Вариант 10

**ToDo-лист** — это список задач, которые нужно выполнить. Он помогает планировать дела, ставить приоритеты и отслеживать выполнение. Обычно включает пункты с кратким описанием, отметкой о статусе (выполнено/не выполнено) и, при необходимости, сроками. Используется как в повседневной жизни, так и в работе для повышения продуктивности.

Глава 1. Аналоги

## **Todoist**

### Технологии:

* **Frontend:** React.js, TypeScript
* **Backend:** Python (Django), PostgreSQL
* **Sync:** WebSockets, Push API
* **Клиенты:** Electron (для десктопа), Swift (iOS), Kotlin (Android)

### Причины использования:

* **React + TypeScript** обеспечивают быстрый интерфейс с типовой безопасностью и хорошей масштабируемостью.
* **Django + PostgreSQL** дают надёжную архитектуру и мощные ORM-возможности. Удобно масштабировать, писать API, работать с правами доступа.
* **WebSockets** позволяют реализовать мгновенную синхронизацию между устройствами.
* **Electron** позволяет поддерживать единую кодовую базу на Windows, macOS и Linux.

### Удобства для пользователя:

* Все изменения задач мгновенно отображаются на всех устройствах.
* Интерфейс работает стабильно, даже при большом количестве задач и проектов.
* Надёжное хранение данных, даже если офлайн — синхронизируется позже.

## **Microsoft To Do**

### Технологии:

* **Frontend:** UWP (Windows), React Native (мобильные)
* **Backend:** Azure Cloud, C# (.NET Core)
* **Sync:** Microsoft Graph API
* **Storage:** Microsoft Exchange / Outlook Tasks (через облако)

### Причины использования:

* **UWP и React Native** позволяют использовать нативные UI-компоненты и хорошо интегрироваться в экосистему Microsoft.
* **Microsoft Graph API** — единый API для доступа к данным Outlook, Calendar, OneDrive и т.д.
* **Azure** обеспечивает масштабируемую облачную платформу и высокую отказоустойчивость.

### Удобства для пользователя:

* Сильная интеграция с Outlook, Microsoft 365 и Windows.
* Уведомления и синхронизация работают "из коробки" на любом устройстве.
* Простой интерфейс без перегрузки лишними функциями.

**Trello**

**Технологии:**

* **Frontend:** React.js, TypeScript (переход с Backbone.js)
* **Backend:** Node.js, MongoDB
* **Sync:** WebSockets (Socket.IO), REST API
* **Клиенты:** Web (SPA), React Native (мобильные приложения)

**Причины использования:**

* **React + TypeScript** обеспечивают современный, модульный и безопасный интерфейс. Проект начал как Backbone-приложение, но постепенно был переведён на React для улучшения производительности и читаемости кода.
* **Node.js + MongoDB** предоставляют гибкую, слабо типизированную архитектуру, идеально подходящую для хранения и манипулирования иерархическими структурами: доска → список → карточка → действия.
* **WebSockets через Socket.IO** позволяют обновлять доски в реальном времени для всех участников без необходимости ручного обновления страницы.
* **React Native** упрощает кроссплатформенную разработку мобильных приложений, обеспечивая единый подход для iOS и Android.

**Удобства для пользователя:**

* **Мгновенная синхронизация** всех изменений на доске между участниками, независимо от платформы.
* **Гибкость работы с задачами** — интерфейс позволяет быстро перемещать карточки, добавлять чек-листы, вложения, метки и комментарии.
* **Надёжная работа в браузере и на мобильных устройствах**, включая офлайн-режим с последующей синхронизацией изменений.
* **Производительность масштабируется** при увеличении количества досок и участников, благодаря событийной модели и асинхронной архитектуре сервера.

**Глава 2. Техническое задание**

**Проект: Веб-приложение ToDo-лист  
Цель: Создать простое, удобное приложение для создания и управления списком задач с возможностью базовой фильтрации и сохранением данных на сервере.**

### **1. Функциональные требования**

* Создание, редактирование и удаление задач
* Отметка задач как выполненных/активных
* Просмотр списка задач с возможностью фильтрации (все, активные, выполненные)
* Простая страница с формой и списком задач
* Возможность сохранять данные между сессиями (через базу SQLite)

### **2. Нефункциональные требования**

* Веб-интерфейс с минимальным дизайном (HTML, CSS, базовый JavaScript)
* Быстрая загрузка страницы и обновление данных без перезагрузки (использовать AJAX)

### **3. Архитектура и технологии**

* **Backend:** Python 3.x + Flask
* **База данных:** SQLite — лёгкая встроенная база, не требует отдельного сервера
* **Frontend:** HTML + CSS + JavaScript
* **Шаблонизация:** Jinja2 (если Flask) для рендеринга страниц
* **API:** REST API для операций с задачами (CRUD)
* **Хостинг:** Локальный сервер, возможность запуска на любом сервере с Python

### **4. Основные API-эндпоинты (пример для REST)**

* GET /tasks — получить список задач
* POST /tasks — создать новую задачу
* PUT /tasks/<id> — обновить задачу (текст, статус)
* DELETE /tasks/<id> — удалить задачу

### **5. Требования к интерфейсу**

* Главная страница с формой для добавления задачи
* Список задач с кнопками “выполнить”, “редактировать”, “удалить”
* Фильтры для просмотра: все / активные / выполненные
* Отмеченные задачи должны визуально отличаться (например, зачёркнутый текст)

### **6. Тестирование**

* Тестирование основных CRUD операций через API
* Тестирование UI на корректность отображения и обновления данных
* Проверка работы приложения после перезапуска сервера (данные сохраняются)

### **7. Возможные расширения (факультативно)**

* Минимальная аутентификация пользователей
* Добавление сроков и напоминаний
* Сортировка задач по дате создания или приоритету
* Экспорт списка задач в формате JSON или CSV

Глава 3. Реализация проекта

## **Описание проекта: Веб-приложение "ToDo-лист"**

### **Общее описание**

Проект представляет собой простое, одностраничное веб-приложение для управления списком задач. Пользователь может добавлять, редактировать, выполнять и удалять задачи, а также фильтровать их по статусу (все, активные, выполненные). Все данные сохраняются в локальной базе данных SQLite и сохраняются между сессиями.

### **Цель проекта**

Создать лёгкое, функциональное и понятное веб-приложение с минимальным стеком технологий, ориентированное на учебную практику, демонстрацию владения Flask, HTML/CSS и работы с базами данных.

### **Функциональность**

#### Основные возможности:

* Добавление новой задачи
* Отображение всех задач
* Редактирование текста задачи
* Удаление задачи
* Переключение статуса "выполнено/не выполнено"
* Фильтрация задач:
  + Все задачи
  + Только активные
  + Только выполненные
* Сортировка задач по дате создания (новые сверху)
* Отображение даты создания задачи

Используемые технологии приведены в таблице 1

| Компонент | Технология | Обоснование выбора |
| --- | --- | --- |
| Язык | Python | Лёгкий, читаемый и широко применяемый |
| Фреймворк | Flask | Минималистичный и удобный для небольших проектов |
| База данных | SQLite | Не требует отдельного сервера, подходит для небольших проектов |
| HTML/CSS | Jinja2-шаблоны + собственный CSS | Простота, удобство стилизации |
| JavaScript | Только базовый (встроенный в HTML) | Нет лишних зависимостей |

Таблица 1 — используемые технологии

### **Архитектура проекта**

* **Модель:** Таблица tasks в SQLite:
  + id — идентификатор
  + title — текст задачи
  + done — статус выполнения (0/1)
  + created\_at — дата создания в формате ISO
* **Контроллеры (Flask routes):**
  + / — главная страница со списком задач
  + /add — добавление задачи (POST)
  + /toggle/<id> — смена статуса задачи
  + /edit/<id> — редактирование задачи (POST)
  + /delete/<id> — удаление задачи
* **Представление:** index.html с отображением задач, формами и фильтрами

app.py

from flask import Flask, render\_template, request, redirect, url\_for

import sqlite3

from datetime import datetime

app = Flask(\_\_name\_\_)

DATABASE = 'todo.db'

def get\_db\_connection():

conn = sqlite3.connect(DATABASE)

conn.row\_factory = sqlite3.Row

return conn

def init\_db():

conn = get\_db\_connection()

conn.execute('''

CREATE TABLE IF NOT EXISTS tasks (

id INTEGER PRIMARY KEY AUTOINCREMENT,

title TEXT NOT NULL,

done BOOLEAN NOT NULL DEFAULT 0,

created\_at TEXT NOT NULL

)

''')

conn.commit()

conn.close()

@app.route('/')

def index():

filter\_status = request.args.get('filter', 'all')

conn = get\_db\_connection()

if filter\_status == 'active':

tasks = conn.execute('SELECT \* FROM tasks WHERE done = 0 ORDER BY created\_at DESC').fetchall()

elif filter\_status == 'done':

tasks = conn.execute('SELECT \* FROM tasks WHERE done = 1 ORDER BY created\_at DESC').fetchall()

else:

tasks = conn.execute('SELECT \* FROM tasks ORDER BY created\_at DESC').fetchall()

conn.close()

return render\_template('index.html', tasks=tasks, filter\_status=filter\_status)

@app.route('/add', methods=['POST'])

def add():

title = request.form['title'].strip()

if title:

conn = get\_db\_connection()

conn.execute(

'INSERT INTO tasks (title, done, created\_at) VALUES (?, ?, ?)',

(title, False, datetime.now().isoformat())

)

conn.commit()

conn.close()

return redirect(url\_for('index'))

@app.route('/toggle/<int:task\_id>')

def toggle(task\_id):

conn = get\_db\_connection()

task = conn.execute('SELECT done FROM tasks WHERE id = ?', (task\_id,)).fetchone()

if task:

new\_status = 0 if task['done'] else 1

conn.execute('UPDATE tasks SET done = ? WHERE id = ?', (new\_status, task\_id))

conn.commit()

conn.close()

return redirect(url\_for('index'))

@app.route('/delete/<int:task\_id>')

def delete(task\_id):

conn = get\_db\_connection()

conn.execute('DELETE FROM tasks WHERE id = ?', (task\_id,))

conn.commit()

conn.close()

return redirect(url\_for('index'))

@app.route('/edit/<int:task\_id>', methods=['POST'])

def edit(task\_id):

new\_title = request.form.get('new\_title', '').strip()

if new\_title:

conn = get\_db\_connection()

conn.execute('UPDATE tasks SET title = ? WHERE id = ?', (new\_title, task\_id))

conn.commit()

conn.close()

return redirect(url\_for('index'))

if \_\_name\_\_ == '\_\_main\_\_':

init\_db()

app.run(debug=True)

index.html

<!DOCTYPE html>

<html lang="ru">

<head>

<meta charset="UTF-8" />

<title>ToDo-лист</title>

<link rel="stylesheet" href="{{ url\_for('static', filename='style.css') }}">

</head>

<body>

<div class="container">

<h1>ToDo-лист</h1>

<form action="{{ url\_for('add') }}" method="post">

<input type="text" name="title" placeholder="Новая задача" required />

<button type="submit">Добавить</button>

</form>

<div class="filters">

<a href="/?filter=all" class="{{ 'active' if filter\_status == 'all' else '' }}">Все</a>

<a href="/?filter=active" class="{{ 'active' if filter\_status == 'active' else '' }}">Активные</a>

<a href="/?filter=done" class="{{ 'active' if filter\_status == 'done' else '' }}">Выполненные</a>

</div>

<ul>

{% for task in tasks %}

<li class="{{ 'done' if task.done else '' }}">

<div class="task-block">

<a href="{{ url\_for('toggle', task\_id=task.id) }}">

{% if task.done %}✔{% else %}○{% endif %}

</a>

<form action="{{ url\_for('edit', task\_id=task.id) }}" method="post" class="edit-form">

<input type="text" name="new\_title" value="{{ task.title }}" class="edit-input" />

<button type="submit">Сохранить</button>

</form>

<small class="timestamp">{{ task.created\_at.split('T')[0] }}</small>

<a href="{{ url\_for('delete', task\_id=task.id) }}" class="delete">✖</a>

</div>

</li>

{% endfor %}

</ul>

</div>

</body>

</html>

style.css

body {

font-family: Arial, sans-serif;

background: #f9f9f9;

padding: 20px;

}

.container {

max-width: 500px;

margin: auto;

background: white;

padding: 20px;

box-shadow: 0 0 8px rgba(0,0,0,0.1);

}

input[type="text"] {

width: 75%;

padding: 10px;

margin-right: 10px;

border: 1px solid #ddd;

border-radius: 4px;

}

button {

padding: 10px 15px;

border: none;

background: #28a745;

color: white;

border-radius: 4px;

cursor: pointer;

}

button:hover {

background: #218838;

}

ul {

list-style-type: none;

padding-left: 0;

margin-top: 20px;

}

li {

padding: 8px 0;

border-bottom: 1px solid #eee;

display: flex;

align-items: center;

justify-content: space-between;

}

li.done {

color: #999;

text-decoration: line-through;

}

li a {

text-decoration: none;

color: #007bff;

margin-right: 10px;

}

li a.delete {

color: #dc3545;

font-weight: bold;

}

.filters {

margin-top: 10px;

}

.filters a {

margin-right: 10px;

text-decoration: none;

color: #007bff;

}

.filters .active {

font-weight: bold;

text-decoration: underline;

}

.task-block {

display: flex;

align-items: center;

gap: 10px;

}

.edit-form {

flex-grow: 1;

display: flex;

gap: 5px;

}

.edit-input {

width: 100%;

border: 1px solid #ccc;

padding: 5px;

border-radius: 3px;

}

.timestamp {

font-size: 0.75em;

color: #888;

}

На рисунке 1 приведена структура проекта

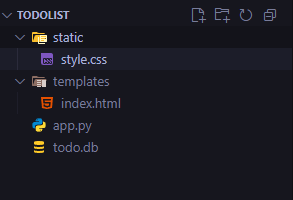


Рисунок 1 — структура проекта

На рисунке 2 показан интерфейс приложения

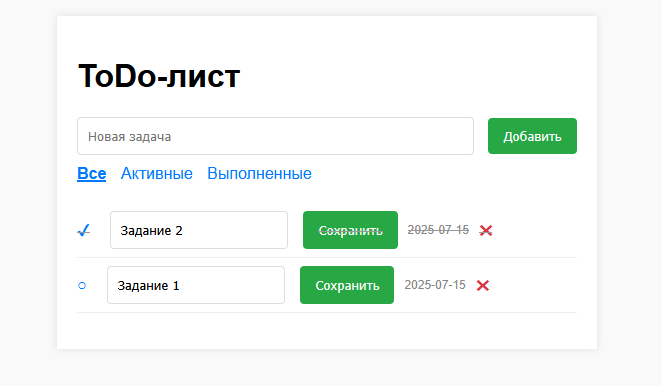


Рисунок 2 — интерфейс приложения

**Вывод**

Разработка ToDo-приложения с использованием Flask и SQLite продемонстрировала, что даже с минимальным стеком технологий можно реализовать устойчивое, интерактивное и удобное веб-приложение.

Проект охватывает полный цикл базового CRUD-функционала:

* пользователь может добавлять, редактировать, отмечать и удалять задачи;
* реализована фильтрация задач по статусу;
* данные хранятся в локальной БД и сохраняются между сессиями.

Список литературы

1. Лутц М. Изучаем Python. 5-е изд. / М. Лутц. — СПб.: Питер, 2019. — 1216 с. — (Серия «Бестселлеры O’Reilly»).
2. Зед А. Шоу. Изучаем Python. / З. А. Шоу. — М.: Эксмо, 2020. — 400 с.
3. Бейтс Д., Доусон Д. Python. Интенсивный курс / Д. Бейтс, Д. Доусон. — М.: Эксмо, 2022. — 640 с.