**Grocery To-Do (Shopping List) Application**

*Report*

Individual Assignment 1 – Software Development and DevOps

Student Name: Sofia Boicenco  
Course: Software Development and DevOps  
Date: 04.10.2025

**Introduction**

This Grocery List App is an application that was created to allow users to keep track of their shopping list daily, through a simple web interface. The app allows you to sort products by categories, as to when someone is in a specific aisle of the supermarket, they can easily see everything they need to buy from that corner of the store, without having to go through the list multiple times.

It was developed using Python (FastAPI) for the backend, SQLite for data storage and HTML/JavaScript for the frontend. The app supports CRUD operations and it is designed for everyone that does their own groceries, from families with numerous members to individuals that need something to help them stay organized and remember what they need to buy. This project is a simple local application that can be later adapted into a scalable DevOps system.

**SDLC Model and Methodology**

This project followed a simplified SDLC Agile model with clear sequential phases that underwent minor changes for improvements. This model was selected because it allows for iterative development and flexibility throughout the project. Given the app's small scope, Agile made it possible to add features such as a web-based UI and categories for items without disrupting the timeline of the project. Short development cycles and regular testing made it easier to refine both the backend as well as the frontend.

The project’s scope was to design and implement a minimal grocery list manager that allows users to add, view, update, and delete items while maintaining local data persistence. The main SMART goals include delivering a fully functional prototype with two core features, building it with lightweight technologies (FastAPI, SQLite, HTML/JS), and completing the development and documentation until the designated deadline (October 5th).

The following summarizes the main points of each SDLC phase developed throughout the project:

* **Planning:** Defined the project scope, target users, and SMART goals.
* **Requirements:** Identified functional and non-functional requirements, focusing on CRUD features and usability.
* **Design:** Developed the architecture connecting FastAPI to SQLite and created a friendly UI with categorized lists.
* **Implementation:** Built the backend API and frontend, tested endpoints using SwaggerUI, and verified data persistence in SQLite.

This approach allowed flexibility to make adjustments during development, such as refining the UI and improving the database design.

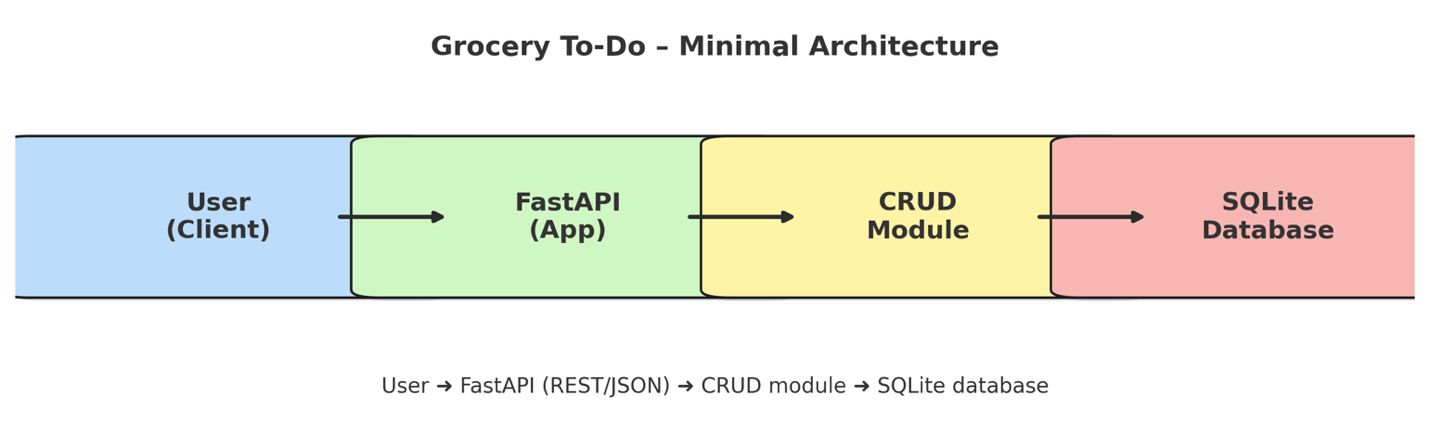
Testing was conducted through FastAPI’s integrated Swagger UI, where each endpoint (Create, Read, Update, Delete) was checked to function correctly. This interactive interface allowed efficient debugging and validation before integrating the frontend.

The separate Planning, SRS, and Design documents can be found inside the *docs* folder of the project repository.

Stakeholder analysis and detailed project goals are also included in the Planning document within the /docs folder.

**System Architecture and Design**

**The overview of the system architecture and design of the Grocery List App describes the main software components, their interactions and how data flows through the system. The architecture was designed to be simple, modular, and easily adaptable for future DevOps deployment.**



**The overall architecture of the system is shown in the figure above. It illustrates how user interactions through the frontend communicate with the backend FastAPI app, which processes requests via the CRUD module and stores data in the SQLite database.**

**Backend/ Application Layer**

**The backend was implemented with FastAPI and handles the app's logic and provides the RESTful endpoints, that each correspond to a CRUD operation such as adding or deleting an item. Swagger UI was automatically generated by FastAPI for testing.**

The backend was built using the **Model-View-Controller (MVC)** pattern.

* models.py defines the data schemas
* crud.py handles Create, Read, Update, and Delete operations with SQLite
* db.py manages the connection and table setup
* main.py contains the FastAPI app and routes  
  The app was tested using **Swagger UI** at http://127.0.0.1:8000/docs.

**Database Layer**

**For persistent storage, a lightweight relational database SQLite was used. It stores all grocery items locally, including their name, quantity, category, and purchase status. The database connection and setup are handled in a separate *db.py* module, ensuring a clean separation between data and logic.**

**Frontend Layer**

**The frontend was created to be a responsive and visually appealing user interface developed using HTML, CSS and JavaScript. It allows users to add, update, delete, and mark items as purchased, while displaying all grocery items in different sections by category. The interface communicates with the backend through HTTP requests using the Fetch API.**

**The application follows a modular folder structure that supports scalability by separating the backend from the frontend and documentation, allowing future integration.**

**grocery\_list/**

**│── app/**

**│ ├── main.py # FastAPI entry point**

**│ ├── models.py # Data schema definitions**

**│ ├── crud.py # Database operations (CRUD logic)**

**│ ├── db.py # SQLite connection setup**

**│── frontend/ # HTML, CSS, and JavaScript files**

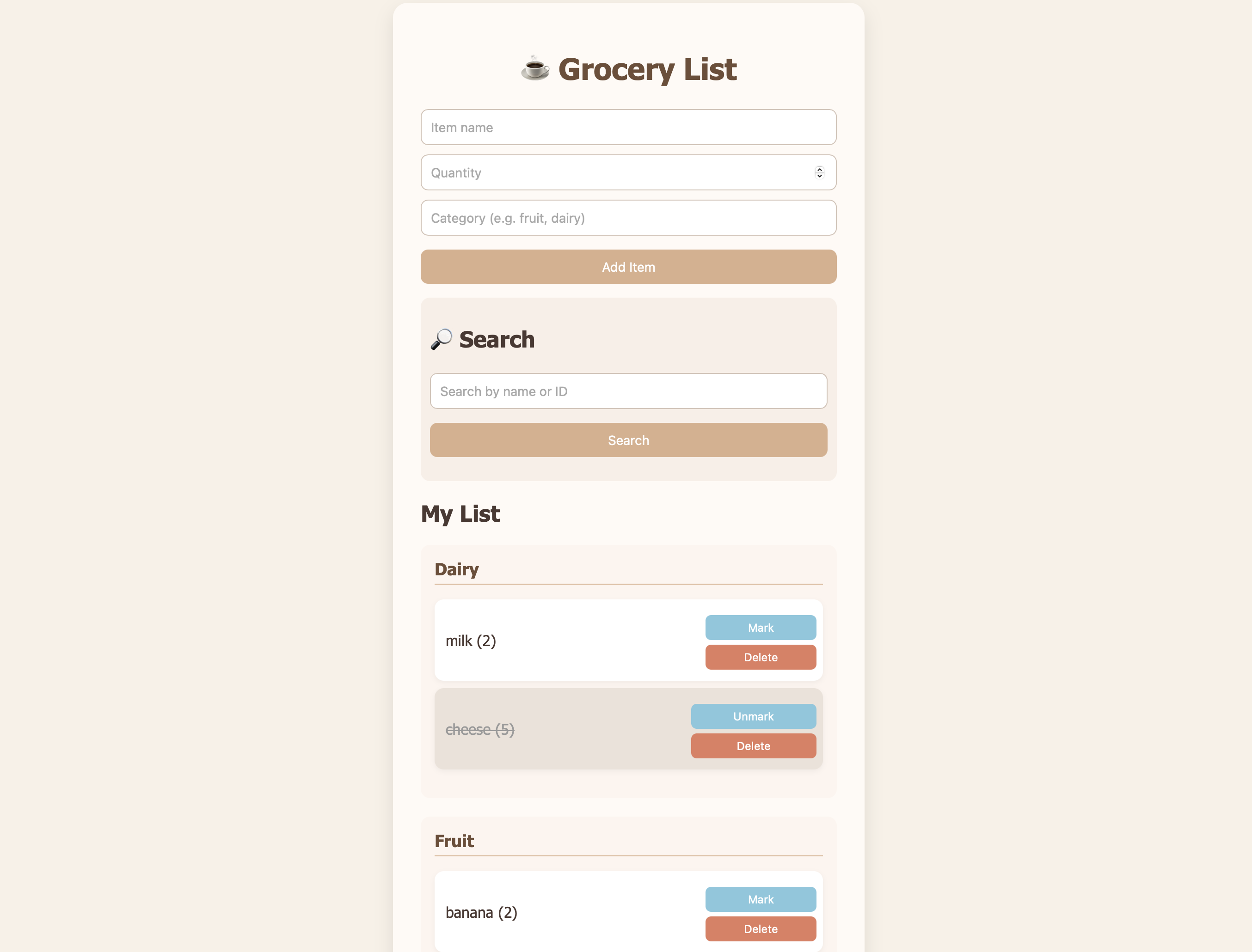
**│── docs/ # SDLC documentation**

**│── requirements.txt # Dependencies**

**│── README.md # Setup guide**

This design emphasizes modularity, readability and maintainability. Because everything is divided in its own separate file (database, models, CRUD logic and frontend), the app can be easily modified and it makes testing simpler as well. The system is structured in a way that allows for future DevOps practices, such as automated builds and deployments.

The UI was designed with a pastel colour palette for a calm and intuitive atmosphere and a search bar and being able to separate items by categories were added to enhance user experience.



**Reflection and DevOps Integration**

Although this project was designed as a minimal grocery list application, its architecture can be easily scaled. The backend could integrate a user authentication system, allowing multiple users to have their own personalized lists. Additionally, the SQLite database could be replaced with a cloud-based one such as Azure SQL to support higher volumes of data and concurrent users.

From a DevOps perspective, this application could be containerized using Docker, enabling easy deployment across different environments. Continuous Integration (CI) and Continuous Deployment (CD) pipelines could be implemented using GitHub or Azure DevOps to automate testing and deployment. This would allow for faster iterations and better software reliability.

Automated testing using pytest could be added to verify API functionality with each commit. For monitoring, lightweight tools like Prometheus or simple logging could track system performance and user interactions.

Overall, this project shows that even a small prototype can be expanded into a scalable and maintainable application aligned with DevOps principles.

**AI Usage Disclosure**

AI (ChatGPT) was used to resolve specific coding errors, clarify documentation structure, and assist in designing the **frontend interface. The frontend (HTML, CSS, and JavaScript) was created using AI that was used to generate, refine, and debug the complete frontend code to visually present the grocery list app in a user-friendly, pastel-themed interface.**

Below are a few examples of the prompts used for AI during development:

**Frontend Creation** - **Prompt:**

“Write a simple but pretty HTML/CSS/JavaScript interface for my FastAPI grocery list app. I want pastel colors and a cappuccino-style design.”

**AI Response (simplified):**

Produced a fully functional HTML layout with input fields, buttons for adding, deleting, and marking items as purchased, and styled the page using soft brown and cream tones for a warm, minimal aesthetic.

**Improving Interactivity** - **Prompt:**

“Add a search bar and tell me how to add an animation or a cute grocery-themed icon at the bottom of the page to make the application more fun.”

**AI Response (simplified):**

Implemented a search feature and button animations using simple CSS transitions and JavaScript filtering logic. (The animation that I wanted at the bottom of the page didn't work out in the end.)

**Connecting Frontend to Backend** - **Prompt:**

“Make this HTML page connect to my FastAPI backend endpoints using JavaScript fetch requests.”

**AI Response (simplified):**

Wrote the JavaScript logic to perform GET, POST, PATCH, and DELETE requests to the FastAPI endpoints and display updates in real time within the browser.

**Appendix – Key Code Excerpts**

app/main.py

This section includes the complete implementation of the FastAPI application - the API routes, how it integrates the database operations and connects all CRUD functionalities.

from typing import Optional, List

from fastapi import FastAPI, HTTPException, Query

from fastapi.middleware.cors import CORSMiddleware

from .db import init\_db

from .models import ItemCreate, ItemUpdate, ItemRead

from . import crud

app = FastAPI(title="Grocery List API", version="1.0.0")

# allow the frontend to call the API during development

app.add\_middleware(

CORSMiddleware,

allow\_origins=["\*"], # only for local development

allow\_credentials=True,

allow\_methods=["\*"],

allow\_headers=["\*"],

)

@app.on\_event("startup")

def \_startup():

init\_db()

@app.post("/items", response\_model=ItemRead, status\_code=201)

def create\_item(item: ItemCreate):

return crud.create\_item(item.model\_dump())

@app.get("/items", response\_model=List[ItemRead])

def list\_items(purchased: Optional[bool] = Query(default=None)):

return crud.list\_items(purchased)

@app.get("/items/{item\_id}", response\_model=ItemRead)

def read\_item(item\_id: int):

item = crud.get\_item(item\_id)

if not item:

raise HTTPException(status\_code=404, detail="Item not found")

return item

@app.patch("/items/{item\_id}", response\_model=ItemRead)

def patch\_item(item\_id: int, patch: ItemUpdate):

# Basic guard: at least one field provided

if patch.model\_dump(exclude\_unset=True) == {}:

raise HTTPException(status\_code=422, detail="No fields to update")

item = crud.update\_item(item\_id, patch.model\_dump(exclude\_unset=True))

if not item:

raise HTTPException(status\_code=404, detail="Item not found")

return item

@app.patch("/items/{item\_id}/toggle", response\_model=ItemRead)

def toggle\_item(item\_id: int):

item = crud.toggle\_item(item\_id)

if not item:

raise HTTPException(status\_code=404, detail="Item not found")

return item

@app.delete("/items/{item\_id}", status\_code=204)

def delete\_item(item\_id: int):

ok = crud.delete\_item(item\_id)

if not ok:

raise HTTPException(status\_code=404, detail="Item not found")

app/crud.py

This file handles all database interactions (Create, Read, Update, Delete, and Toggle operations).

from typing import Optional, Dict, Any, List

from .db import get\_conn

def \_row\_to\_dict(row) -> Dict[str, Any]:

return {

"id": row["id"],

"name": row["name"],

"quantity": row["quantity"],

"category": row["category"],

"purchased": bool(row["purchased"]),

}

def create\_item(data: Dict[str, Any]) -> Dict[str, Any]:

with get\_conn() as conn:

cur = conn.cursor()

cur.execute(

"INSERT INTO items (name, quantity, category, purchased) VALUES (?,?,?,?)",

(data["name"], data.get("quantity", 1), data.get("category", ""), int(data.get("purchased", False))),

)

item\_id = cur.lastrowid

row = cur.execute("SELECT \* FROM items WHERE id=?", (item\_id,)).fetchone()

return \_row\_to\_dict(row)

def list\_items(purchased: Optional[bool] = None) -> List[Dict[str, Any]]:

with get\_conn() as conn:

cur = conn.cursor()

if purchased is None:

rows = cur.execute("SELECT \* FROM items ORDER BY id").fetchall()

else:

rows = cur.execute("SELECT \* FROM items WHERE purchased=? ORDER BY id", (int(purchased),)).fetchall()

return [\_row\_to\_dict(r) for r in rows]

def get\_item(item\_id: int) -> Optional[Dict[str, Any]]:

with get\_conn() as conn:

row = conn.execute("SELECT \* FROM items WHERE id=?", (item\_id,)).fetchone()

return \_row\_to\_dict(row) if row else None

def update\_item(item\_id: int, fields: Dict[str, Any]) -> Optional[Dict[str, Any]]:

if not fields:

return get\_item(item\_id)

sets, vals = [], []

if "name" in fields: sets.append("name=?"); vals.append(fields["name"])

if "quantity" in fields: sets.append("quantity=?"); vals.append(fields["quantity"])

if "category" in fields: sets.append("category=?"); vals.append(fields["category"])

if "purchased" in fields: sets.append("purchased=?"); vals.append(int(fields["purchased"]))

if not sets:

return get\_item(item\_id)

with get\_conn() as conn:

cur = conn.cursor()

vals.append(item\_id)

cur.execute(f"UPDATE items SET {', '.join(sets)} WHERE id=?", tuple(vals))

if cur.rowcount == 0:

return None

row = cur.execute("SELECT \* FROM items WHERE id=?", (item\_id,)).fetchone()

return \_row\_to\_dict(row)

def toggle\_item(item\_id: int) -> Optional[Dict[str, Any]]:

with get\_conn() as conn:

cur = conn.cursor()

row = cur.execute("SELECT purchased FROM items WHERE id=?", (item\_id,)).fetchone()

if not row:

return None

new\_val = 0 if row["purchased"] else 1

cur.execute("UPDATE items SET purchased=? WHERE id=?", (new\_val, item\_id))

row2 = cur.execute("SELECT \* FROM items WHERE id=?", (item\_id,)).fetchone()

return \_row\_to\_dict(row2)

def delete\_item(item\_id: int) -> bool:

with get\_conn() as conn:

cur = conn.cursor()

cur.execute("DELETE FROM items WHERE id=?", (item\_id,))

return cur.rowcount > 0