**C++ CRUD Application Backend Project using Crow Framework**

**Title Page**

* **Project Name:** CrowBackend
* **Developer(s):** Grigorii Tishchuk
* **Date:** 2024-12-26
* **Supervisor/Instructor: Markus Maja**

**Table of Contents**

1. Introduction
2. Objectives and Requirements
3. Design and Architecture
   * Class Design
   * File Structure
4. Roadmap and Timeline
5. Implementation Details
   * Core Functionalities
   * Save Mechanism
   * Libraries Used
6. Challenges and Resolutions
7. Testing
8. Setup and Launch Instructions
9. Future Enhancements
10. References
11. Appendix
    * Archived Code
    * Developer’s Diary

**1. Introduction**

The **CrowBackend** project is a C++ application for managing tasks with basic CRUD (Create, Read, Update, Delete) functionality. The application is designed to interact via HTTP requests, utilizing the Crow framework for routing and the nlohmann\_json library for JSON handling. The primary purpose is to build a modular, extendable, and robust system while adhering to C++ project requirements.

**2. Objectives and Requirements**

**Objectives**

* Develop a task management application.
* Implement saving and loading functionality to maintain task data persistently.
* Ensure modular design with class-based architecture.

**Requirements**

* Use **classes** (e.g., Task, TaskManager) with definitions in separate files.
* Implement a **save/load mechanism** for data persistence.
* Incorporate **sufficient functionality** beyond basic operations.
* Retain and archive unused or non-functional code for evaluation.
* Optionally, use other programming languages (Python, JavaScript) if necessary.

**3. Design and Architecture**

**Class Design**

**Task**

* Represents an individual task.
* Attributes:
  + id: Unique identifier.
  + description: Task description.
  + completed: Status of the task.
* Methods:
  + getId(), getDescription(), isCompleted().

**TaskManager**

* Manages a collection of tasks.
* Attributes:
  + tasks: A map of task IDs to Task objects.
  + nextId: Counter for generating unique IDs.
* Methods:
  + addTask(), deleteTask(), updateTask(), getTask().

**File Structure**

└── 📁crowbackend

└── 📁.vscode

└── launch.json

└── settings.json

└── tasks.json

└── CMakeLists.txt

└── main.cpp

└── Task.cpp

└── Task.hpp

└── TaskManager.cpp

└── TaskManager.hpp

└── vcpkg.json

**4. Roadmap and Timeline**

| **Phase** | **Week** | **Tasks** |
| --- | --- | --- |
| Initial Setup | Week 1 | Define requirements, set up environment, configure Crow and JSON libs. |
| Core Functionality | Week 2 | Implement Task and TaskManager classes with basic CRUD operations. |
| Save/Load Mechanism | Week 3 | Add JSON-based save/load functionality. |
| Integration & Testing | Week 4 | Test functionality, add unit tests, and document the project. |

**5. Implementation Details**

**Core Functionalities**

1. **Task Management:**
   * Add, update, delete, and retrieve tasks via HTTP endpoints.
   * Example: /tasks/add endpoint for adding tasks.
2. **Data Persistence:**
   * Save tasks to a JSON file.
   * Load tasks from the JSON file on application startup.
3. **Error Handling:**
   * Validate inputs and handle exceptions gracefully.

**Save Mechanism**

* **Save:** Writes the tasks map to a JSON file using nlohmann::json.
* **Load:** Reads the JSON file into tasks during initialization.

**Libraries Used**

* **Crow:** Crow is a modern C++ microframework that simplifies the creation of RESTful APIs and web applications. It provides robust routing mechanisms and lightweight HTTP handling.
* **nlohmann\_json:** A popular single-header library for JSON parsing and serialization in C++. It simplifies the conversion of C++ objects to JSON and vice versa, making it ideal for managing persistent task data.

**6. Challenges and Resolutions**

**Challenge 1: Crow and vcpkg configuration issues.**

**Resolution:** Configuring Crow and nlohmann\_json using Visual Studio Code required integrating multiple settings files (CMakeLists.txt, settings.json, tasks.json, launch.json). The issue was resolved by linking libraries within the project folder rather than relying on system-level vcpkg paths.

**Challenge 2: Implementing save/load functionality.**

**Resolution:** Utilized nlohmann\_json library to serialize and deserialize task objects efficiently.

**Challenge 3: Handling linker errors for to\_json and from\_json methods.**

**Resolution:** The to\_json and from\_json methods for the Task class were implemented in the Task.hpp header file and marked as inline. This was necessary to prevent linker errors caused by defining these methods in multiple translation units. By marking them as inline, the compiler allows their definitions to be included in multiple translation units without conflict.

**Challenge 4: Configuring CMake to work with Visual Studio Code.**

**Resolution:** The project setup involved ensuring correct linking of Crow and nlohmann\_json libraries by adjusting file paths in CMAKE\_TOOLCHAIN\_FILE and CMAKE\_PREFIX\_PATH to point to the project folder’s vcpkg setup. Steps to resolve build issues:

1. Delete CMake cache and files in the build folder:
2. Remove-Item -Recurse -Force CMakeCache.txt, CMakeFiles
3. Reconfigure the project:
4. cmake ..
5. Build the project:
6. cmake --build . --config Debug
7. Locate the executable in the Debug folder (e.g., CrowBackend.exe) and launch it locally on localhost:18080.

**7. Testing**

**Test Cases**

**Add Task**

* **Input:** POST request with {"description": "Task 1"}.
* **Expected Output:** Task created successfully with ID and default completed status as false.

**Update Task**

* **Input:** PUT request to /tasks/<id> with {"completed": true}.
* **Expected Output:** Task status updated successfully.

**Delete Task**

* **Input:** DELETE request to /tasks/<id>.
* **Expected Output:** Task removed from the collection.

**Testing Framework:**

* Google Test (gtest) for unit testing.

**8. Setup and Launch Instructions**

**Prerequisites**

To set up and run the project on another machine, ensure the following prerequisites are installed:

1. **Compiler:** GCC (MinGW on Windows) or any C++17-compliant compiler.
2. **CMake:** Version 3.15 or later.
3. **vcpkg:** Installed and configured for package management.
4. **Visual Studio Code:** For managing the build and editing the code.
5. **Crow and nlohmann\_json libraries:** These will be automatically handled through vcpkg.json.

**Step-by-Step Setup**

1. **Clone the Repository**
   * Clone the project repository to your local machine:
   * git clone <repository\_url>
   * cd crowbackend
2. **Install Dependencies**
   * Use vcpkg to install the required libraries, because vcpkg.json contains necessary libraries for this project. Installation works in manifesto mode. I If you need other libraries, add them to json file and execute install again:
   * vcpkg install
3. **Set Up Visual Studio Code**
   * Ensure the .vscode folder contains the necessary configuration files:
     + settings.json
     + tasks.json
     + launch.json
   * These files are pre-configured to work with the local vcpkg\_installed folder.
4. **Build the Project**
   * Open the project in Visual Studio Code.
   * Run the CMake build task (usually accessible via Ctrl+Shift+B).
5. **Run the Application**
   * Use the launch.json configuration to launch the application directly within Visual Studio Code or execute the compiled binary manually:
   * ./crowbackend

**Troubleshooting Build Issues**

1. Delete CMake cache and files in the build folder:
2. Remove-Item -Recurse -Force CMakeCache.txt, CMakeFiles
3. Reconfigure the project:
4. cmake ..
5. Build the project:
6. cmake --build . --config Debug
7. Locate the executable in the Debug folder (e.g., CrowBackend.exe) and launch it locally on localhost:18080.

**9. Future Enhancements**

1. Add user authentication for managing tasks.
2. Create a front-end interface using React.js or plain HTML.
3. Enhance error handling with detailed logs.

**10. References**

* [Crow Documentation](https://crowcpp.org/)
* [nlohmann\_json GitHub](https://github.com/nlohmann/json)
* [CMake Documentation](https://cmake.org/documentation/)

**11. Appendix**

**Archived Code**

* TaskManagerTests.cpp:

#include "../TaskManager.hpp"

#include <gtest/gtest.h>

#include <fstream>

/\*

CURRENTLY TESTS DO NOT WORK: EXE FILE IS NOT COMPILED CORRECTLY.

UNDER DEVELOPMENT CURRENTLY

\*/

// Helper function to remove the test file after each test

void removeTestFile(const std::string& fileName) {

    std::remove(fileName.c\_str());

}

// Test fixture for TaskManager tests

class TaskManagerTest : public ::testing::Test {

protected:

    std::string testFileName = "test\_tasks.json";

    TaskManager taskManager;

    TaskManagerTest() : taskManager(testFileName) {}

// Constructor that initializes the TaskManager with the test file name

    void SetUp() override {

        // Remove the test file before each test

        removeTestFile(testFileName);

    }

// Destructor that removes the test file after each test

    void TearDown() override {

        // Remove the test file after each test

        removeTestFile(testFileName);

    }

};

// Test adding a task

TEST\_F(TaskManagerTest, AddTask) {

    Task task = taskManager.addTask("Test Task");

    auto tasks = taskManager.getAllTasks();

    ASSERT\_EQ(tasks.size(), 1);

    ASSERT\_EQ(tasks[1].getDescription(), "Test Task");

}

// Test deleting a task

TEST\_F(TaskManagerTest, DeleteTask) {

    Task task = taskManager.addTask("Test Task");

    taskManager.deleteTask(task.getId());

    auto tasks = taskManager.getAllTasks();

    ASSERT\_EQ(tasks.size(), 0);

}

// Test updating task status

TEST\_F(TaskManagerTest, UpdateTaskStatus) {

    Task task = taskManager.addTask("Test Task");

    taskManager.updateTask(task.getId(), "Updated Task Description");

    auto tasks = taskManager.getAllTasks();

    ASSERT\_EQ(tasks[task.getId()].getDescription(), "Updated Task Description");

}

// Test loading tasks from file

TEST\_F(TaskManagerTest, LoadFromFile) {

    Task task = taskManager.addTask("Test Task");

    taskManager.saveToFile();

    TaskManager newTaskManager(testFileName);

    newTaskManager.loadFromFile();

    auto tasks = newTaskManager.getAllTasks();

    ASSERT\_EQ(tasks.size(), 1);

    ASSERT\_EQ(tasks[1].getDescription(), "Test Task");

}

/\*\*

 \* @brief Main entry point for the test executable

 \*

 \* Initializes the Google Test framework and runs all tests.

 \*

 \* @param argc The number of command line arguments

 \* @param argv The command line arguments

 \* @return The exit status of the test executable

 \*/

int main(int argc, char \*\*argv) {

    ::testing::InitGoogleTest(&argc, argv);

    return RUN\_ALL\_TESTS();

}