейти к спискам входящих (Inbox) и исходящих сообщений (Outbox), информацию об авторизованном пользователе.

На странице Inbox расположена таблица, содержащая входящие письма. Каждое письмо представлено адресантом, темой письма и содержимым письма, которое при переполнении первой строки скрывается. Для каждого письма предусмотрена кнопка удаления. Структура страницы Outbox аналогична. При нажатии на письмо происходит переход на страницу Message, на которой отображена более подробная информация о письме: адресат и список адресантов, время получения письма, тема письма, полное содержимое письма. На каждой странице приложения справа от заголовка страницы расположена кнопка Compose, которая открывает модальное окно, позволяющее написать письмо.

На рисунках 3.1–3.4 представлен интерфейс веб-приложения.

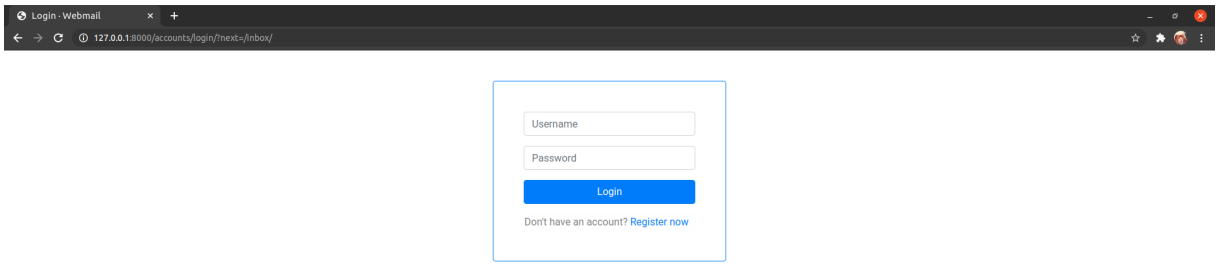

Рис. 3.1: Авторизация в веб-приложении

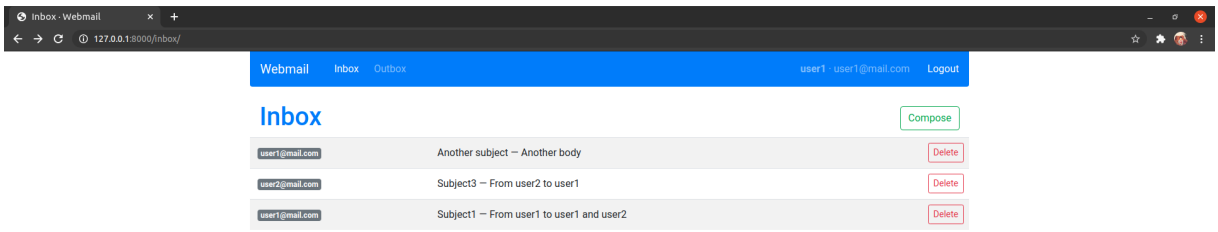

Рис. 3.2: Вкладка входящих сообщений

3.4 Тестирование и отладка

Для программы использовался отладчик и статический анализатор кода, встроенные в среду разработки PyCharm.

Библиотеки, SMTP-сервер и веб-приложение были протестированы в полном объеме, все обнаруженные ошибки были исправлены.

3.5 Выводы

В результате разработки было реализовано программное обеспечение в полном соответствии с техническим заданием и предъявляемыми требованиями.

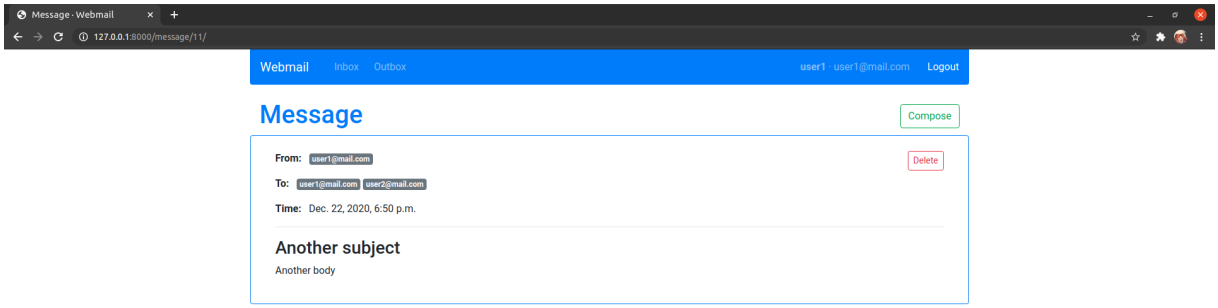

Рис. 3.3: Просмотр сообщения

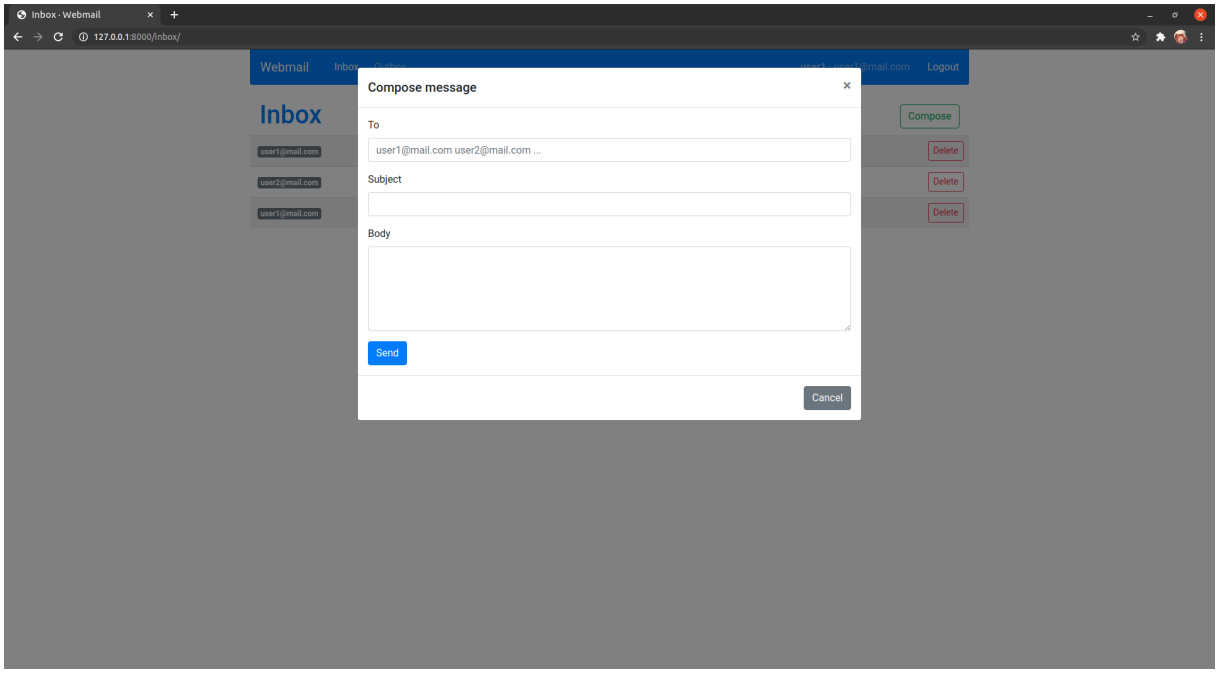

Рис. 3.4: Создание сообщения

Заключение

В результате выполнения курсовой работы были разработаны веб-приложение для обмена данными через локальный почтовый сервис и библиотеки, реализующие протокол SMTP.

В процессе выполнения курсовой работы были выполнены все поставленные задачи, а именно: проанализирован протокол SMTP; спроектированы и разработаны библиотеки, реализующие протокол SMTP, SMTP-сервер; разработано веб-приложение для обмена данными через почтовый сервис. Программное обеспечение было протестировано в полном объеме.

С развитием в данный проект может быть добавлено следующее:

- аутентификация;
- SSL-подключение;
- функции ответа и пересылки сообщения;
- корзина, для хранения удаленных сообщений.

Список использованных источников

1. Hughes, L. Internet e-mail Protocols, Standards and Implementation (англ.). — Artech House Publishers (англ.), 1998. — ISBN 0-89006-939-5.

2. Стандарт RFC 5321 [Электронный ресурс] / Режим доступа — https://tools.ietf.org/html/rfc5321 (дата обращения: 05.11.2020)

3. Документация языка программирования Python [Электронный ресурс] / Режим доступа — https://www.python.org/doc/ (дата обращения: 05.11.2020)

4. Документация фреймворка Django [Электронный ресурс] / Режим доступа — https://docs.djangoproject.com/en/3.1/ (дата обращения: 05.11.2020)

5. Документация фреймворка Bootstrap [Электронный ресурс] / Режим доступа — https://getbootstrap.com/docs/5.0/getting-started/introduction/ (дата обращения: 05.11.2020)