Store Manager

Popis programu: Tento program je aplikácia na správu obchodu (Store Manager). Jeho hlavnou úlohou je kompletná evidencia zamestnancov a skladových zásob (produktov) prostredníctvom webového rozhrania. Užívateľ môže cez prehliadač jednoducho pridávať nové záznamy, vyhľadávať ich podľa rôznych kritérií, upravovať existujúce údaje alebo ich mazať.

Používateľská príručka:

Po spustení aplikácie a otvorení stránky v prehliadači (http://localhost:8000) má používateľ plnú kontrolu nad správou obchodu. Rozhranie je rozdelené na dve hlavné sekcie: Zamestnanci a Produkty. Užívateľ má nasledovný prístup ku databáze:

1. Pridávanie a Úprava: V režime Add/Edit sa vpĺnajú formuláre pre nové záznamy. Ak vo vyhľadávaní kliknete na tlačidlo Edit pri konkrétnom zázname, formulár sa predvyplní existujúcimi údajmi, ktoré môžete zmeniť a uložiť.
2. Vyhľadávanie: V režime Search môžete filtrovať dáta podľa ľubovoľných kritérií. Tlačidlo Load all zobrazí kompletný zoznam.
3. Mazanie: Každý nájdený záznam obsahuje tlačidlo Delete, ktorým ho okamžite a nenávratne odstránite z databázy.

Práca so súbormi (Databáza a JSON) Aplikácia na pozadí automaticky spravuje dva kľúčové súbory, ktoré môže používateľ ovplyvniť:

store.db (Databáza): Toto je hlavný pracovný súbor, kde sa ukladajú všetky zmeny v reálnom čase. Ak tento súbor bude zmazaný, aplikácia si ho pri ďalšom štarte sama vytvorí. store_data.json (Záloha a Import): Tento súbor slúži ako čitateľná záloha.

Export: Pri korektnom vypnutí aplikácie (napr. cez Ctrl+C) sa všetky dáta z databázy automaticky uložia do tohto JSON súboru.

Import: Ak aplikácia pri štarte nenájde súbor store.db, automaticky načíta dáta práve z tohto JSON súboru.

Manuálny zásah: Používateľ môže tento súbor otvoriť v textovom editore a ručne doplniť alebo upraviť veľké množstvo dát (napr. hromadný import tovaru), ktoré sa pri najbližšom štarte nahrajú do systému.

Zoznam použitých knižníc:

1. tokio

Verzia: 1.48.0 (features: full)
Účel: Asynchrónny runtime pre Rust. Poháňa celú aplikáciu a umožňuje jej robiť viac vecí naraz (napr. čakať na databázu a zároveň počúvať na HTTP požiadavky).

Využitie v projekte:

- main.rs: Makro #[tokio::main] spúšťa hlavnú funkciu v asynchrónnom režime.
- server.rs: Používa sa tokio::net::TcpListener na vytvorenie sieťového spojenia.
- main.rs: tokio::spawn sa používa na spustenie servera v samostatnom vlákne a tokio::signal na čakanie na ukončenie programu (Ctrl+C).

2. axum

Verzia: 0.8.6
Účel: Webový framework na tvorbu HTTP serverov a REST API. Je postavený na tokio a hyper.

Využitie v projekte:

- V api.rs a server.rs: Definuje routovanie (Router), HTTP metódy (get, post, delete, put) a handlery (funkcie, ktoré spracujú požiadavku).
- Spracováva JSON dáta (Json<T>) a extrahuje parametre z URL (Path).
- Poskytuje State na zdieľanie pripojenia k databáze medzi jednotlivými požiadavkami.

3. sqlx

Verzia: 0.8.6 (features: sqlite, runtime-tokio-native-tls, chrono)
Účel: Asynchrónna knižnica na prácu s SQL databázami. Poskytuje bezpečnú komunikáciu s databázou a kontrolu typov.

Využitie v projekte:

- V db.rs: Spravuje pripojenie k SQLite databáze (SqlitePool).
- Vykonáva SQL príkazy: CREATE TABLE, INSERT, SELECT, UPDATE, DELETE.
- Mapuje riadky z databázy priamo do Rust štruktúr.

4. serde

Verzia: 1.0.228
Účel: Framework na serializáciu a deserializáciu dát (prevod dátových štruktúr do formátu pre uloženie/prenos a naopak).

Využitie v projekte:

- V structs.rs a db_filler.rs: Makrá #[derive(Serialize, Deserialize)] umožňujú automaticky prevádzať štruktúry Employee a Product do JSON formátu a späť.

5. serde_json

Verzia: 1.0.145
Účel: Konkrétna implementácia serde pre formát JSON.

Využitie v projekte:

- V db_filler.rs: Číta dáta zo súboru store_data.json (from_reader) a zapisuje ich tam (to_writer_pretty).

6. anyhow

Verzia: 1.0
Účel: Knižnica na jednoduchú správu chýb (Error Handling). Umožňuje používať typ Result<T> bez nutnosti definovať vlastné typy chýb pre každú funkciu.

Využitie v projekte:

- V celom projekte (main.rs, db.rs, server.rs): Používa sa ako návratový typ Result<()> alebo Result<Self>, čo zjednodušuje prácu s chybami, ktoré môžu nastať pri práci s diskami, sieťou alebo databázou.

7. chrono

Verzia: 0.4 (features: serde)
Účel: Knižnica na prácu s dátumom a časom.

Využitie v projekte:

- V structs.rs: Typ NaiveDate sa používa na ukladanie dátumov (napr. hire_date, date_added).
- Spolupracuje so sqlx (ukladanie dátumu do DB) a serde (formátovanie dátumu v JSONe).

Funkcionalita tried:

1. StoreDB (db.rs) Táto trieda slúži ako hlavný prístupový bod k databáze, pričom spravuje pripojenie k SQLite súboru a automaticky vytvára potrebné tabuľky pri prvom spustení. Obsahuje metódy na vykonávanie všetkých operácií s dátami, ako je pridávanie, čítanie, úprava a mazanie záznamov o zamestnancoch a produktoch.

2. Server (server.rs) Trieda Server je zodpovedná za inicializáciu a spustenie webového servera na definovanom sieťovom porte. Jej hlavnou úlohou je prepojiť databázovú vrstvu s API routerom a následne počúvať na prichádzajúce HTTP požiadavky od používateľov. Vďaka nej je aplikácia dostupná cez webový prehliadač.

3. DBFiller (db_filler.rs) DBFiller funguje ako pomocný nástroj na synchronizáciu dát medzi databázou a záložným JSON súborom. Pri štarte aplikácie dokáže načítať dáta zo súboru do prázdnej databázy a pri vypnutí zabezpečuje export aktuálneho stavu späť do súboru. Týmto spôsobom garantuje, že dáta ostanú zachované a čitateľné aj po reštarte systému.

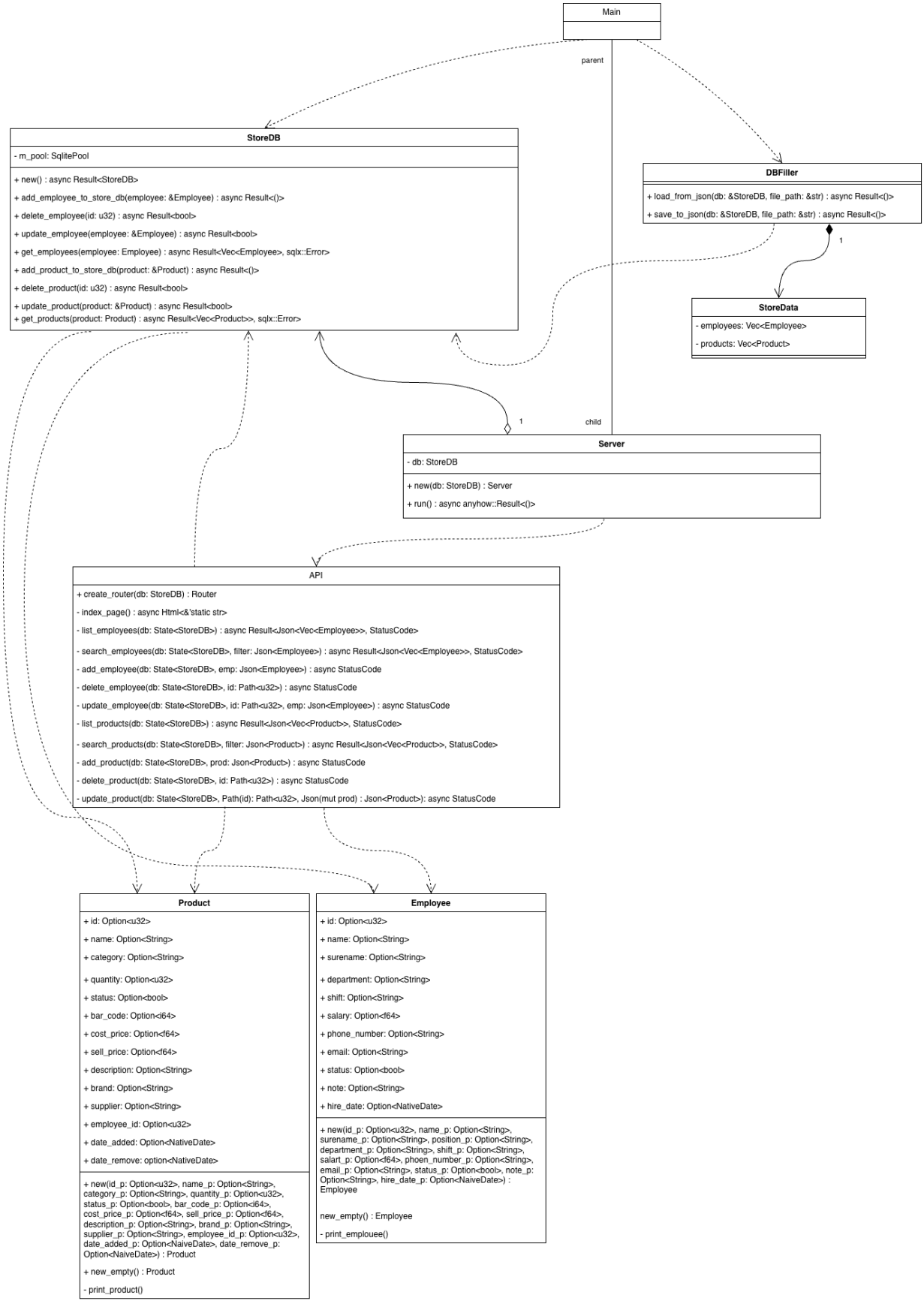
4. Employee (structs.rs) Táto dátová trieda reprezentuje konkrétneho zamestnanca v systéme a obsahuje všetky jeho atribúty, ako sú meno, pozícia atď. Slúži ako univerzálny prepravka na prenos informácií o zamestnancoch medzi databázou, logikou aplikácie a webovým rozhraním vo formáte JSON.

5. Product (structs.rs) Trieda Product definuje štruktúru tovaru v sklade a uchováva informácie ako názov, cena, množstvo či kategória. Podobne ako pri zamestnancoch, aj táto trieda slúži na manipuláciu s dátami o produktoch a ich prenos v rámci celého systému.

6. Modul API (api.rs) Tento modul definuje štruktúru webového rozhrania aplikácie a mapuje konkrétne URL adresy na funkcie, ktoré majú spracovať požiadavky. Obsahuje logiku pre prijímanie dát z prehliadača, volanie príslušných databázových operácií a odosielanie odpovedí späť používateľovi.

7. Modul Main (main.rs) Main predstavuje vstupný bod celého programu. Zodpovedá za inicializáciu databázy, načítanie zálohy, spustenie servera a nakoniec za bezpečné uloženie dát pri ukončení programu.

UML diagram:


Main
parent
StoreDB
- m_pool: SqlitePool
+ new() : async Result<StoreDB>
+ add_employee_to_store_db(employee: &Employee) : async Result<()>
+ delete_employee(id: u32) : async Result<bool>
+ update_employee(employee: &Employee) : async Result<bool>
+ get_employees(employee: Employee) : async Result<Vec<Employee>, sqlx::Error>
+ add_product_to_store_db(product: &Product) : async Result<()>
+ delete_product(id: u32) : async Result<bool>
+ update_product(product: &Product) : async Result<bool>
+ get_products(product: Product) : async Result<Vec<Product>>, sqlx::Error>
DBFiller
+ load_from_json(db: &StoreDB, file_path: &str) : async Result<()>
+ save_to_json(db: &StoreDB, file_path: &str) : async Result<()>
1
StoreData
- employees: Vec<Employee>
- products: Vec<Product>
1
child
Server
- db: StoreDB
+ new(db: StoreDB) : Server
+ run() : async anyhow::Result<()>
API
+ create_router(db: StoreDB) : Router
- index_page() : async Html<&'static str>
- list_employees(db: State<StoreDB>) : async Result<Json<Vec<Employee>>, StatusCode>
- search_employees(db: State<StoreDB>, filter: Json<Employee>) : async Result<Json<Vec<Employee>>, StatusCode>
- add_employee(db: State<StoreDB>, emp: Json<Employee>) : async StatusCode
- delete_employee(db: State<StoreDB>, id: Path<u32>) : async StatusCode
- update_employee(db: State<StoreDB>, id: Path<u32>, emp: Json<Employee>) : async StatusCode
- list_products(db: State<StoreDB>) : async Result<Json<Vec<Product>>, StatusCode>
- search_products(db: State<StoreDB>, filter: Json<Product>) : async Result<Json<Vec<Product>>, StatusCode>
- add_product(db: State<StoreDB>, prod: Json<Product>) : async StatusCode
- delete_product(db: State<StoreDB>, id: Path<u32>) : async StatusCode
- update_product(db: State<StoreDB>, Path(id): Path<u32>, Json(mut prod) : Json<Product>): async StatusCode
Product
+ id: Option<u32>
+ name: Option<String>
+ category: Option<String>
+ quantity: Option<u32>
+ status: Option<bool>
+ bar_code: Option<i64>
+ cost_price: Option<f64>
+ sell_price: Option<f64>
+ description: Option<String>
+ brand: Option<String>
+ supplier: Option<String>
+ employee_id: Option<u32>
+ date_added: Option<NativeDate>
+ date_remove: option<NativeDate>
+ new(id_p: Option<u32>, name_p: Option<String>, category_p: Option<String>, quantity_p: Option<u32>, status_p: Option<bool>, bar_code_p: Option<i64>, cost_price_p: Option<f64>, sell_price_p: Option<f64>, description_p: Option<String>, brand_p: Option<String>, supplier_p: Option<String>, employee_id_p: Option<u32>, date_added_p: Option<NaiveDate>, date_remove_p: Option<NaiveDate>) : Product
+ new_empty() : Product
- print_product()
Employee
+ id: Option<u32>
+ name: Option<String>
+ surename: Option<String>
+ department: Option<String>
+ shift: Option<String>
+ salary: Option<f64>
+ phone_number: Option<String>
+ email: Option<String>
+ status: Option<bool>
+ note: Option<String>
+ hire_date: Option<NativeDate>
+ new(id_p: Option<u32>, name_p: Option<String>, surename_p: Option<String>, position_p: Option<String>, department_p: Option<String>, shift_p: Option<String>, salart_p: Option<f64>, phoen_number_p: Option<String>, email_p: Option<String>, status_p: Option<bool>, note_p: Option<String>, hire_date_p: Option<NaiveDate>) : Employee
new_empty() : Employee
- print_emplouee()