УДК 621.396.06 Научная статья

https://doi.org/10.32603/2071-8985-2023-16-4-72-77

# Спецификация разработки программы передачи информации пользователю и технической системе

**Н. А. Смирнов**

Санкт-Петербургский государственный электротехнический университет «ЛЭТИ» им. В. И. Ульянова (Ленина), Санкт-Петербург, Россия

smirnov-n1@mail.ru

**Аннотация.** В статье наглядно показана реализация приложения для обмена информацией как между людьми, так и между человеком и техникой. Создание такой программы возможно во многом благодаря библиотеке Flask, которая позволит упростить синтаксис программы. Проведен разбор функционала приложения с точки зрения архитектуры с использованием декораторов для более удобного отображения. Разобраны технические аспекты, архитектура взаимодействия модулей программы и написание кода. Показано, что пользователь с начальным и средним уровнем знания программирования может с помощью Flask и ряда других библиотек создать полноценную программу для отображения информации, не обязательно технического характера, и использовать ее в своих проектах.

**Ключевые слова:** программа, отображение, обработка, сервер, архитектура запросов, работа с модулями

**Для цитирования:** Смирнов Н. А. Спецификация разработки программы передачи информации пользователю и технической системе // Изв. СПбГЭТУ «ЛЭТИ». 2023. Т. 16, № 4. С. 72–77. doi: 10.32603/2071-8985-2023-16-4-72-77.

Original article

# Specification of the Development of a Program for Transmitting Information to Users and Technical Systems

**N. A. Smirnov**

Saint Petersburg Electrotechnical University, Saint Petersburg, Russia

smirnov-n1@mail.ru

**Abstract.** This article demonstrates how to implement an application for the exchange of information, both between people and between man and technical equipment. The Flask library facilities the creation of such a program by simplifying the syntax of the program. An analysis of the application functionality was carried out in terms of its architecture, using decorators for more convenient display. Various technical aspects, the interaction architecture of program modules, as well as the questions of code writing, are discussed. It is shown that the use of Flask and some other libraries allow users with elementary and intermediate level of programming knowledge to create a full-fledged program to display information, not necessarily of technical nature, for its further application in various projects.

**Keywords:** program, display, processing, server, query architecture, working with modules

**For citation:** Smirnov N. A. Specification of the Development of a Program for Transmitting Information to Users and Technical Systems // LETI Transactions on Electrical Engineering & Computer Science. 2023. Vol. 16, no. 4. P. 72–77. doi: 10.32603/2071-8985-2023-16-4-72-77.

**Введение.** Чтобы получать техническую информацию от аппаратных средств, пользователю или специалисту, ведущему наблюдение за технической системой или аппаратурой, необходима программа для обмена информацией, которую можно использовать не только в качестве обмена

 

информации между людьми, но и между техническими средствами и человеком. На данный момент нет общедоступных решений для большинства пользователей. Существуют системы по типу «умного дома», которые способны передавать информацию о состоянии приборов в доме, однако конкретной информации о системах, предназначенных для работы на заводах и технических предприятиях с техническим оборудованием, практически нет. При работе над статьей удалось узнать о создании подобных систем для работы с аккумуляторными батареями, потенциально способных предоставлять пользователю всю информацию об устройстве и дополнительную статистику. Однако все эти разработки – крайне нишевые, под конкретные задачи. Необходимо сделать простую программу для передачи информации, которая требует минимального знания о библиотеках языка Python, в частности о Flask, PyQT5.

**Постановка задачи.** *Создание программы*, используемой для передачи информации пользователю или обмена информацией между пользователями, т. е. программы, *имеющей гибкую настройку конфигураций*. Это возможно благодаря библиотеке Flask, которая и позволит упростить синтаксис программы благодаря автоматической приемке url-запросов и использованию методов, связанных с отправкой сообщений, – до этого обычно использовалась встроенная библиотека request. Сама программа разделена на три части, однако разобраны будут только две. Третья часть представляет собой набор элементов интерфейса, генерируемых библиотекой PyQT5 при налаживании элементов поля пользователя, и логики не содержит, поэтому рассматриваться не будет. Две части как раз и составляют программу, в которой одна часть занимается хранением, отправкой и приемкой запросов от пользователя, а вторая используется как само приложение, с которыми контактирует пользователь. Программа позволяет добавлять функции на отправку и приемку сообщений в зависимости от требований пользователя, а также передавать информацию в формате мессенджера. Т. е. выдавать информацию в окно сообщений, ограничивая разовое количество, не нагружая сервер в случае, если количество пользователей и сообщений начнут нагружать серверную часть. Иными словами, сделана простая пагинация, что позволит даже человеку без технических знаний использовать ее в своих целях. Следует добавить, что устройство такой системы для передачи информации человеку, который захочет расширить программу дополнительными функциями, простота создания программы, ее обслуживания и улучшения по мере надобности служит предметом исследования в данной статье.

**Разработка серверной части.** Начать разработку программы стоит с серверной части, так как именно она – основополагающая в цепи передачи информации. Скрипт собирает нужную информацию, хранит ее и реагирует на запросы пользователя – это и есть головная часть программы. Разберем ее архитектуру сверху вниз.

Вначале импортируются все нужные для работы над проектом библиотеки – Flask, request, abort, datetime и time. Каждая библиотека отвечает за те элементы, которые нужны для работы сервера; модуль time отслеживает реальное время при работе программы. Модуль datetime нужен для присвоения даты каждому отправленному сообщению, т. е., как в любом другом мессенджере, будет видна дата отправки сообщения, это же потенциально можно использовать для фильтрации сообщений по дате. Модуль Flask – основополагающий, на его основе строится программа, заменяя собой нативный синтаксис. Этот фреймворк позволяет не перегружать программу синтаксисом и просто использовать встроенные методы для написания приема и обработки запросов от пользователя, а также работать с информацией, уже полученной программой. Следующий модуль request – это стандартный модуль для языка Python в рамках работы с вебом, обязательный при разработке API (Application Programming Interface) и других серверных программ. В целом модуль нужен для отправки страниц HTTP (HyperText Transfer Protocol). Модуль abort нужен для того, чтобы оборвать происходящий процесс на случай, если не выполнено какое-то условие. В конце создается сам экземпляр класса Flask, так как именно этот класс был импортирован из фреймворка, и в него передается параметр __name__, что также служит стандартом для этого класса. Затем создается список db (data base, база данных), он нужен для хранения сообщений.

```
import time
from datetime import datetime
from flask import Flask, request, abort

app = Flask(__name__)
db = []
```

Далее следует рассмотреть функцию status. Это первая функция данной программы, которая передает данные обо всех сообщениях, т. е. позволяет посмотреть историю сообщений и дать пользователю информацию об отправленных сообщениях и его личном номере. Сначала указывается сама функция – с помощью встроенного декоратора @app.route можно указать url-адрес, который должна получить программа для реакции на запрос. Создается переменная dn (date now, дата сейчас), после этого функция возвращает словарь с набором переменных, содержащих информацию о программе, а именно статус запроса, имя программы, дату и время, количество отправленных сообщений и количество пользователей. Стоит заметить, что время отформатировано по формату «день, месяц, год и часы, минуты, секунды». Количество сообщений можно узнать, измерив размер базы данных (списка db) с помощью стандартной функции len. Последняя конструкция сделана из генератора, который подсчитывает количество имен и делает из них список, затем использует функцию set, которая делает данный список множеством уникальных имен, и с помощью функции len подсчитывает их количество:

```
@app.route("/status")
def status():
    dn = datetime.now()
    return {
        'status': True,
        'name': 'Messenger',
        'time': dn.strftime('%d.%m.%Y
%H:%M:%S'),
        'messages_count': len(db),
        'users_count': len(set(message['name']
for message in db)),
    }
```

Следующая функция позволяет принять сообщение пользователя и добавить его в список сообщений. Для начала стоит сказать, что имя самой функции выглядит немного по-другому – тут присутствует дополнительный параметр methods, которому установлено значение ['POST'], что позволяет ему с помощью Flask автоматически без уточнений пользователя использовать ее как функцию для размещения сообщения. Т. е. она принимает полученное сообщение, которое приходит в формате json, и, достав нужные данные, размещает их на сервере и запоминает. Данный кусок программы содержит в себе переменную data с запросом с сообщением от пользователя, который был отправлен при помощи библиотеки request. После этого в список сообщений добавляется словарь, который содержит в себе следующие данные: номер сообщения – высчитывается нахождением длины самой базы данных; имя отправителя, полученное из запроса пользователя; сам текст сообщения, также взятого из имени отправителя; временная отметка, т. е. время, когда было отправлено сообщение. После чего программа возвращает словарь с меткой о выполнении добавления сообщения в список:

```
@app.route("/send",methods=['POST'])
def send():
    data = request.json
    db.append({
        'id': len(db),
        'name': data['name'],
        'text': data['text'],
        'timestamp': time.time()
    })
    return {'ok': True}
```

Разбор работы серверной части программы заканчивается функцией обработки отображения сообщений. Она показывает все сообщения в рамках заданного диапазона, т. е. отображает историю сообщений. Функция получает запрос от пользователя, в самом запросе содержатся определенные данные – такие, как временная метка. Происходит проверка: находится ли конкретное время в запросе, и если это так, то создается переменная, в которую записывается время из запроса, иначе время будет равно нулю. Дальше в программе делается пагинация, т. е. за один раз сервер будет выдавать только определенное количество сообщений, иначе сервер будет перегружен запросами на вывод всех хранящихся сообщений ввиду их большого количества. Лимит устанавливается на 100 сообщений. Если словопоказатель лимита находится в запросе, то создается переменная с лимитом из запроса, которая не должна превышать максимальный лимит, иначе программа выдаст ошибку. Также предусмотрено предоставление своего лимита, если его не отправила программа пользователя. Затем создается переменная after_id, нужная для того, чтобы понять, с какого именно сообщения нужно выдать историю сообщений. Для этого создается цикл, в котором созданную ранее временную метку сверяют с временной меткой в каждом сообщении. С момента, когда метка времени в сообщении превысит метку в запросе, нужно отобразить ис-

торию. Стоит понимать, что история обновляется в автоматическом режиме, когда пользователь заходит в приложение либо при принудительном обновлении. После того как найдено нужное сообщение, функция возвращает ответ, в котором в сообщение помещает часть базы данных от последнего сообщения, отображаемого переменной after_id, которую увеличивали каждый раз, когда параметр времени был меньше времени в запросе и до этого же числа с прибавлением лимита пагинации. В конце же программы происходит запуск работы через app.run().

```
@app.route("/messages")
def messages():
    if 'after_timestamp' in request.args:
        after_timestamp =
float(request.args['after_timestamp'])
    else:
        after_timestamp = 0
    max_limit = 100
    if 'limit' in request.args:
        limit = int(request.args['limit'])
        if limit > max_limit:
            abort(400, 'too big limit')
    else:
        limit = max_limit
    after_id = 0
    for message in db:
        if message['timestamp'] > af-
ter_timestamp:
            break
        after_id += 1
    return {'messages':
db[after_id:after_id+limt]}
app.run()
```

**Пользовательская часть.** Вторая часть программы представляет собой логику для приложения по обмену связью, т. е. часть, с которой взаимодействует сам пользователь. Такая программа обращается к серверу с запросом на получение либо же для размещения сообщения или другой информации на сервере. В данном случае у пользователя есть возможность видеть сообщения, отправлять их. Данный скрипт представляет собой большой класс, который унаследован от библиотеки PyQt5, в частности модули QtWidgets и Ui_MainWindow. Они не будут разбираться в рамках данной статьи, так как являются модулями для создания графического интерфейса, что здесь не представляет ценности ввиду рассмотрения самой работы и устройства приложения для передачи информации. В классе конструктора __init__ используется метод super, который автоматически берет методы из родительского класса. Из логических переменных нужно сказать о переменных url и after_timestamp, о которых рассказывалось ранее. Первая служит для хранения и принятия адреса запроса по типу обновления истории сообщений и т. д. Вторая отвечает за время отправки сообщения и, как следствие, связана с пагинацией; при включении клиента она изначальна равна –1, так как сообщения не должны отображаться до их обновления. Sendbutton.pressed – это реакция на нажатие клавиши на панели интерфейса при отправке сообщения на сервер. Функция load_messages отвечает за загрузку истории сообщений, в частности она взаимодействует с другой функцией. Последние строки – это дополнительная библиотека для таймера, связанная с PyQt5, и две оставшиеся также связаны со временем, это стандартная для подобного конструктора запись.

```
Class Messenger(QtWidgets.QMainWindow,
Ui_MainWindow):
    def __init__(self, url):
        super().__init__()
        self.setupUi(self)
        self.url = url
        self.after_timestamp = -1

self.sendButton.pressed.connect(self.button_pressed)
        self.load_messages()
        self.timer = QtCore.QTimer()

self.timer.timeout.connect(self.update_messages)
        self.timer.start(1000)
```

Следующая функция – это функция запроса обновления истории сообщений. Она позволяет пользователю просматривать все отправленные сообщения в окне приложения, насколько это позволяет лимит пагинации, описанной ранее. Вначале задается переменная ответа, по умолчанию она равна None, так как сообщения еще не обновлены. Затем используется конструкция try-except, часто используемая в веб-разработке, так как позволяет, несмотря на ошибку в запросе, продолжить выполнение программы. В самой же конструкции применяется метод get из модуля request, который служит для извлечения нужных данных из запроса по переданному адресу и, в частности, помимо запроса по адресу /messages, описанный ранее в функции обновления сообщений. Метод запросит переменную params, в которую поместит время последнего сообщения, так

как история начнется именно с него, и в серверной части записывается время отправки сообщения для хранения в базе данных. Если в ходе запроса произойдет ошибка, то программа ничего не будет делать. В случае успешной обработки запроса происходит проверка получения запроса и его статус-кода. Статус-код получен по умолчанию, так как это часть функционала get-запроса. Если есть ответ и он имеет статус 200, то в переменную-сообщение помещается json-файл, далее все полученные сообщения перебираются и отображаются через функцию-декоратор pretty_print либо создается более понятное представление для пользователя текста. Помимо этого передается время каждого сообщения, т. е. программа понимает, какое сообщение было последним, чтобы в следующий раз обновлять историю, начиная с него. После того как все сообщения отправлены и ответы сформированы, сервер выдает сообщения, и пользователь видит историю сообщений.

```
    def update_messages(self):
        response = None
        try:
            response = requests.get(
                self.url + '/messages',
                params = {'after_timestamp':
self.after_timestamp}
            )
        except:
            pass
        if response and response.status_code == 200:
            messages = response.json()['messages']
            for message in messages:
                self.pretty_print(message)
                self.after_timestamp = mes-
sage['timestamp']
            return messages
```

Последняя функция – это функция отправки сообщения, т. е. когда пользователь хочет отправить сообщение на сервер, он может, просто вписав его в приложение, отправить, как и в любом другом приложении для обмена информацией. В функции используется взаимодействие с файлом из библиотеки PyQt5 – текст, введенный в окне приложения, созданного с помощью данной библиотеки, передается серверной части. В начале функции создаются переменные – имя, текст и словарь, в который эти переменные помещаются. В эти переменные из окна приложения записываются данные, в частности имя пользователя и введенный текст, после чего им присваиваются имена name и text в словаре, и записанные значения оказываются внутри этого словаря. После создается переменная с ответом, но уже для отправки сообщения в отличие от прошлой функции. В знакомой конструкции try-except в переменную response записываются созданные ранее данные, в частности словарь data, и используется метод post, который также взят из модуля request, который, в отличие от get, не получает, а отправляет данные в функцию публикации сообщения, описанную ранее, для добавления сообщений в базу данных. Ответ, который отправляется серверной частью, содержит имя и текст сообщения; если же сообщения нет, то оно и будет не отправлено. Затем, если есть ответ и его статус-код, то приложение очистит строку ввода и перерисует ее для отправки нового сообщения, в противном случае выведется сообщение об ошибке, и сообщение перерисуется.

```
    def button_pressed(self):
        name = self.nameInput.text()
        text = self.textInput.toPlainText()
        data = {'name': name, 'text': text}
        response = None
        try:
            response = requests.post(self.url + '/send',
json=data)
        except:
            pass
        if response and response.status_code:
            self.textInput.clear()
            self.textInput.repaint()
        else:
            self.messagesBrowser.append('При от-
правке произошла ошибка')
            self.messagesBrowser.append('')
            self.messagesBrowser.repaint()
```

В конце программы создается экземпляр класса для интерфейса, задается порт, в примере он создается на локальном сервере и служит атрибутом созданного класса. Следует упомянуть, что в начале скрипта с логической частью пользователя были задействованы модули datetime и request. Также существует файл, генерируемый с помощью библиотеки Ui_MainWindow, однако по сути она генерирует шаблон в зависимости от настройки расположения элементов в окне пользователя и не содержит логики. Стоит уточнить, что основная логика описана именно в функциях, и ценность представляет именно она, так как остальная часть – просто типовая запись для дан-

ных фреймворков. Можно упомянуть и функцию pretty_print, не разбиравшуюся в данной статье, так как она не содержит логических элементов, однако существует, чтобы в приемлемом виде отображать получаемую информацию. Без функции-декоратора пользователь увидит стандартную, нативную форму записи языка Python. Это можно понять, создав простую функцию, в которой полученные данные разбираются на части и представляются в понятном для человека виде. Из запроса данные достаются и помещаются в определенные формы, заранее подготовленные строки с добавленной разметкой для чтения, состоящей из нативных элементов.

```
app = QtWidgets.QApplication([])
window = Messenger('https://127.0.0.1/5000/')
window.show()
app.exec_()
```

**Выводы.** Программа создана для реального использования в небольших проектах. Однако, в отличие от многих библиотек, которые не позволяют сделать полноценное приложение, ее можно даже применять на практике. Единственная сложность состоит в том, что для работы такого приложения потребуется купить облачный сервис, который предоставит публичный API-адрес. Иными словами, работа и функционал данной программы ограничены только потребностями разработчика и средствами, которые разработчик готов предоставить для работы сервиса. Говоря о технической части данной программы, можно констатировать, что человек со средним навыком программирования может без лишних сложностей освоить и применить ее в той области, где может потребоваться непрерывное получение информации от исследуемого объекта.

Разобрана архитектура запросов и ответов, показана работа всех переменных, которые есть в программе, а также прочие детали ее работы. Такая программа с полноценным функционалом поможет начинающему исследователю выполнить проект без значительных усилий. Данная программа отображения информации может быть использована в качестве коммерческой, так как она рассчитана сразу на несколько пользователей, т. е. программа-сервер может отвечать сразу на несколько запросов.

## Информация об авторе

**Смирнов Никита Артемович** – аспирант кафедры робототехники и автоматизации производственных систем СПбГЭТУ «ЛЭТИ».
E-mail: smirnov-n1@mail.ru

## Information about the author

**Nikita A. Smirnov** – postgraduate student of the Department of Robotics and Automation of Production Systems, Saint Petersburg Electrotechnical University.
E-mail: smirnov-n1@mail.ru

Статья поступила в редакцию 17.02.2023; принята к публикации после рецензирования 23.02.2023; опубликована онлайн 25.04.2023.

Submitted 17.02.2023; accepted 23.02.2023; published online 25.04.2023.