MSCS-632: Residency Project: Deliverable 3 Report: Simple Chat Application: Final Report

---

Shabnam Shaikh, Sakchham Sangroula, Sandesh Pokharel, Nihar Turumelle, Romika Souda

University of the Cumberlands

MSCS-632 Advanced Programming Languages

Dr. Vanessa Cooper

April 06, 2025

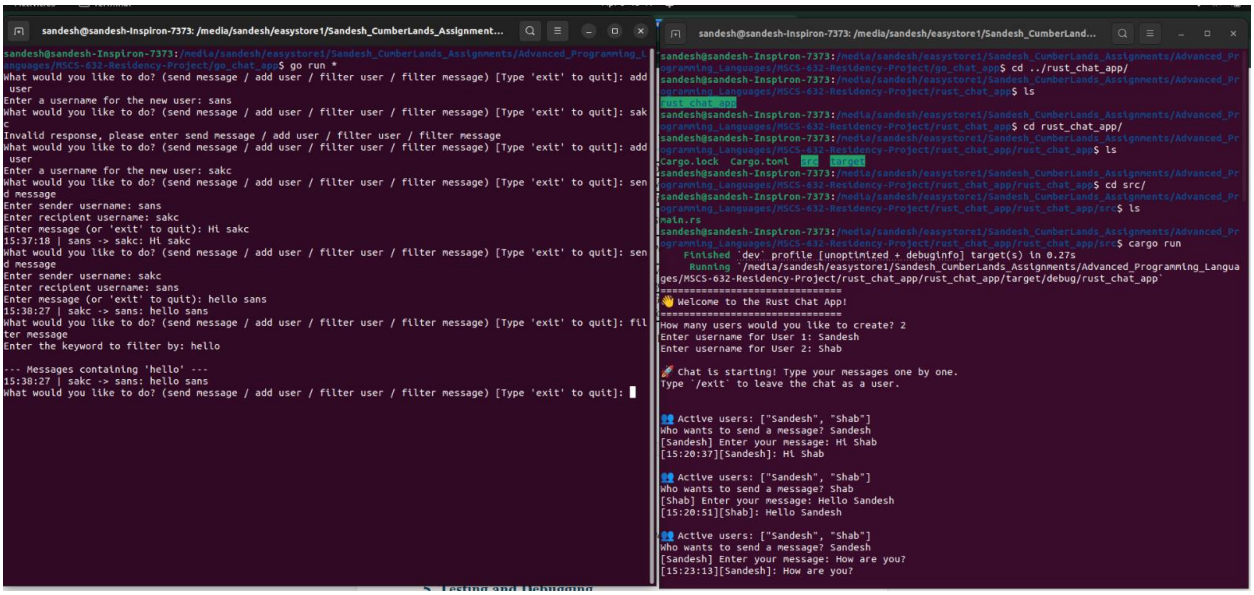
1. Introduction

This report presents a detailed comparison of the final implementation of our CLI-based Chat Application using two different programming languages: **Rust** and **Go**. The project was designed as part of the MSCS-632 Advanced Programming Languages course and aims to showcase how different languages approach the same application problem using their own syntax rules, concurrency models, and language-specific paradigms.

Throughout the development, our goal was to maintain functional parity between the two versions while also exploring the unique idioms and strengths of each language. The Rust version focuses on safety, strict ownership, and asynchronous message handling using tokio (Jonsson et al., 2020). On the other hand, the Go version leverages lightweight concurrency using goroutines and channels, offering a more traditional menu-driven approach to CLI interactions (Google, 2023).
GitHub Link: **https://github.com/sanspokharel26677/MSCS-632-Residency-Project#**

This report discusses the differences, challenges, and developer experiences between the two implementations, using code snippets and screenshots to support our evaluation.

📸 **Screenshot :** Application outputs from both Rust and Go versions running in the terminal.



2. Application Summary & Final Features

The final version of the application in both languages includes the following core features:

- Dynamic user creation
- CLI-based message input and output
- Message history with timestamps
- Per-user message tracking
- Message filtering by user and keyword
- User exit functionality
- End-of-chat summary report

Despite differences in syntax and control flow, both versions achieve identical output expectations and match the feature checklist outlined in the project requirements.

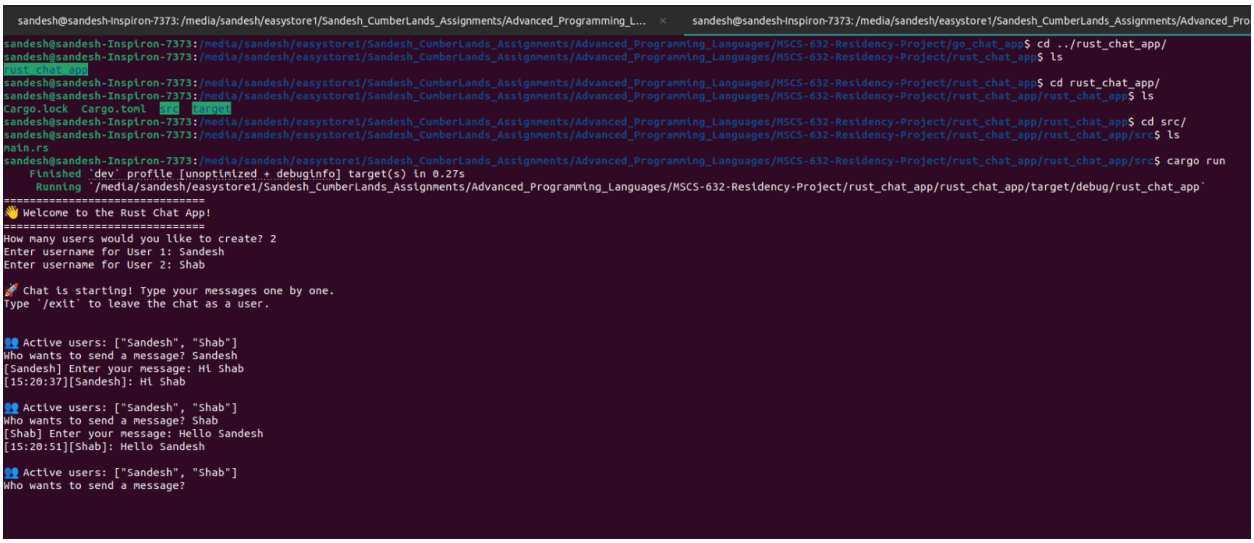
Rust

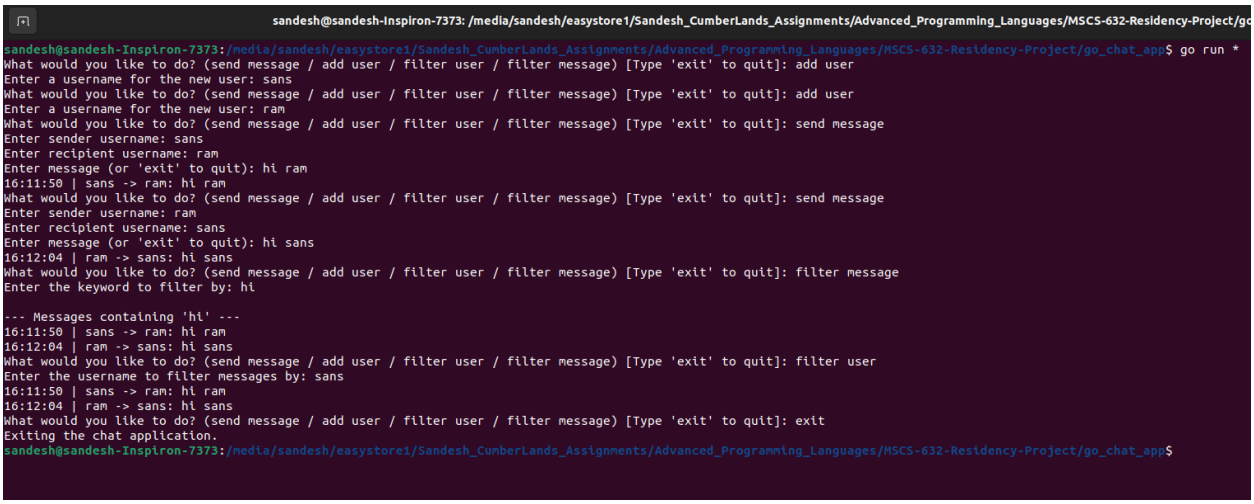
- Turn-based interaction
- User rotation for message input
- Message queue (VecDeque)
- Shared server state using Arc<Mutex<>>

Go

- Menu-based interaction with user commands
- Real-time message routing using goroutines
- Per-user channels for message receipt
- Message history stored globally for filtering

📸 **Screenshot:** Rust and Go feature menus or CLI flows





3. Code Architecture & Design Patterns

Rust Architecture

Rust was implemented in a single file (main.rs) using multiple structs and a shared server instance. The key architectural decision was to use Arc<Mutex<>> to allow concurrent, thread-safe access to the shared ChatServer.

let server = Arc::new(Mutex::new(ChatServer::new()));

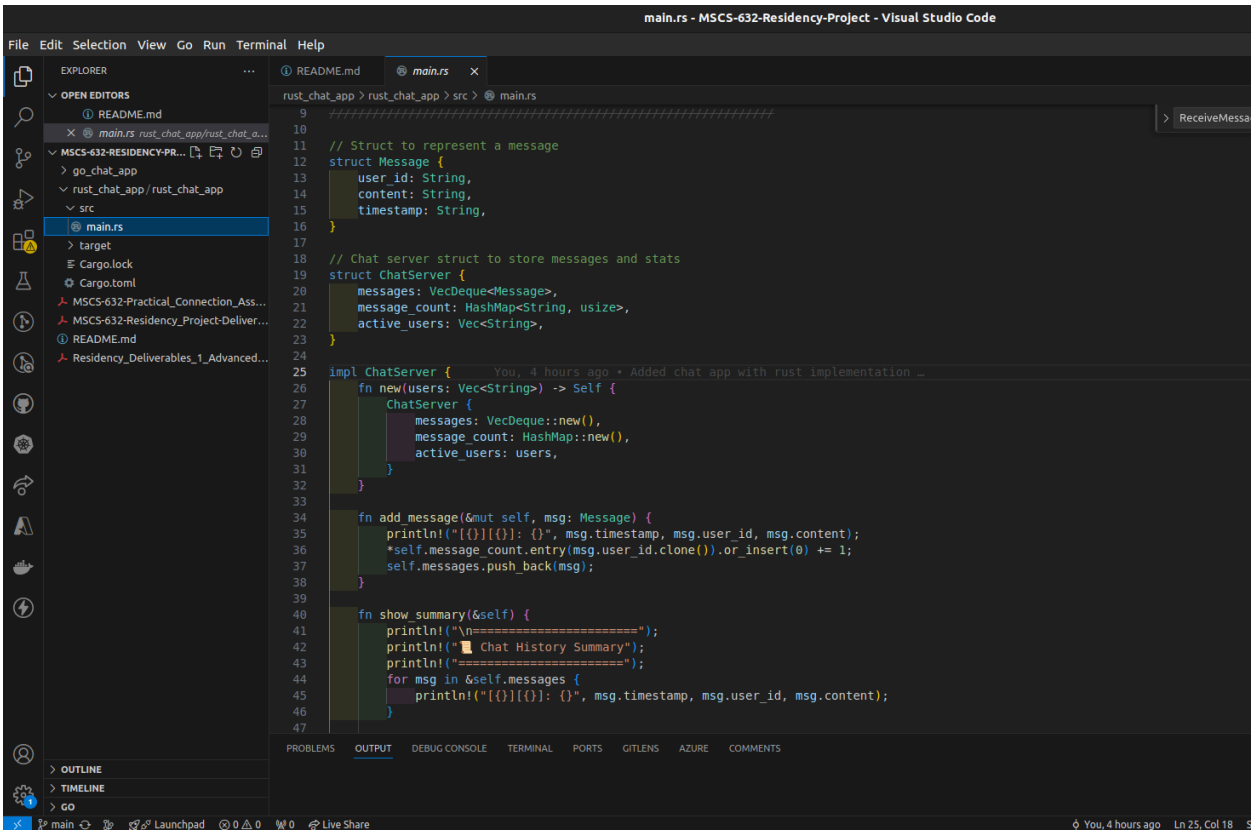
The server holds all active users and message history. Each user is managed by an async fn task and takes turns sending messages through the terminal.

async fn start_user(id: &str, server: Arc<Mutex<ChatServer>>) {

    // CLI interaction loop

}

📸 **Screenshot Placeholder:** Rust start_user() function and shared state setup



Go Architecture

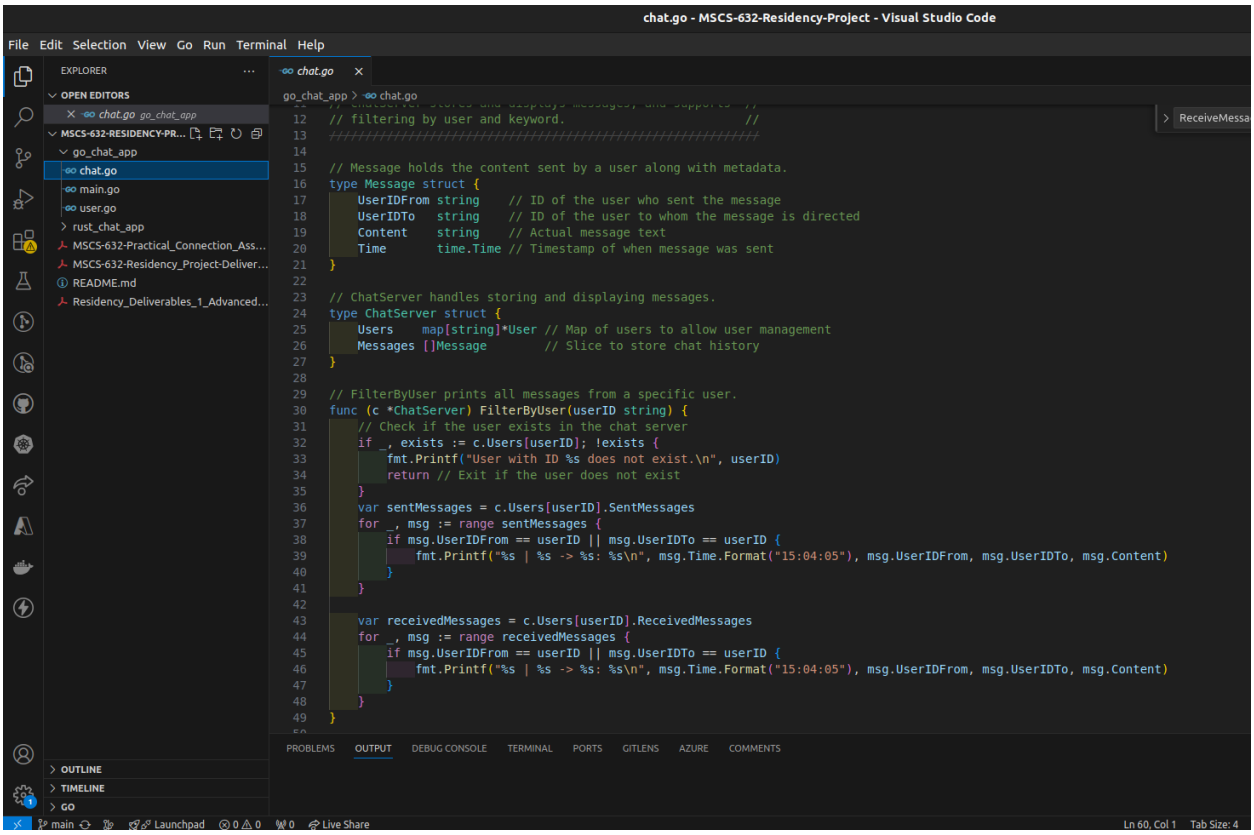
The Go version uses multiple source files (main.go, chat.go, and user.go) to separate concerns. Each user has a Received channel and a goroutine that continuously listens for incoming messages.

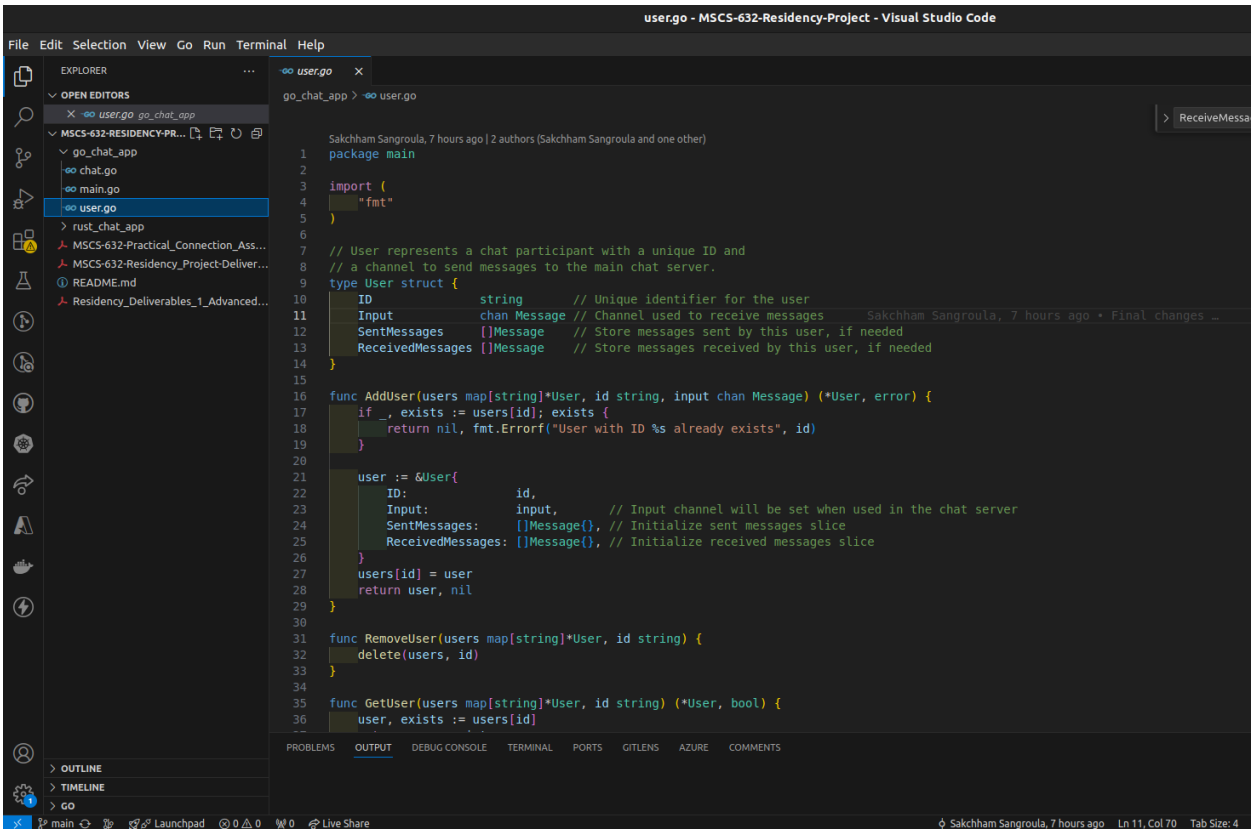
```
go user.ReceiveMessages()
```

Messages are passed through the ChatServer which acts as the router.

```
type ChatServer struct {
    Users    map[string]*User
    Messages []Message
}
```

Screenshot: Go goroutine handling and message routing in chat.go and user.go





4. Language-Specific Features & Syntax Differences

Memory Safety and Ownership (Rust)

Rust's strict ownership system required careful handling of shared data. All shared state had to be wrapped in Arc<Mutex<>>, and all updates to shared resources required explicit locking (Klabnik & Nichols, 2019).

Concurrency Models

- **Rust:** Uses async functions, the tokio runtime, and .await for non-blocking I/O.

- **Go:** Uses goroutines (lightweight threads) and channels to send/receive data concurrently (Google, 2023).

Data Structures

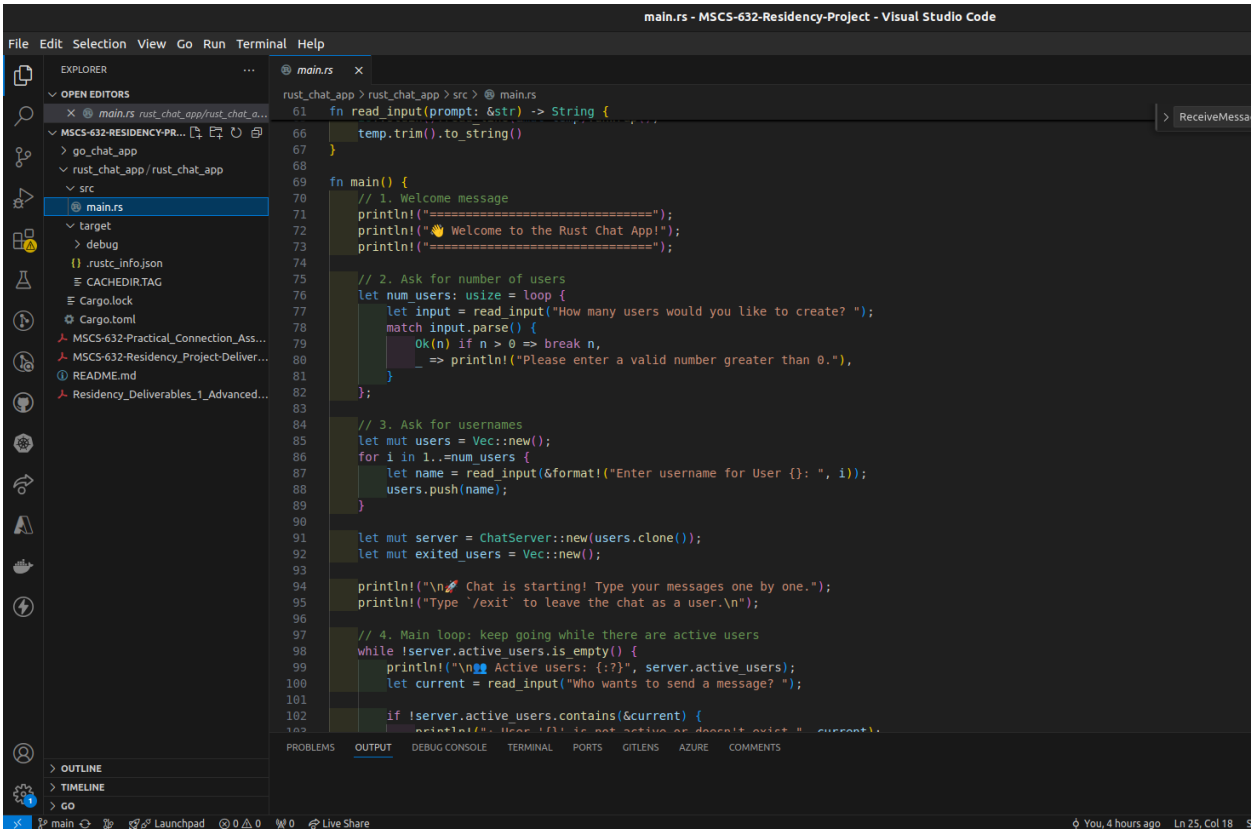
Rust's collections like VecDeque and HashMap are strongly typed and require explicit borrowing and mutability.

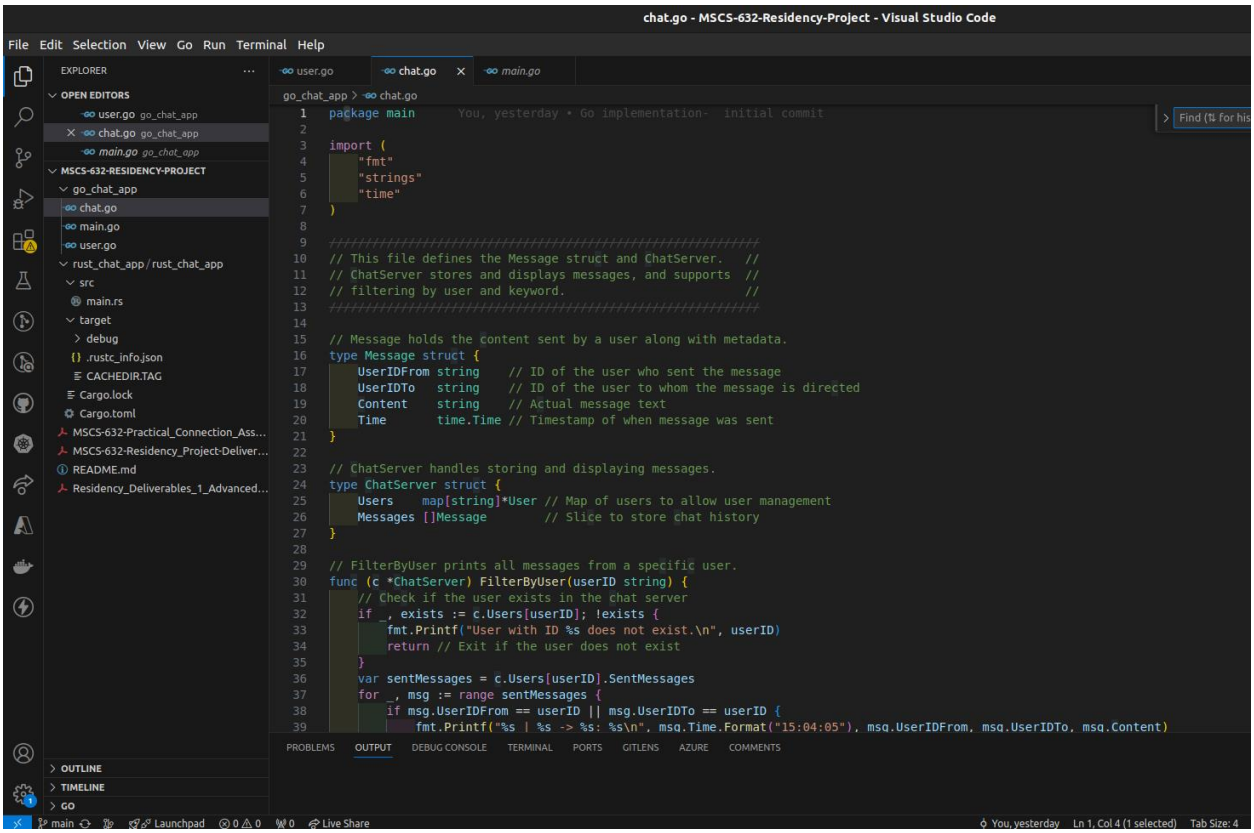
Go uses slices, maps, and basic structs with simpler syntax.

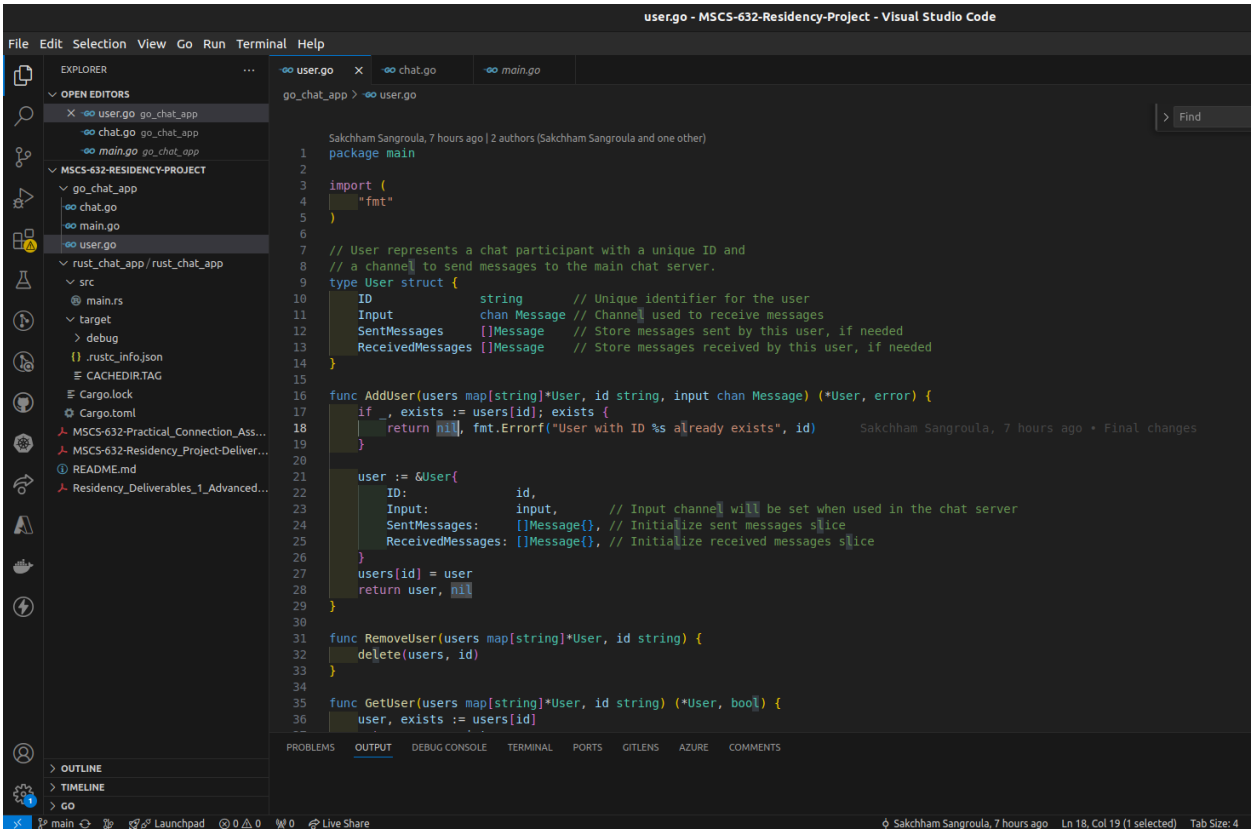
Syntax & Type System

- Rust has a more verbose, strict syntax with an emphasis on safety.
- Go uses a minimalist, beginner-friendly syntax that focuses on simplicity and readability.

Screenshot Placeholder: Rust locking and async blocks vs Go goroutines and channels




chat.go - MSCS-632-Residency-Project - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
OPEN EDITORS
user.go go_chat_app
chat.go go_chat_app
main.go go_chat_app
MSCS-632-RESIDENCY-PROJECT
go_chat_app
chat.go
main.go
user.go
rust_chat_app / rust_chat_app
src
main.rs
target
debug
.rustc_info.json
CACHEDIR.TAG
Cargo.lock
Cargo.toml
MSCS-632-Practical_Connection_Ass...
MSCS-632-Residency_Project-Deliver...
README.md
Residency_Deliverables_1_Advanced...
user.go
chat.go
main.go
go_chat_app > chat.go
You, yesterday • Go implementation- initial commit
package main
import (
"fmt"
"strings"
"time"
)
// This file defines the Message struct and ChatServer. //
// ChatServer stores and displays messages, and supports //
// filtering by user and keyword. //
// Message holds the content sent by a user along with metadata.
type Message struct {
UserIDFrom string // ID of the user who sent the message
UserIDTo string // ID of the user to whom the message is directed
Content string // Actual message text
Time time.Time // Timestamp of when message was sent
}
// ChatServer handles storing and displaying messages.
type ChatServer struct {
Users map[string]*User // Map of users to allow user management
Messages []Message // Slice to store chat history
}
// FilterByUser prints all messages from a specific user.
func (c *ChatServer) FilterByUser(userID string) {
// Check if the user exists in the chat server
if _, exists := c.Users[userID]; !exists {
fmt.Printf("User with ID %s does not exist.\n", userID)
return // Exit if the user does not exist
}
var sentMessages = c.Users[userID].SentMessages
for _, msg := range sentMessages {
if msg.UserIDFrom == userID || msg.UserIDTo == userID {
fmt.Printf("%s | %s -> %s: %s\n", msg.Time.Format("15:04:05"), msg.UserIDFrom, msg.UserIDTo, msg.Content)
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS AZURE COMMENTS
OUTLINE
TIMELINE
GO
main Launchpad 0 0 0 Live Share
You, yesterday Ln 1, Col 4 (1 selected) Tab Size: 4


user.go - MSCS-632-Residency-Project - Visual Studio Code
File Edit Selection View Go Run Terminal Help
EXPLORER
OPEN EDITORS
user.go go_chat_app
chat.go go_chat_app
main.go go_chat_app
MSCS-632-RESIDENCY-PROJECT
go_chat_app
chat.go
main.go
user.go
rust_chat_app / rust_chat_app
src
main.rs
target
debug
.rustc_info.json
CACHEDIR.TAG
Cargo.lock
Cargo.toml
MSCS-632-Practical_Connection_Ass...
MSCS-632-Residency_Project-Deliver...
README.md
Residency_Deliverables_1_Advanced...
user.go
chat.go
main.go
go_chat_app > user.go
Sakchham Sangroula, 7 hours ago | 2 authors (Sakchham Sangroula and one other)
package main
import (
"fmt"
)
// User represents a chat participant with a unique ID and
// a channel to send messages to the main chat server.
type User struct {
ID string // Unique identifier for the user
Input chan Message // Channel used to receive messages
SentMessages []Message // Store messages sent by this user, if needed
ReceivedMessages []Message // Store messages received by this user, if needed
}
func AddUser(users map[string]*User, id string, input chan Message) (*User, error) {
if _, exists := users[id]; exists {
return nil, fmt.Errorf("User with ID %s already exists", id)
Sakchham Sangroula, 7 hours ago • Final changes
}
user := &User{
ID: id,
Input: input, // Input channel will be set when used in the chat server
SentMessages: []Message{}, // Initialize sent messages slice
ReceivedMessages: []Message{}, // Initialize received messages slice
}
users[id] = user
return user, nil
}
func RemoveUser(users map[string]*User, id string) {
delete(users, id)
}
func GetUser(users map[string]*User, id string) (*User, bool) {
user, exists := users[id]
PROBLEMS OUTPUT DEBUG CONSOLE TERMINAL PORTS GITLENS AZURE COMMENTS
OUTLINE
TIMELINE
GO
main Launchpad 0 0 0 Live Share
Sakchham Sangroula, 7 hours ago Ln 18, Col 19 (1 selected) Tab Size: 4

5. Challenges Faced During Implementation

Rust Challenges

- **Ownership and Borrowing:** Rust's strict ownership model required a deep understanding of when and how data is borrowed or moved. We faced frequent compiler errors when attempting to access shared mutable state without proper locking.
- **Async Context Issues:** Working with tokio::spawn meant we had to ensure all asynchronous tasks were Send. This led to errors when locking Mutexes across await points, forcing us to refactor parts of the code to isolate those accesses.
- **Verbose Error Handling:** While Rust's compiler is helpful, we had to explicitly handle every possible failure scenario such as locking errors, unwraps, and input handling, which increased code length and complexity.

Go Challenges

- **Channel Coordination:** Ensuring that every goroutine properly read from and wrote to channels without leaking or blocking required careful design. A missed range or close() would crash the app.
- **Message Flow Control:** Go lacks strict enforcement of type contracts, so message passing logic could silently fail if not thoroughly tested. We had to include manual nil-checks and extra validation.

- **Managing File Structure:** Since the Go implementation was split into main.go, chat.go, and user.go, it required more effort to keep track of imports, initialization sequences, and consistent naming across files.

Shared Challenges

- **CLI Responsiveness:** Building an interactive and intuitive terminal interface while processing input and output asynchronously (Rust) or concurrently (Go) was tricky. Ensuring the prompt flow stayed user-friendly was a repeated challenge.
- **Real-Time Filtering and Logging:** After each user exits, filtering messages by user and keyword needed us to manage message storage properly and loop through historical data structures accurately without mutating the live state.
- **Feature Parity Across Languages:** Maintaining consistent functionality across Rust and Go while respecting each language's idiomatic practices was a non-trivial balancing act throughout the project.

6. Performance, Readability, and Language Experience

Performance

- Rust: Fast and safe, but complex setup
- Go: Fast and simple, suited for rapid prototyping

Readability & Maintainability

- Rust: Enforces best practices with more effort
- Go: Easy for teams to follow and maintain

Developer Experience

- Rust: Rewarding but steep
- Go: Smooth and productive

7. Conclusion & Team Reflection

Both languages allowed us to meet all functional goals but offered different experiences. Rust emphasized memory safety and concurrency control, while Go encouraged faster iterations with fewer barriers. As a team, we gained insight into choosing the right language for the right job.

8. References

Google. (2023). *Effective Go: Concurrency*. https://go.dev/doc/effective_go#concurrency

Jonsson, C., Jonsson, L., & Contributors. (2020). *Asynchronous Programming in Rust with async-std and Tokio*. https://tokio.rs/tokio/tutorial

Klabnik, S., & Nichols, C. (2019). *The Rust Programming Language* (2nd ed.). No Starch Press. https://doc.rust-lang.org/book/