Project Report: Building a Task Management Application with Go Backend and React Frontend

Daniyar Anuar

Table of Contents

Executive Summary

The main goal of this project was to create a full-stack task management application that allows users to easily create, view, update, and delete tasks. The backend was developed using Go (Golang), while the frontend was built with React. The project explores several key aspects of software development, including object-oriented programming in Go, database integration using GORM, and the creation of APIs. The frontend and backend communicate smoothly through RESTful API endpoints, with data exchanged in JSON format. Additionally, the system includes routing for seamless page transitions and a responsive design to ensure an optimal user experience.

Introduction

Task management apps play an important role in both personal and work life. They help people stay organized, manage their priorities, and keep track of what needs to be done. For this project, our goal was to build a reliable and easy-to-use task management system. By using Go for the backend and React for the frontend, we were able to create an application that’s both fast and user-friendly. I chose Go and React because they are not only easy to work with but also highly scalable, making them popular choices in the development world.

Project Objectives

The main goals of this project:

- Build a full-stack application that can handle basic task management features like creating, reading, updating, and deleting (CRUD).
- Develop RESTful API endpoints to manage tasks efficiently.
- Create a user-friendly, responsive interface that is easy to navigate.
- Ensure smooth communication between the backend and frontend by using JSON for data exchange.
- Focus on performance and security while managing tasks effectively.

Introduction to Go

Go, or Golang, is a statically-typed, compiled programming language known for its simplicity and speed. With a rich standard library and built-in concurrency support, Go is a great choice for building web applications.

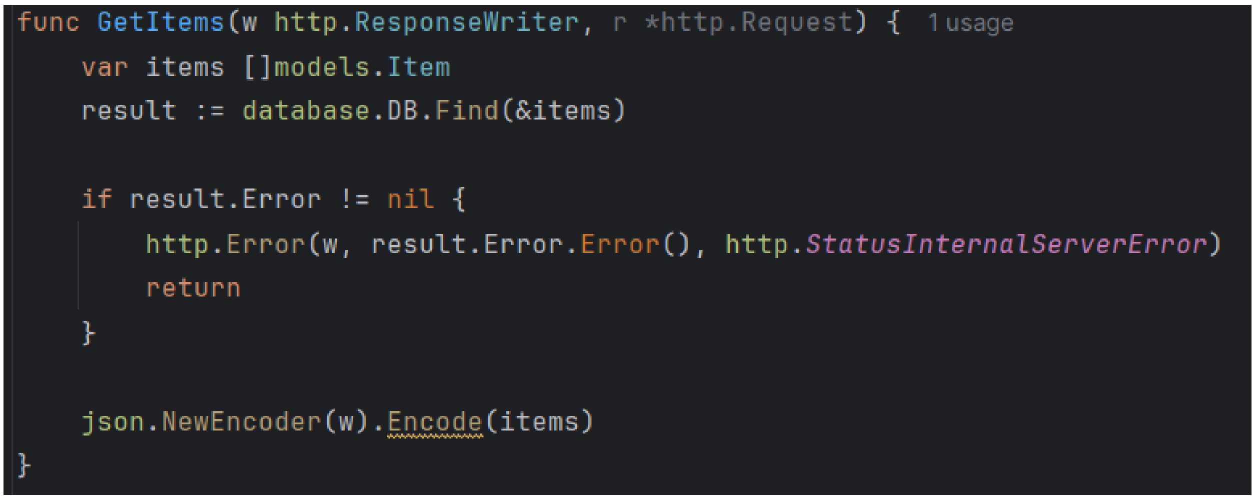
Development Environment: I started by setting up the Go environment, which includes the Go compiler and package manager. I used Go modules to handle dependencies, making the project more organized and easier to manage.

Syntax Basics: Go's syntax is clean and easy to understand:

- **Variables** are declared with var or the shorthand :=, making it quick to set up.
- **Control structures** like if, for, and switch work similarly to other languages like C, so they feel familiar.

- **Functions** in Go are powerful—they are first-class citizens and can return multiple values, making it easy to handle complex operations.

Example of a Go function used in the project:

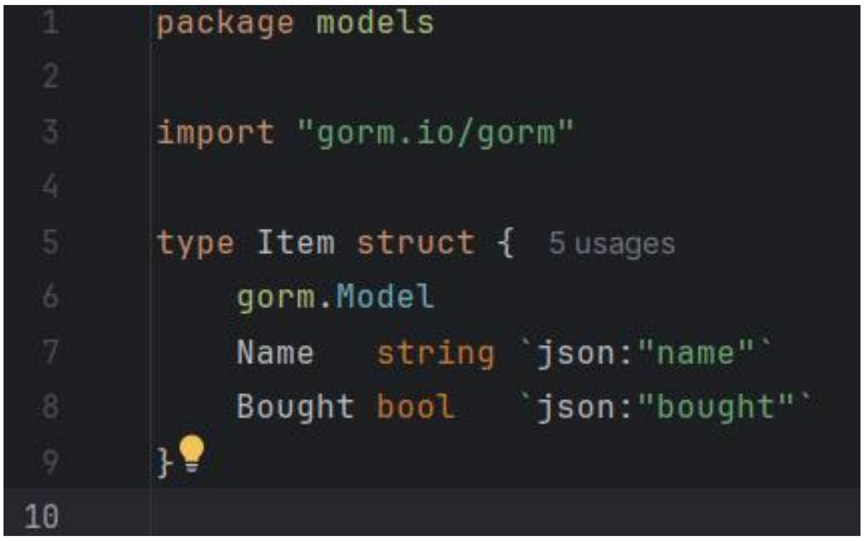


```go
func GetItems(w http.ResponseWriter, r *http.Request) {  1 usage
    var items []models.Item
    result := database.DB.Find(&items)

    if result.Error != nil {
        http.Error(w, result.Error.Error(), http.StatusInternalServerError)
        return
    }

    json.NewEncoder(w).Encode(items)
}
```

Object-Oriented Programming in Golang

Even though Go isn't a purely object-oriented language, it still supports many OOP concepts like encapsulation and methods. Instead of traditional classes, Go uses structs to define custom data types. You can also define methods associated with these structs, allowing for an object-like behavior.

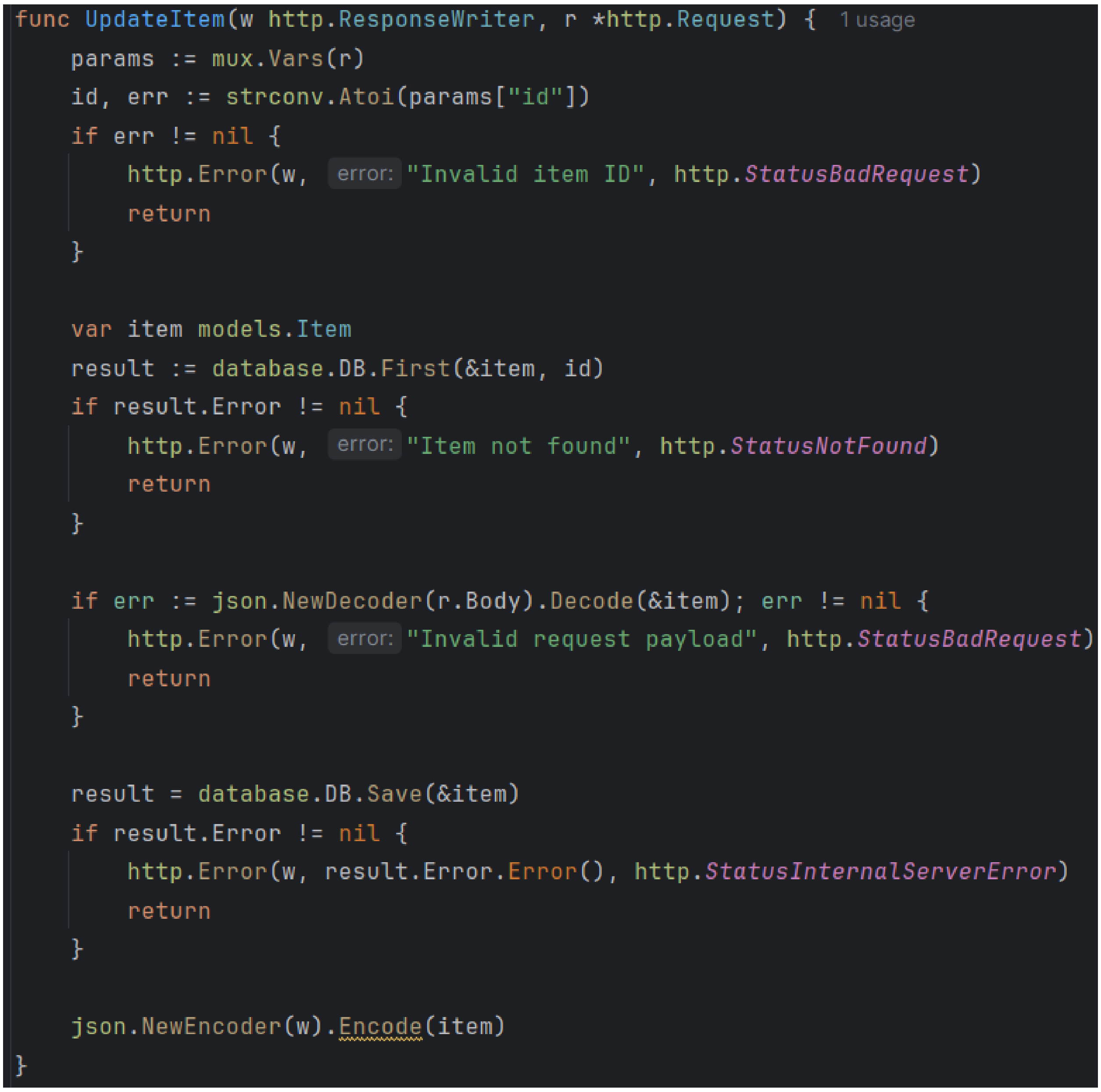
In my task management system, we defined a struct called "Item" to represent a task. Each task has fields for an ID, name, and whether it has been bought:



```go
package models

import "gorm.io/gorm"

type Item struct {  5 usages
    gorm.Model
    Name   string `json:"name"`
    Bought bool   `json:"bought"`
}
```

Here, Item is the core data structure of the application, encapsulating the details of each task. Go uses exported (starting with a capital letter) and unexported fields to control access to data, promoting encapsulation.

I also added methods to interact with the Item struct. For example, the method below allows us to update a task in the database:


```go
func UpdateItem(w http.ResponseWriter, r *http.Request) {  1 usage
    params := mux.Vars(r)
    id, err := strconv.Atoi(params["id"])
    if err != nil {
        http.Error(w,  error: "Invalid item ID", http.StatusBadRequest)
        return
    }

    var item models.Item
    result := database.DB.First(&item, id)
    if result.Error != nil {
        http.Error(w,  error: "Item not found", http.StatusNotFound)
        return
    }

    if err := json.NewDecoder(r.Body).Decode(&item); err != nil {
        http.Error(w,  error: "Invalid request payload", http.StatusBadRequest)
        return
    }

    result = database.DB.Save(&item)
    if result.Error != nil {
        http.Error(w, result.Error.Error(), http.StatusInternalServerError)
        return
    }

    json.NewEncoder(w).Encode(item)
}
```

This method handles task updates by retrieving an Item based on its ID, updating its fields, and saving it back to the database. The struct, combined with methods, allows for a clean and structured approach to managing tasks, which reflects the core principles of object-oriented programming in Go.

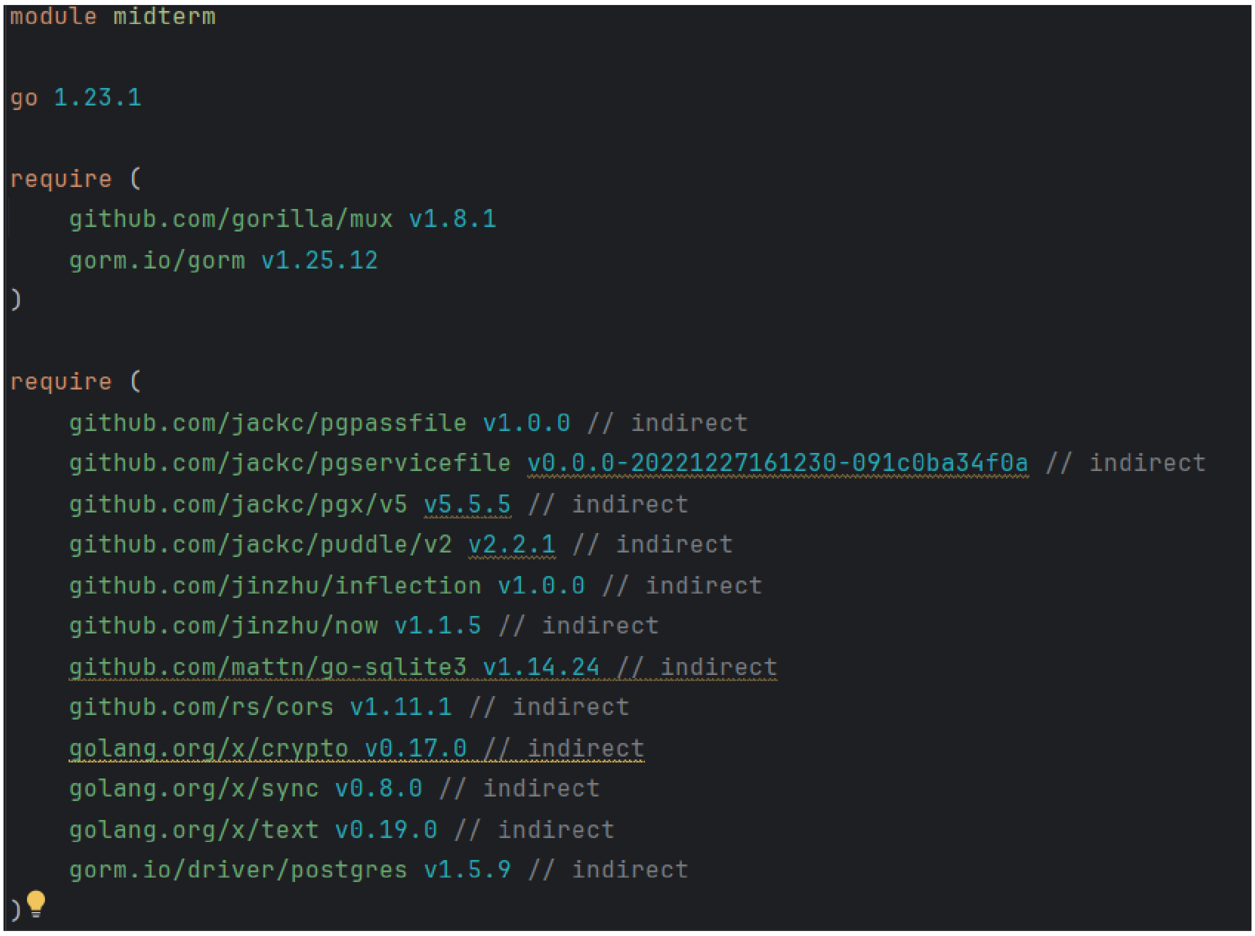
Dependency Management

Go modules (go.mod and go.sum) were used for managing dependencies in this project. By using Go modules, we ensured that all external libraries such as Gorilla Mux for routing and GORM for ORM were properly versioned and easily installed.

Project Structure: The project was organized into the following key directories:

- **controllers/**: Contains the logic for handling HTTP requests and responses.
- **database/**: Manages the database connection and initialization.
- **models/**: Defines the data structures and models used in the application.

The following command I used to initialize the Go module system:



```
module midterm

go 1.23.1

require (
    github.com/gorilla/mux v1.8.1
    gorm.io/gorm v1.25.12
)

require (
    github.com/jackc/pgpassfile v1.0.0 // indirect
    github.com/jackc/pgservicefile v0.0.0-20221227161230-091c0ba34f0a // indirect
    github.com/jackc/pgx/v5 v5.5.5 // indirect
    github.com/jackc/puddle/v2 v2.2.1 // indirect
    github.com/jinzhu/inflection v1.0.0 // indirect
    github.com/jinzhu/now v1.1.5 // indirect
    github.com/mattn/go-sqlite3 v1.14.24 // indirect
    github.com/rs/cors v1.11.1 // indirect
    golang.org/x/crypto v0.17.0 // indirect
    golang.org/x/sync v0.8.0 // indirect
    golang.org/x/text v0.19.0 // indirect
    gorm.io/driver/postgres v1.5.9 // indirect
)
```

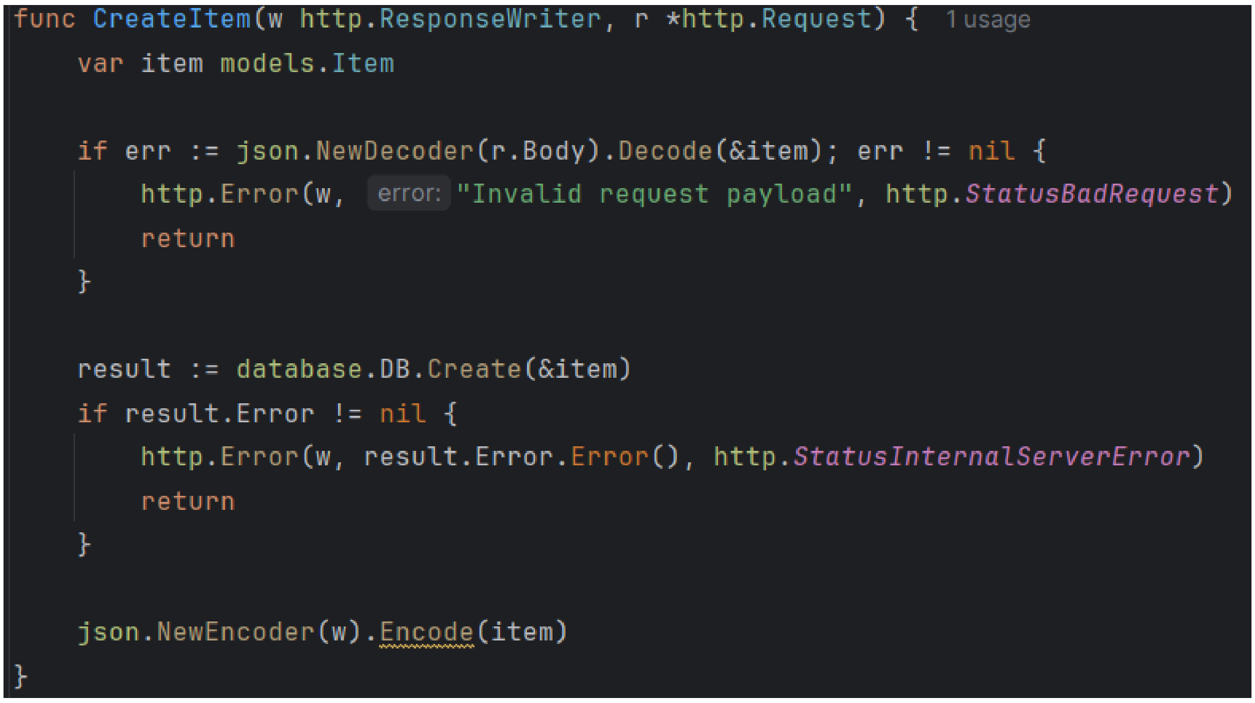
Working with Database

7.1 Part 1: Basic CRUD Operations

The database setup was done using PostgreSQL, and GORM was chosen as the ORM to interact with the database. I created the table for tasks, defined models, and implemented the following CRUD operations:

- **Create**: Allows users to add new tasks to the database.
- **Read**: Fetches tasks and displays them on the frontend.
- **Update**: Modifies existing tasks.
- **Delete**: Removes tasks from the system.

Example of GORM for creating a new task:

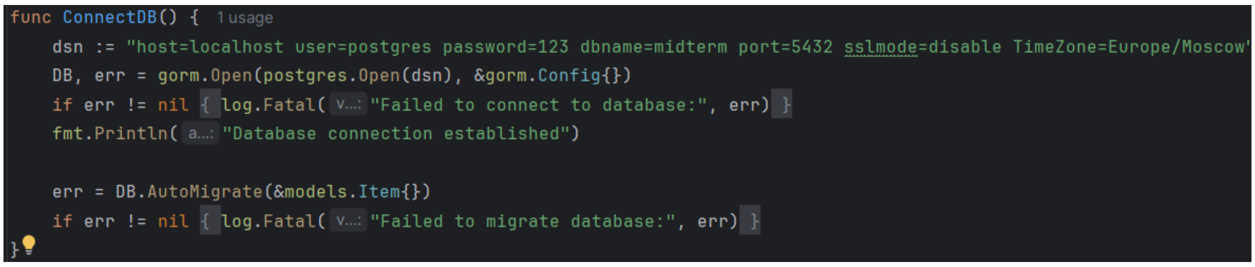


```go
func CreateItem(w http.ResponseWriter, r *http.Request) {  1 usage
    var item models.Item

    if err := json.NewDecoder(r.Body).Decode(&item); err != nil {
        http.Error(w,  error: "Invalid request payload", http.StatusBadRequest)
        return
    }

    result := database.DB.Create(&item)
    if result.Error != nil {
        http.Error(w, result.Error.Error(), http.StatusInternalServerError)
        return
    }

    json.NewEncoder(w).Encode(item)
}
```

7.2 Part 2: Complex Queries and Migrations

In addition to basic CRUD operations, I implemented database migrations to handle schema changes. GORM provides built-in support for database migrations, ensuring that updates to the model structure can be applied without losing data.

Example migration command:



```go
func ConnectDB() {  1 usage
    dsn := "host=localhost user=postgres password=123 dbname=midterm port=5432 sslmode=disable TimeZone=Europe/Moscow'
    DB, err = gorm.Open(postgres.Open(dsn), &gorm.Config{})
    if err != nil { log.Fatal( v...: "Failed to connect to database:", err) }
    fmt.Println( a...: "Database connection established")

    err = DB.AutoMigrate(&models.Item{})
    if err != nil { log.Fatal( v...: "Failed to migrate database:", err) }
}
```

Connecting with Frontend

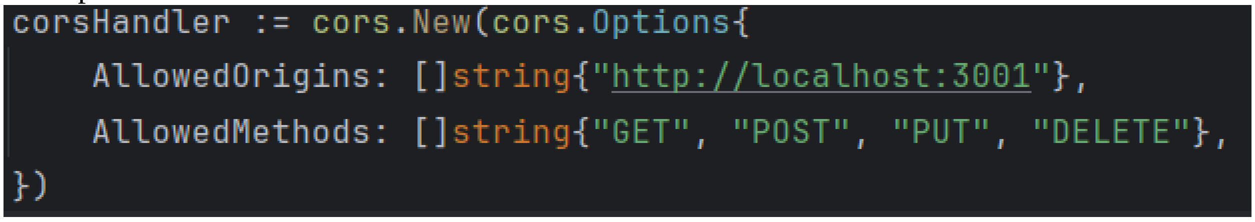
The backend was connected to the React frontend via a RESTful API. The backend exposes endpoints for each CRUD operation, and the frontend communicates with the API using fetch or Axios to send requests.

API Endpoints:

- GET /items: Fetch all tasks.
- POST /items: Create a new task.
- PUT /items/{id}: Update a task.
- DELETE /items/{id}: Delete a task.

Data was exchanged in **JSON** format, ensuring compatibility between the backend and frontend.

Example of connection to front-end:



```
corsHandler := cors.New(cors.Options{
    AllowedOrigins: []string{"http://localhost:3001"},
    AllowedMethods: []string{"GET", "POST", "PUT", "DELETE"},
})
```

Frontend Development with React

9.1 Components and Routing

We used **Create React App** to set up the React frontend. The main components created were:

- **TaskList**: Displays all tasks.
- **TaskForm**: A form for adding or editing tasks.
- **TaskDetail**: Shows details of a selected task.

React Router was used for client-side routing. This allows navigation between different pages without reloading the entire application.

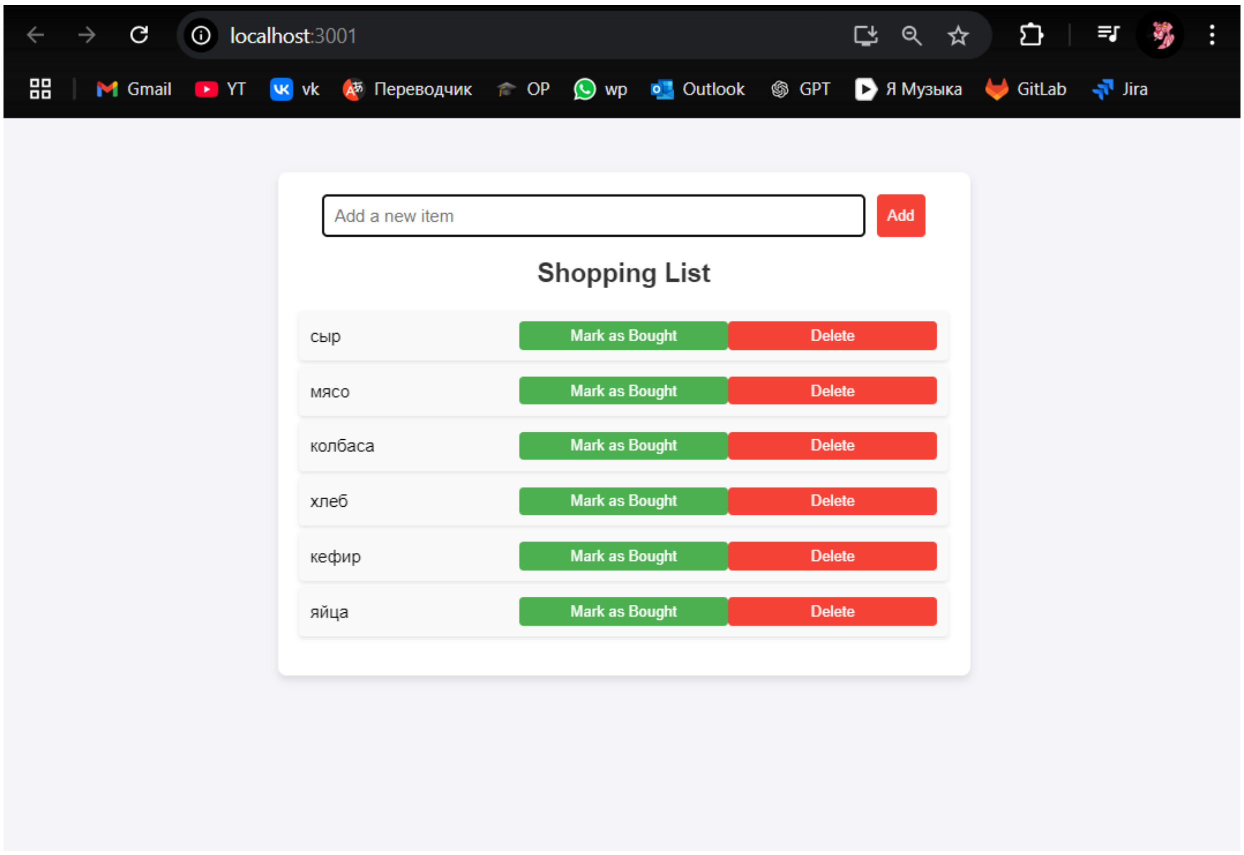
9.2 Pages Overview

In my project, all core features — viewing tasks, adding new tasks, and editing existing ones — are implemented on a single page. This allows users to manage tasks without navigating between different views, making the app simpler and more user-friendly.

Main Page Structure:

The main page contains a header, a form for adding or editing tasks, and a task list. Components like TaskList and TaskForm are all placed on the same page, enabling users to manage tasks in one location.

Illustration of Main Page:



Shopping List

сыр	Mark as Bought	Delete
мясо	Mark as Bought	Delete
колбаса	Mark as Bought	Delete
хлеб	Mark as Bought	Delete
кефир	Mark as Bought	Delete
яйца	Mark as Bought	Delete

Operations on the page:

- **Viewing tasks**: All tasks are displayed using the **TaskList** component. Users can see the full list of tasks and their statuses (e.g., whether a task is completed or not).
- **Adding tasks**: The **TaskForm** component is used for creating new tasks, and it's displayed alongside the task list.
- **Editing tasks**: Users can select a task to edit, and the form will update with the current data of the selected task.

Conclusion

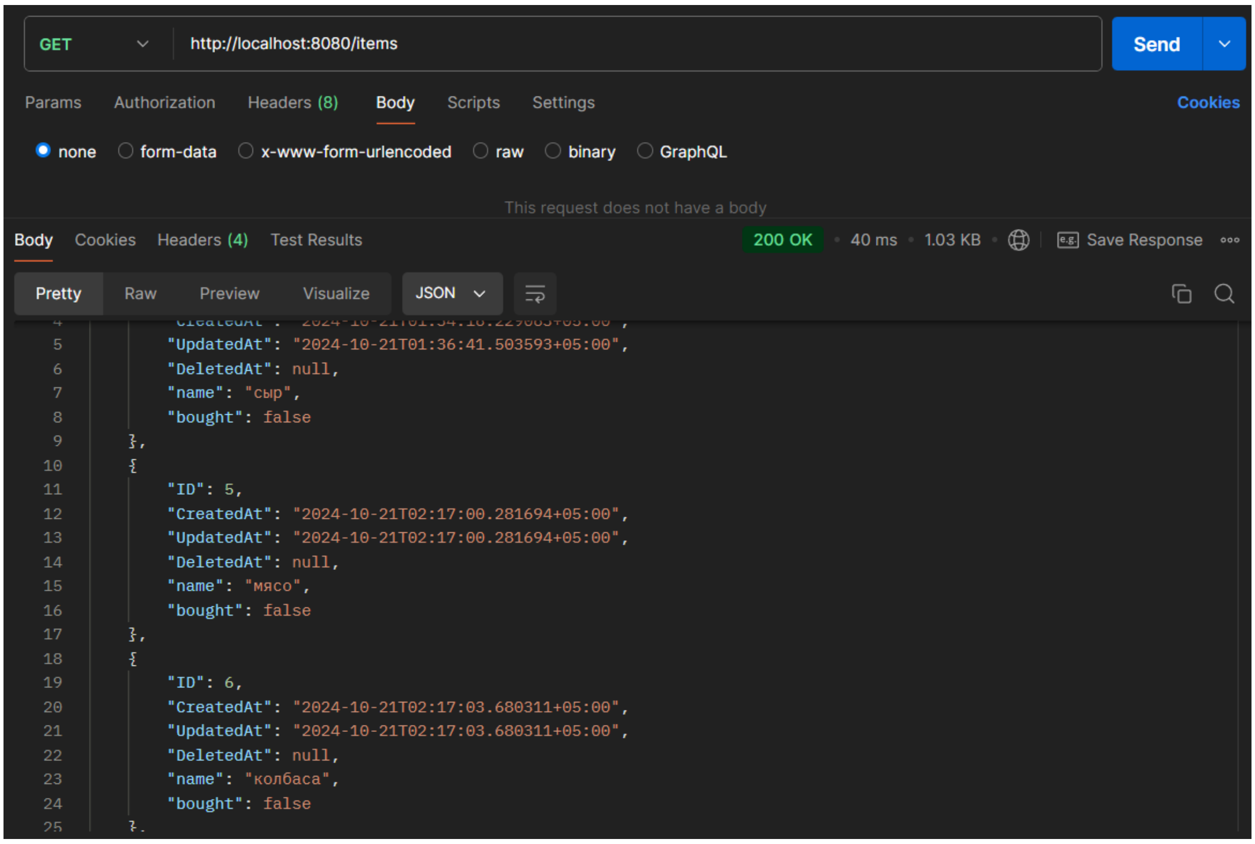
This project successfully demonstrated how to build a full-stack task management application using Go for the backend and React for the frontend. I've achieved objectives of creating a responsive, user-friendly application that supports basic task management operations. The integration of GORM for database interactions and React Router for client-side navigation allowed for a smooth user experience. Future improvements could include adding user authentication and task prioritization features.

References

- Go Documentation: https://golang.org/doc/
- React Documentation: https://reactjs.org/
- GORM Documentation: https://gorm.io/docs/
- PostgreSQL Documentation: https://www.postgresql.org/docs/

Appendices

Check in postman

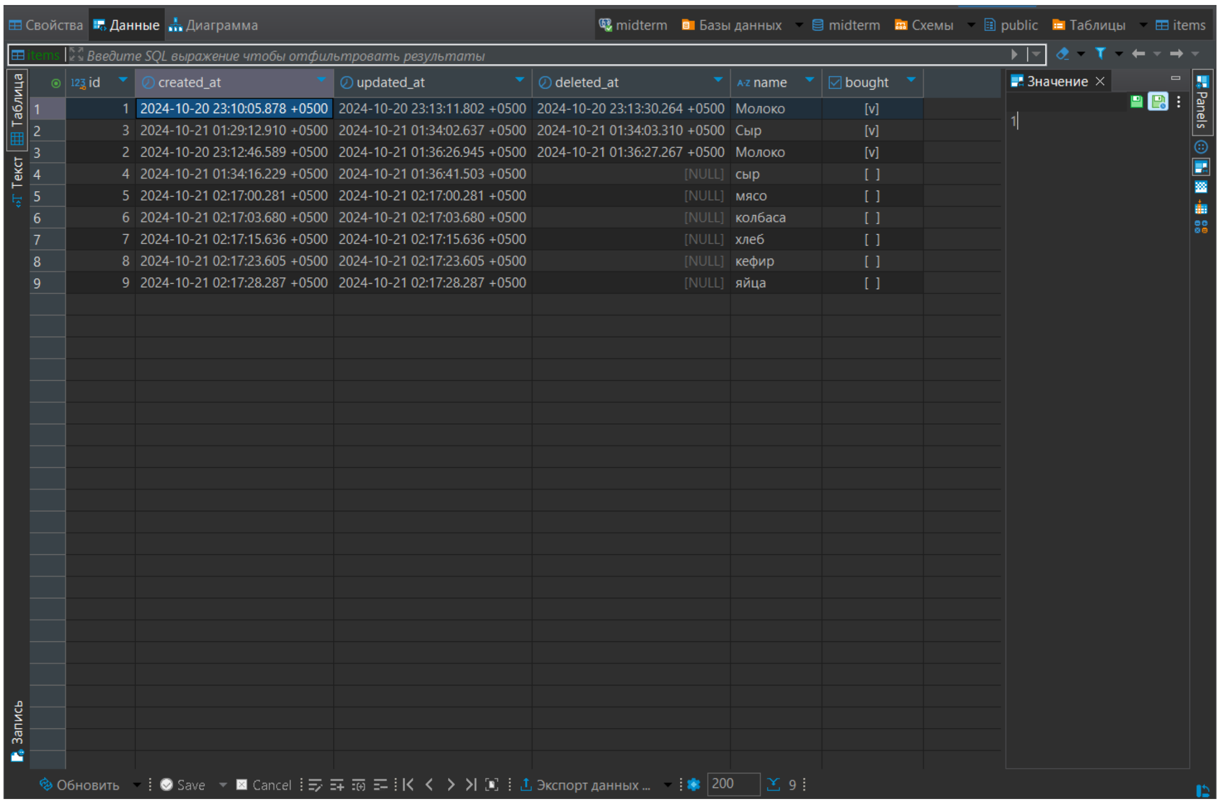

GET http://localhost:8080/items

This request does not have a body

200 OK · 40 ms · 1.03 KB

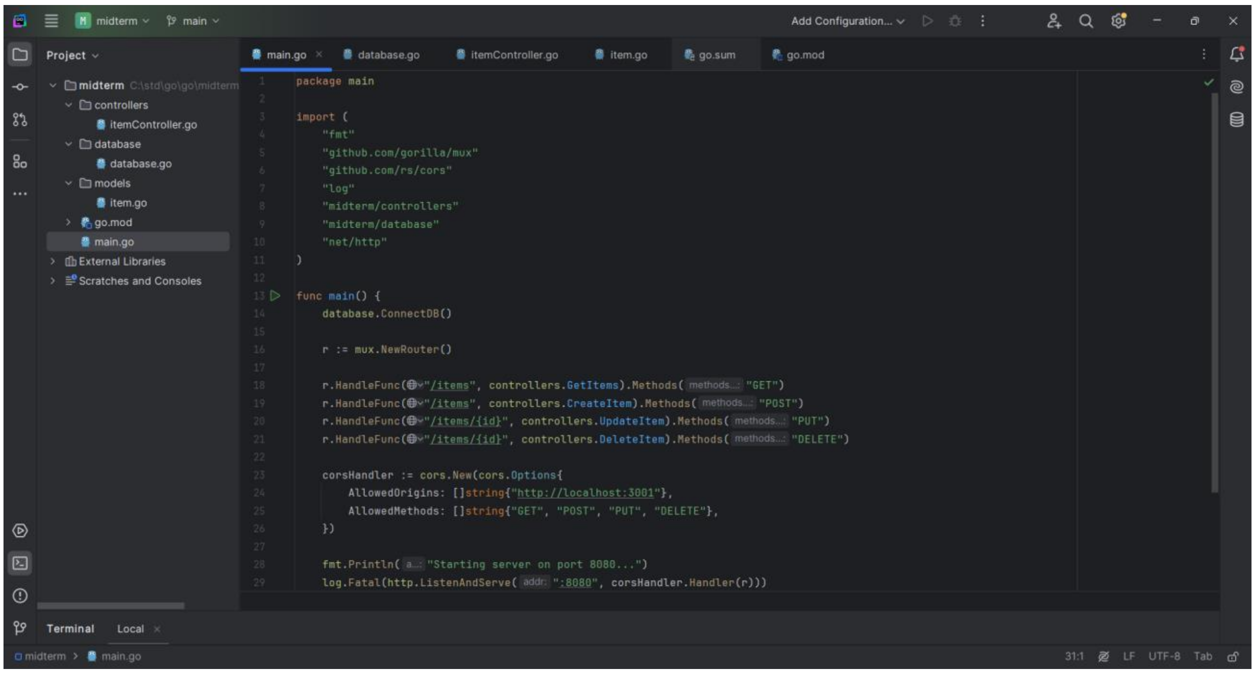
```
        "UpdatedAt": "2024-10-21T01:36:41.503593+05:00",
        "DeletedAt": null,
        "name": "сыр",
        "bought": false
    },
    {
        "ID": 5,
        "CreatedAt": "2024-10-21T02:17:00.281694+05:00",
        "UpdatedAt": "2024-10-21T02:17:00.281694+05:00",
        "DeletedAt": null,
        "name": "мясо",
        "bought": false
    },
    {
        "ID": 6,
        "CreatedAt": "2024-10-21T02:17:03.680311+05:00",
        "UpdatedAt": "2024-10-21T02:17:03.680311+05:00",
        "DeletedAt": null,
        "name": "колбаса",
        "bought": false
    },
```

Database:


Введите SQL выражение чтобы отфильтровать результаты

	id	created_at	updated_at	deleted_at	name	bought
1	1	2024-10-20 23:10:05.878 +0500	2024-10-20 23:13:11.802 +0500	2024-10-20 23:13:30.264 +0500	Молоко	[v]
2	3	2024-10-21 01:29:12.910 +0500	2024-10-21 01:34:02.637 +0500	2024-10-21 01:34:03.310 +0500	Сыр	[v]
3	2	2024-10-20 23:12:46.589 +0500	2024-10-21 01:36:26.945 +0500	2024-10-21 01:36:27.267 +0500	Молоко	[v]
4	4	2024-10-21 01:34:16.229 +0500	2024-10-21 01:36:41.503 +0500	[NULL]	сыр	[ ]
5	5	2024-10-21 02:17:00.281 +0500	2024-10-21 02:17:00.281 +0500	[NULL]	мясо	[ ]
6	6	2024-10-21 02:17:03.680 +0500	2024-10-21 02:17:03.680 +0500	[NULL]	колбаса	[ ]
7	7	2024-10-21 02:17:15.636 +0500	2024-10-21 02:17:15.636 +0500	[NULL]	хлеб	[ ]
8	8	2024-10-21 02:17:23.605 +0500	2024-10-21 02:17:23.605 +0500	[NULL]	кефир	[ ]
9	9	2024-10-21 02:17:28.287 +0500	2024-10-21 02:17:28.287 +0500	[NULL]	яйца	[ ]

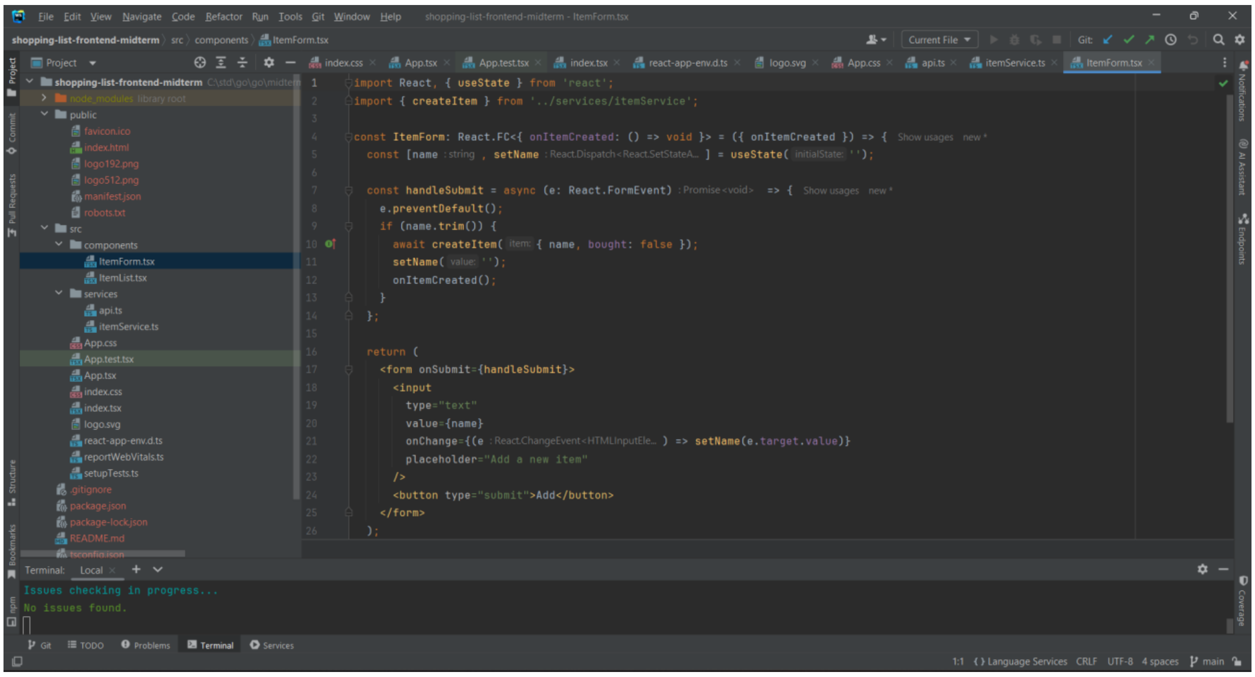
Structure of Go project:



```
midterm C:\std\go\go\midterm
  controllers
    itemController.go
  database
    database.go
  models
    item.go
  go.mod
  main.go
External Libraries
Scratches and Consoles
```

```go
package main

import (
	"fmt"
	"github.com/gorilla/mux"
	"github.com/rs/cors"
	"log"
	"midterm/controllers"
	"midterm/database"
	"net/http"
)

func main() {
	database.ConnectDB()

	r := mux.NewRouter()

	r.HandleFunc("/items", controllers.GetItems).Methods("GET")
	r.HandleFunc("/items", controllers.CreateItem).Methods("POST")
	r.HandleFunc("/items/{id}", controllers.UpdateItem).Methods("PUT")
	r.HandleFunc("/items/{id}", controllers.DeleteItem).Methods("DELETE")

	corsHandler := cors.New(cors.Options{
		AllowedOrigins: []string{"http://localhost:3001"},
		AllowedMethods: []string{"GET", "POST", "PUT", "DELETE"},
	})

	fmt.Println("Starting server on port 8080...")
	log.Fatal(http.ListenAndServe(":8080", corsHandler.Handler(r)))
```

Structure of React project:



```
shopping-list-frontend-midterm C:\std\go\go\midterm
  node_modules library root
  public
    favicon.ico
    index.html
    logo192.png
    logo512.png
    manifest.json
    robots.txt
  src
    components
      ItemForm.tsx
      ItemList.tsx
    services
      api.ts
      itemService.ts
    App.css
    App.test.tsx
    App.tsx
    index.css
    index.tsx
    logo.svg
    react-app-env.d.ts
    reportWebVitals.ts
    setupTests.ts
  .gitignore
  package.json
  package-lock.json
  README.md
  tsconfig.json
```

```tsx
import React, { useState } from 'react';
import { createItem } from '../services/itemService';

const ItemForm: React.FC<{ onItemCreated: () => void }> = ({ onItemCreated }) => {
  const [name, setName] = useState('');

  const handleSubmit = async (e: React.FormEvent) => {
    e.preventDefault();
    if (name.trim()) {
      await createItem({ name, bought: false });
      setName('');
      onItemCreated();
    }
  };

  return (
    <form onSubmit={handleSubmit}>
      <input
        type="text"
        value={name}
        onChange={(e) => setName(e.target.value)}
        placeholder="Add a new item"
      />
      <button type="submit">Add</button>
    </form>
  );
```

```
Issues checking in progress...
No issues found.
```