



TODO APPLICATION

Intern Software Engineer - Interview assignment

Maryam Nawas
maryamnawas2002@gmail.com
+94 76 994 4874

Table of Contents

Table of Contents	1
1. Introduction	2
2. Getting Started	3
2.1. Prerequisites	3
2.2. Backend Setup	3
2.3. Frontend Setup	3
2.4 Access the Application	3
2.5 Challenges faced	4
3. Backend (C# .NET)	5
3.1 Overview	5
3.2 API Endpoints	5
3.3 Key Code Snippets.....	5
3.4 Console Output	8
3.5 Error Handling	8
4. Frontend (ReactJS with TypeScript)	10
4.1 Overview	10
4.2 Key Components	10
4.3 Key Code Snippets.....	10
4.4 Styling and UI Design	13
5. Conclusion	17

1. Introduction

This project involves the development of a simple Todo application aimed at enabling users to efficiently manage their tasks. The application features essential CRUD operations—Create, Read, Update, and Delete allowing users to add new tasks, view existing tasks, modify task details, and remove tasks that are no longer needed. The backend of the application is built using C# .NET and integrates with an SQLite database to store and manage Todo data. The frontend is developed using ReactJS with TypeScript to provide a responsive and user-friendly interface. While the core functionality focuses on task management, optional user authentication can be included to enhance security and personalise the user experience. The project's goal is to deliver a complete, functional solution that demonstrates the integration of frontend and backend technologies, effective error handling, and a clean, intuitive user interface.

2. Getting Started

Follow these steps to set up and run the Todo application locally:

2.1. Prerequisites

Backend:

.NET 6.0 SDK

SQLite

Frontend:

Node.js (v14 or later)

npm (v6 or later)

2.2. Backend Setup

1. Clone the Repository: git clone <https://github.com/maryamnawas/ToDo-App.git>
cd Todo-App
2. Navigate to the Backend Directory: cd Backend
3. Install Dependencies: dotnet restore
4. Update Database: Ensure the SQLite database is correctly set up - dotnet ef database update
5. Run the Backend Server: dotnet run

2.3. Frontend Setup

1. Navigate to the Frontend Directory: cd Todo App
2. Install Dependencies: npm install
3. Run the Frontend Server: npm start

2.4 Access the Application

Open your web browser and navigate to <http://localhost:5173>.

The frontend should now be able to interact with the backend API, allowing you to manage your Todo items. Ensure that both the frontend and backend servers are running simultaneously for the application to function correctly.

2.5 Challenges faced

During the development of the Todo application, I encountered several challenges:

- **Unfamiliarity with TypeScript:** Initially, TypeScript was different from JavaScript, but I became comfortable with it through practice.
- **Learning C# .NET and SQLite:** Although familiar with C#, I was new to .NET and SQLite. I relied on documentation, ChatGPT, and YouTube tutorials to learn how to set up the backend.
- **Frontend Development:** I completed the frontend using ReactJS and TypeScript without significant issues due to my prior experience with ReactJS.
- **Backend CRUD Operations:** I faced difficulties with CRUD operations in the backend. The application worked with local storage but had issues integrating with the C# .NET backend, which I couldn't resolve due to my limited prior knowledge.

3. Backend (C# .NET)

3.1 Overview


The backend of this Todo application is developed using C# .NET and integrates with an SQLite database. It handles the core functionality of the application, including creating, reading, updating, and deleting (CRUD) Todo items. The backend API is structured to ensure smooth communication with the frontend, providing a robust and scalable solution for managing Todo data.

3.2 API Endpoints

- **GET /api/todos:** Retrieve a list of all Todo items.
- **POST /api/todos:** Create a new Todo item.
- **PUT /api/todos/{id}:** Update an existing Todo item by its ID.
- **DELETE /api/todos/{id}:** Delete a Todo item by its ID.

3.3 Key Code Snippets

Setting Up the Database Context

```
backend > TodoApi > Data >  TodoContext.cs > ...
1 | using Microsoft.EntityFrameworkCore;
2 | using TodoApi.Models;
3 |
4 | namespace TodoApi.Data
5 | {
6 |     5 references
7 |     public class TodoContext : DbContext
8 |     {
9 |         0 references
10 |         public TodoContext(DbContextOptions<TodoContext> options) : base(options) { }
11 |
12 |         6 references
13 |         public DbSet<Todo> Todos { get; set; }
    }
```

Model Definition

```

using System.ComponentModel.DataAnnotations;

namespace TodoApi.Models
{
    6 references
    public class Todo
    {
        3 references
        public int Id { get; set; }

        [Required]
        [StringLength(100, MinimumLength = 1, ErrorMessage = "Text length must be between 1 and 100 characters.")]
        0 references
        public string Text { get; set; } = string.Empty;

        0 references
        public bool Completed { get; set; }
    }
}

```

API Controller

```

using Microsoft.AspNetCore.Mvc;
using Microsoft.EntityFrameworkCore;
using TodoApi.Data;
using TodoApi.Models;

namespace TodoApi.Controllers
{
    [Route("api/[controller]")]
    [ApiController]
    1 reference
    public class TodoController : ControllerBase
    {
        11 references
        private readonly TodoContext _context;

        0 references
        public TodoController(TodoContext context)
        {
            _context = context;
        }

        // GET: api/Todo
        [HttpGet]
        0 references
        public async Task<ActionResult<IEnumerable<Todo>>> GetTodos()
        {
            return await _context.Todos.ToListAsync();
        }
    }
}

```

```

// GET: api/Todo/5
[HttpGet("{id}")]
1 reference
public async Task<ActionResult<Todo>> GetTodo(int id)
{
    var todo = await _context.Todos.FindAsync(id);

    if (todo == null)
    {
        return NotFound(new { message = $"Todo with id {id} not found." });
    }

    return todo;
}

// POST: api/Todo
[HttpPost]
0 references
public async Task<ActionResult<Todo>> PostTodo(Todo todo)
{
    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    _context.Todos.Add(todo);
    await _context.SaveChangesAsync();

    return CreatedAtAction(nameof(GetTodo), new { id = todo.Id }, todo);
}

```

```

// PUT: api/Todo/5
[HttpPut("{id}")]
0 references
public async Task<IActionResult> PutTodo(int id, Todo todo)
{
    if (id != todo.Id)
    {
        return BadRequest(new { message = "Id in the URL does not match the Id in the request body." });
    }

    if (!ModelState.IsValid)
    {
        return BadRequest(ModelState);
    }

    _context.Entry(todo).State = EntityState.Modified;

    try
    {
        await _context.SaveChangesAsync();
    }
    catch (DbUpdateConcurrencyException)
    {
        if (!TodoExists(id))
        {
            return NotFound(new { message = $"Todo with id {id} not found." });
        }
        else
        {
            throw;
        }
    }

    return NoContent();
}

```



```

// DELETE: api/ToDo/5
[HttpDelete("{id}")]
0 references
public async Task<IActionResult> DeleteToDo(int id)
{
    var todo = await _context.Todos.FindAsync(id);
    if (todo == null)
    {
        return NotFound(new { message = $"Todo with id {id} not found." });
    }

    _context.Todos.Remove(todo);
    await _context.SaveChangesAsync();

    return NoContent();
}

1 reference
private bool TodoExists(int id)
{
    return _context.Todos.Any(e => e.Id == id);
}
}

```

3.4 Console Output

To demonstrate successful build of the backend application, the console output of dotnet build confirms:

```

Done Building Project "D:\Github\Projects\ToDo-App\backend\ToDoApi\ToDoApi.csproj" (Clean target(s)).

Build succeeded.
    0 Warning(s)
    0 Error(s)

Time Elapsed 00:00:02.83
PS D:\Github\Projects\ToDo-App\backend\ToDoApi> dotnet run
Building...
D:\Github\Projects\ToDo-App\backend\ToDoApi\Middlewares\ErrorHandlerMiddleware.cs(40,48): warning CS8604: Possible null reference for parameter 'text' in 'Task HttpResponseMessageExtensions.WriteAsync(HttpResponse response, string text, CancellationToken cancellationToken, CancellationToken default(CancellationToken))'. [D:\Github\Projects\ToDo-App\backend\ToDoApi\ToDoApi.csproj]
warn: Microsoft.AspNetCore.Server.Kestrel.Core.KestrelServer[8]
      The ASP.NET Core developer certificate is not trusted. For information about trusting the ASP.NET Core developer certificate, visit https://aka.ms/aspnet/https-trust-dev-cert.
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: https://localhost:5001
info: Microsoft.Hosting.Lifetime[14]
      Now listening on: http://localhost:5000
info: Microsoft.Hosting.Lifetime[0]
      Application started. Press Ctrl+C to shut down.
info: Microsoft.Hosting.Lifetime[0]
      Hosting environment: Development
info: Microsoft.Hosting.Lifetime[0]
      Content root path: D:\Github\Projects\ToDo-App\backend\ToDoApi

```

3.5 Error Handling

```
// GET: api/ToDo/5
[HttpGet("{id}")]
1 reference
public async Task<ActionResult<ToDo>> GetToDo(int id)
{
    var todo = await _context.Todos.FindAsync(id);

    if (todo == null)
    {
        return NotFound(new { message = $"ToDo with id {id} not found." });
    }

    return todo;
}
```

These code snippets illustrate the core setup and functionality of the backend API, showcasing the database context, model definition, API controller, and basic error handling.

4. Frontend (ReactJS with TypeScript)

4.1 Overview

The frontend of this Todo application is developed using ReactJS with TypeScript, offering a responsive and intuitive user interface for managing Todo items. It interacts seamlessly with the backend API to perform CRUD operations, ensuring real-time updates and a smooth user experience.

4.2 Key Components

- **TodoList Component:** Displays a list of Todo items with options to edit, delete, and mark as completed.
- **TodoForm Component:** Allows users to add new Todo items with input validation and error handling.
- **App Component:** Main component rendering the application header and integrating TodoList component.

4.3 Key Code Snippets

TodoList Component:

```
import React, { useState, useEffect } from "react";
import TodoService from "../TodoService";
import TodoTypes from "../todo";
import TodoForm from "../TodoForm";
import "../CSS/ToDoList.css";
import { FaEdit, FaCheck } from "react-icons/fa";
import { RiDeleteBin5Fill } from "react-icons/ri";
import { GiCancel } from "react-icons/gi";

const TodoList: React.FC = () => {
  const [todos, setTodos] = useState<TodoTypes[]>([]);
  const [editingTodoId, setEditingTodoId] = useState<number | null>(null);
  const [editedTodoText, setEditedTodoText] = useState<string>("");

  useEffect(() => {
    const fetchTodos = async () => {
      try {
        const fetchedTodos = await TodoService.getTodos();
        setTodos(fetchedTodos);
      } catch (error) {
        console.error("Error fetching todos:", error);
        alert("Failed to fetch todos. Please check your network connection.");
      }
    };
    fetchTodos();
  }, []);
```

```

const handleEditStart = (id: number, text: string) => {
  setEditingTodoId(id);
  setEditedTodoText(text);
};

const handleEditCancel = () => {
  setEditingTodoId(null);
  setEditedTodoText("");
};

const handleEditSave = async (id: number) => {
  if (editedTodoText.trim() !== "") {
    try {
      const updatedTodo = await TodoService.updateTodo({
        id,
        text: editedTodoText,
        completed: false,
      });
      setTodos((prevTodos) =>
        prevTodos.map((todo) => (todo.id === id ? updatedTodo : todo))
      );
      setEditingTodoId(null);
      setEditedTodoText("");
    } catch (error) {
      console.error("Error updating todo:", error);
      alert("Failed to update todo. Please try again.");
    }
  } else {
    alert("Todo text cannot be empty.");
  }
};

```

```

const handleDeleteTodo = async (id: number) => {
  try {
    await TodoService.deleteTodo(id);
    setTodos((prevTodos) => prevTodos.filter((todo) => todo.id !== id));
  } catch (error) {
    console.error("Error deleting todo:", error);
    alert("Failed to delete todo. Please try again.");
  }
};

```

```

return (
  <div className="todoContainer">
    <TodoForm setTodos={setTodos} />
    <div className="todos">
      {todos.map((todo) => (
        <div className="items" key={todo.id}>
          {editingTodoId === todo.id ? (
            <div className="editText">
              <input
                type="text"
                value={editedTodoText}
                onChange={(e) => setEditedTodoText(e.target.value)}
                autoFocus={true}
              />
              <button onClick={() => handleEditSave(todo.id)}>
                <FaCheck />
              </button>
              <button className="cancelBtn" onClick={handleEditCancel}>
                <GiCancel />
              </button>
            </div>
          ) : (
            <div className="editBtn">
              <span>{todo.text}</span>
              <button onClick={() => handleEditStart(todo.id, todo.text)}>
                <FaEdit />
              </button>
            </div>
          )}
          <button onClick={() => handleDeleteTodo(todo.id)}>
            <RiDeleteBin5Fill />
          </button>
        </div>
      ))}
    </div>
  </div>
)

```

TodoForm Component

```

import React, { Dispatch, SetStateAction, useState } from "react";
import TodoService from "../TodoService";
import TodoTypes from "../todo";
import "../CSS/ToDoForm.css";

interface PropTypes {
  setTodos: Dispatch<SetStateAction<TodoTypes[]>>;
}

const TodoForm: React.FC<PropTypes> = ({ setTodos }) => {
  const [newTodoText, setNewTodoText] = useState<string>("");

  const handleAddTodo = async () => {
    if (newTodoText.trim() !== "") {
      try {
        const newTodo = await TodoService.addTodo(newTodoText);
        setTodos((prevTodos) => [...prevTodos, newTodo]);
        setNewTodoText("");
      } catch (error) {
        console.error("Error adding todo:", error);
        alert("Failed to add todo. Please try again.");
      }
    } else {
      alert("Todo text cannot be empty.");
    }
  };
};

```

```

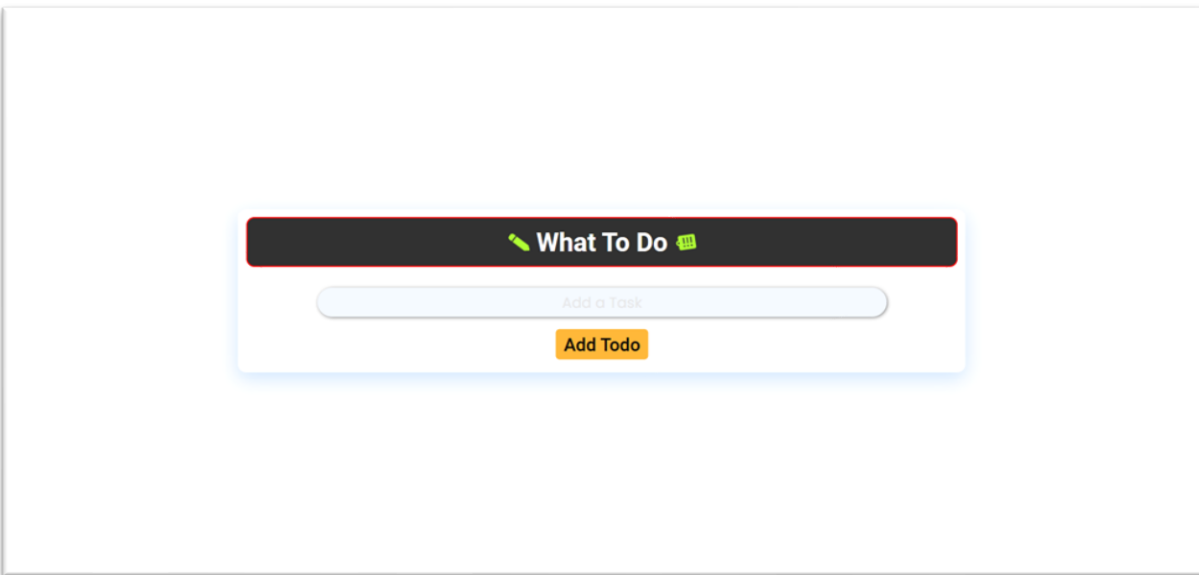
return (
  <div className="inputForm">
    <input
      type="text"
      value={newTodoText}
      onChange={(e) => setNewTodoText(e.target.value)}
      autoFocus={true}
      placeholder="Add a Task"
    />
    <button onClick={handleAddTodo}>Add Todo</button>
  </div>
);
};

export default TodoForm;

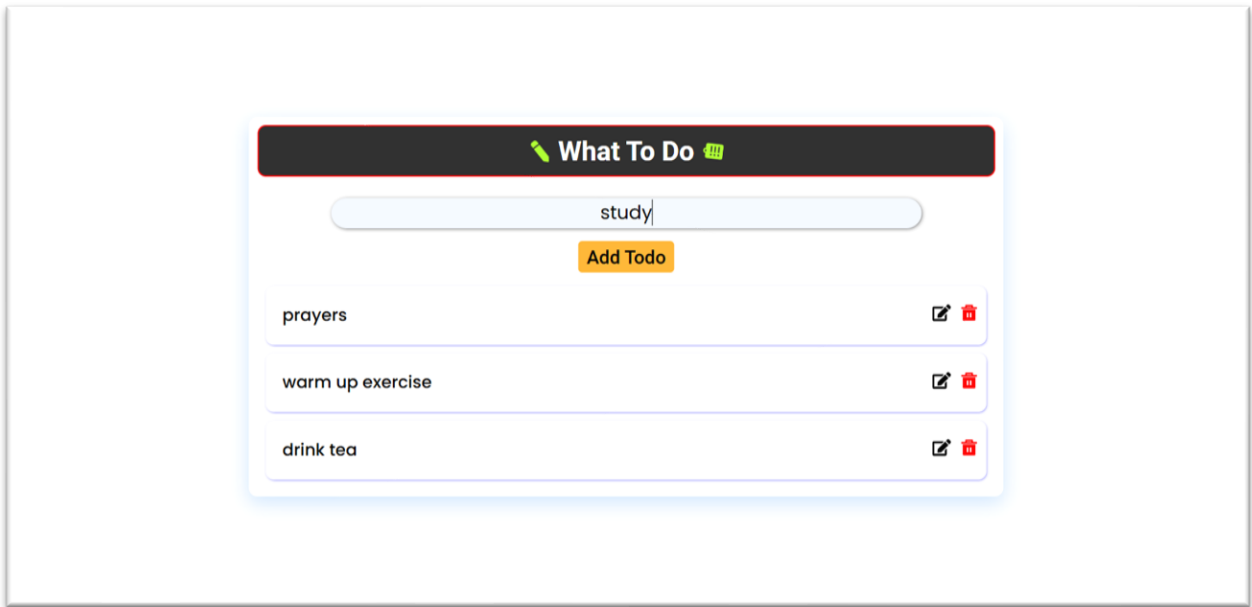
```

4.4 Styling and UI Design

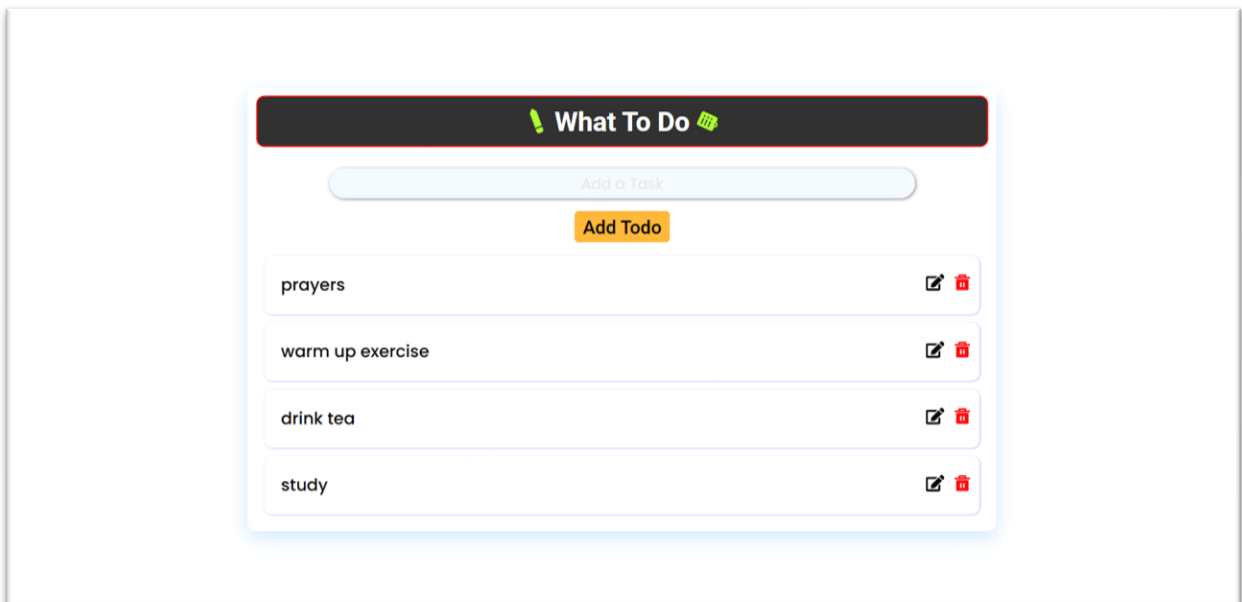
Ensure the application is styled for an aesthetically pleasing user experience, using CSS or styled components to create a cohesive design that enhances usability and visual appeal.



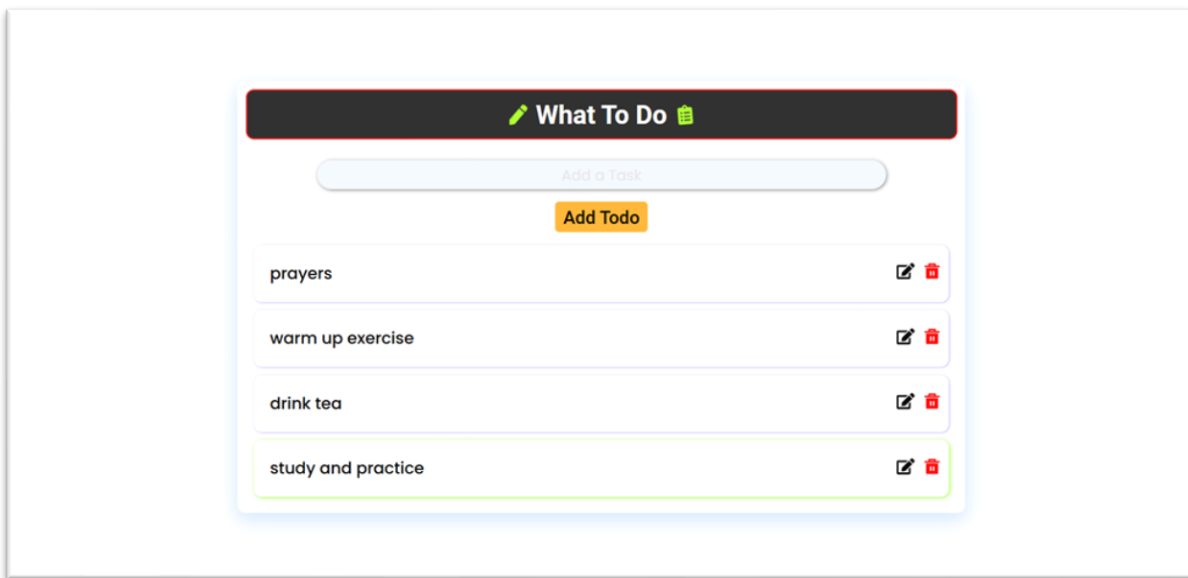
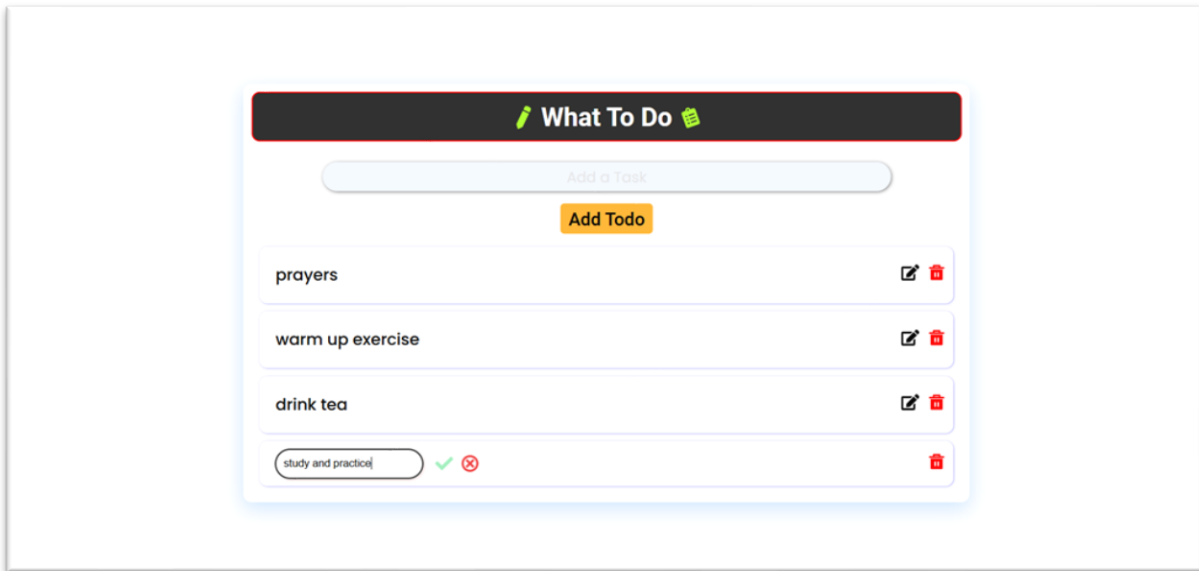
Create (Add Todo)



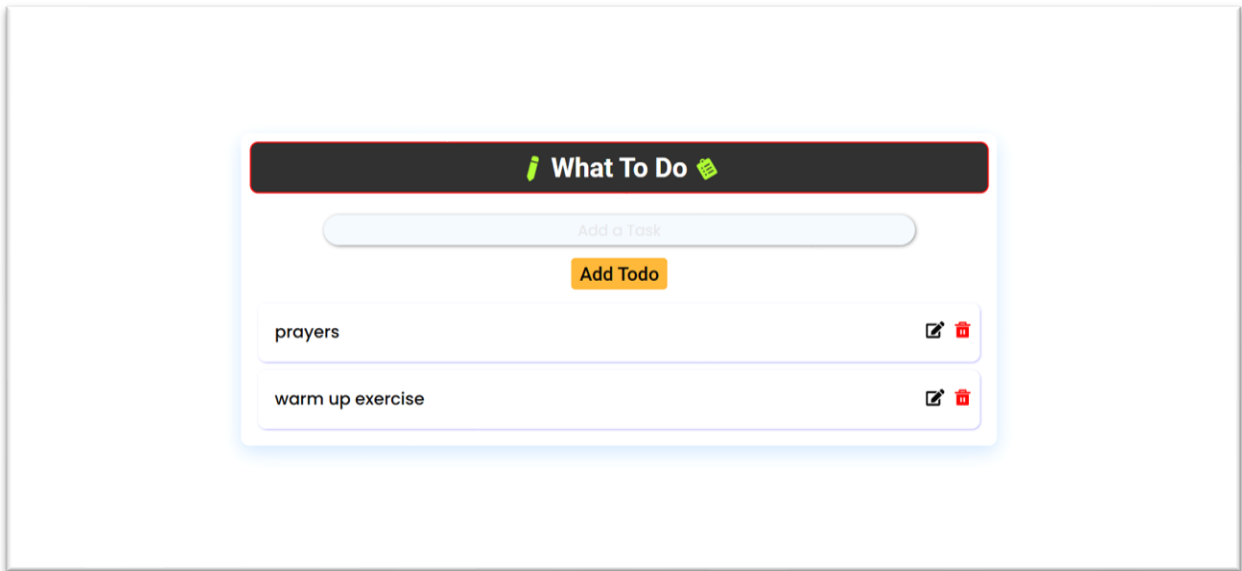
Read (View todo list)



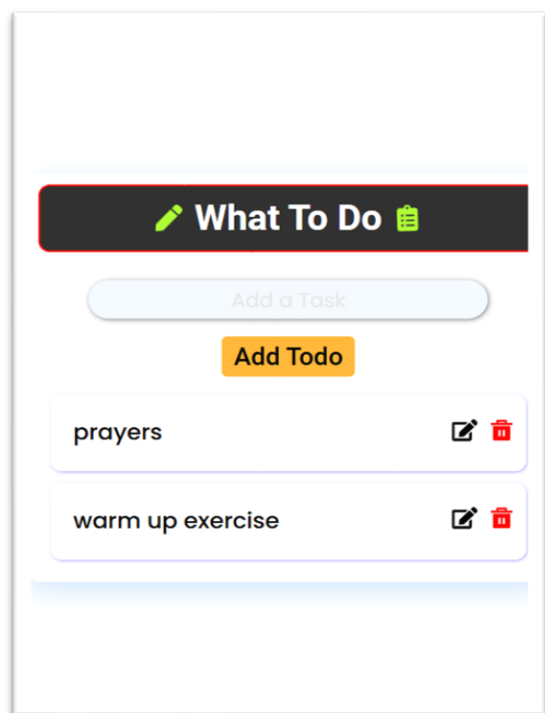
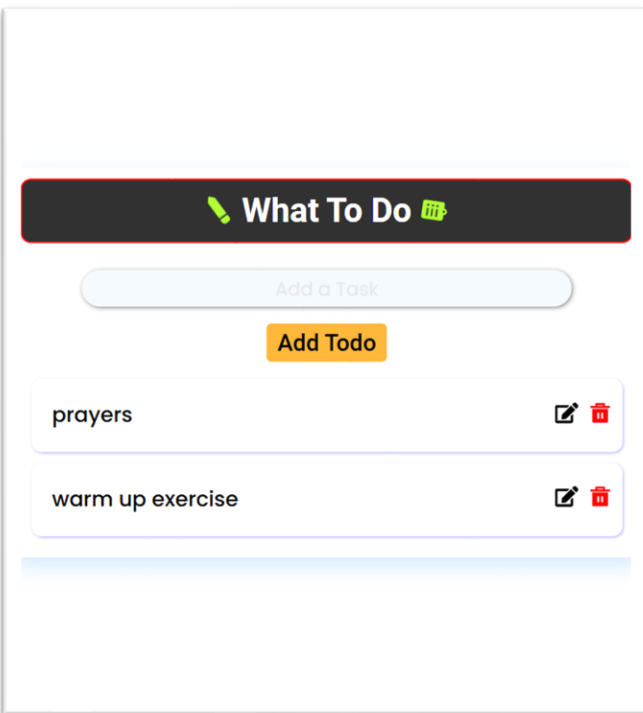
Update (Edit Todo)



Delete (Remove Todo)



Responsive



5. Conclusion

This Todo application project has been a valuable learning experience in integrating frontend and backend technologies. Using ReactJS with TypeScript on the frontend and C# .NET with SQLite on the backend, I successfully developed a functional task management system.

Key highlights include:

- **Technology Learning Curve:** Overcoming initial challenges with TypeScript and C# .NET, I effectively implemented CRUD operations on both ends of the application.
- **Frontend Development:** The ReactJS application showcases responsive UI components like `ToDoList` and `ToDoForm`, ensuring a seamless user experience for task management.
- **Backend Implementation:** Leveraging C# .NET, I built RESTful API endpoints for CRUD operations. Challenges were faced during CRUD integration, which impacted backend functionality.

Moving forward, refining backend CRUD operations and implementing robust error handling are crucial for enhancing application reliability. Despite challenges, this project underscores my proficiency in frontend and backend development, paving the way for future projects and learning opportunities.