Leo De Silva

A Level Computer Science

Mini Project: SpaceTraders

A TUI client for the SpaceTraders API

Contents

1 Analysis

1.1 Problem Defenition

This project aims to produce a user-friendly means to interact with the SpaceTraders API, an online space trading game that uses http requests to communicate with the server and manage a fleet of ships, however that lacks a frontend client for users to interact with. The core gameplay loop revolves around accepting loans, and performing mining operations to gather the required resources to repay those loans. Credits can then be used to improve your fleet and in turn, increase one's mining capacity – allowing larger loans to be taken for more credits.

1.2 Programming Language

The first decision to make regarding to this project is that of programming language – more specifically a choice between two stable, statically typed languages with strong module support and asynchronous runtimes. Go and Rust. The former (renowned for its simplicity and concurrency) would offer an expediated development cycle with an extensive standard library. Rust, however, offers memory safety, performance and a strict typing system – enforcing good programming practices. In addition to a superior multiplatform bundler for distributing the program as a single binary without dependencies. Furthermore, Rust has a more mature and documented ecosystem for Terminal User Interfaces (TUI) and Command Line Interfaces (CLI) – which I intend to explore as a means for the user to interact with the API. Below is the same program demonstrating registration with the SpaceTraders API written in Rust and Go respectively to illustrate their differences.

```
1   use std::collections::HashMap;
2   use reqwest::{Client, Error};
3
4   async fn register() -> Result<String, Error> {
5       let client = Client::new();
6
7       let agent = HashMap::from([
8           ("symbol", "L30_DESILVA"),
9           ("faction", "COSMIC")
10      ]);
11
12      let res = client
13          .post("https://api.spacetraders.io/v2/register")
14          .header("Content-Type", "application/json")
15          .json(&agent)
16          .send()
17          .await?
18          .json::<serde_json::Value>()
19          .await?;
20
21      if let Some(error) = res.get("error") {
```

```
22          println!("{}", error["message"])
23      } else {
24          println!("Congratulations, {}. You have been
25              registered. Please note your token.",
26              res["data"]["agent"]["symbol"]);
27          println!("{}", res["data"]["token"]);
28      }
29
30      Ok(res["data"]["token"].to_string());
31  }
32
33  #[tokio::main]
34  async fn main() {
35      let _ = register().await.unwrap();
36  }
```

```
1   $ http_prototype git:(master): cargo run
2       Compiling http_prototype v0.1.0
3         Finished dev [unoptimized] target(s) in 0.51s
4         Running 'target/debug/http_prototype'
5
6   Congratulations, "L30_DESILVA". You have been registered.
7   Please note your token:
8   "eyJhbGciO[...]QhEdECLg"
9
10  $ http_prototype git:(master): cargo run
11      Finished dev [unoptimized] target(s) in 0.07s
12       Running 'target/debug/http_prototype'
13
14  "Cannot register agent. Agent symbol L30_DESILVA has already
        ↪ been claimed."
```

The equivalent Go code, due to its simplicity, can be tedious and lengthy to write.

```
1   package main
2
3   import (
4       "bytes"
5       "encoding/json"
6       "fmt"
7       "net/http"
8   )
9
```

```go
type Register struct {
    Data map[string]interface{}
    Error map[string]interface{}
}

func main() {
    body, err := json.Marshal(map[string]string{
        "symbol": "TRUCKER",
         "faction": "COSMIC",
    })

    if err != nil {
        panic(err)
    }

    req, err := http.NewRequest(
        "POST",
        "https://api.spacetraders.io/v2/register",
        bytes.NewBuffer(body)
    )

    if err != nil {
        panic(err)
    }

    req.Header.Add("Content-Type", "application/json")
    client := &http.Client{}
    res, err := client.Do(req)
    if err != nil {
        panic(err);
    }

    defer res.Body.Close()

    data := Register{}
    err = json.NewDecoder(res.Body).Decode(&data)
    if err != nil {
        panic(err)
    }

    if len(data.Error) != 0 {
        fmt.Println(data.Error["message"])
    } else {
        var agent map[string]interface{}
        agentJson, err := json.Marshal(data.Data);
        if err != nil {
```

```
56          panic(err)
57      }
58
59      err = json.Unmarshal(agentJson, &agent);
60      if err != nil {
61          panic(err)
62      }
63
64      fmt.Printf("Congratulations, You've been successfully
          ↪   registered. Please note your access token:\n")
65      fmt.Println(agent["token"])
66   }
67 }
```

Due to its stability, performance, security and mature, centralised ecosystem I will use rust for this project. Although Rust's somewhat convoluted approach to asynchronous programming will require careful design considerations as to not introduce bugs. And it's combination of a restrictive borrow checker and strict compiler enforces good programming practices and memory safety, with verbose handling of all potential errors mitigting the risk of crashes and ensuring programs are always stable, performant and reliable.

1.3 API

A web API (Application Programming Interface) is a set of standard protocols to interact with a web server. The Space Traders API offers HTTP end-points with which programs can access, and in turn: play the open universe trading game. Actions are performed via http requests to the Space Traders server, and such actions can range from locating all available shipyards in a system (SpaceTraders [2]):

```
1     curl 'https://api.spacetraders.io/v2/systems/:
        ↪  systemSymbol/waypoints?traits=SHIPYARD' --header '
        ↪  Authorization: Bearer INSERT\_TOKEN\_HERE'
```

To selling ship cargo:

```
1 curl --request POST \
2  --url 'https://api.spacetraders.io/v2/my/ships/:
       ↪  miningShipSymbol/sell'
3  --header 'Authorization: Bearer INSERT_TOKEN_HERE' \
4  --header 'Content-Type: application/json' \
5  --data '{
6    "symbol": "IRON_ORE",
7    "units": "100"
8   }'
```

However, there are 2 versions of the SpaceTraders API in production, the complete V1, and the alpha V2 release. The former is a simpler system, more mature, and has a wider variety of existing frontend clients to learn from. Yet lacks much of the functionality of the second release. Whereas V2 consists of a larger universe, a wider breadth of features, and comprehensive documentation. (SpaceTraders [2]) Thus, my project will utilise the V2 API specification albeit hesitant of potential bugs due to the alpha nature of its release.

1.4   Existing Systems

There are various existing frontends that use either a command line interface (CLI), or a graphical user interface (GUI) to interract with the SpaceTraders API. A command line interface (as illustrated below) allows for expediated development, however can appear confusing, intimidating, and untintuitive without experience, wheras a GUI whilst visually intuitive, offers less flexibility and a slower development cycle. A common thread between all clients is the manner in which systems distill the complex http queries into a simpler interface when interacting with the server.

Trade Commander (DotEfekts [1]) is a CLI client for the SpaceTraders API that uses a split screen approach where commands are input in the console on the left and their output shown graphically through the display on the right. Trade Commander thus merges the flexibility of textual interfaces with the intuitiveness of a GUI. It offers a dashboard layout where commonly relevant information can be accessed at a glance: such as ones fleet, star map, owed loans, and trade market. This has the advantage of being efficient to use, however comprises a considerable learning curve when initially getting started with Space Traders.
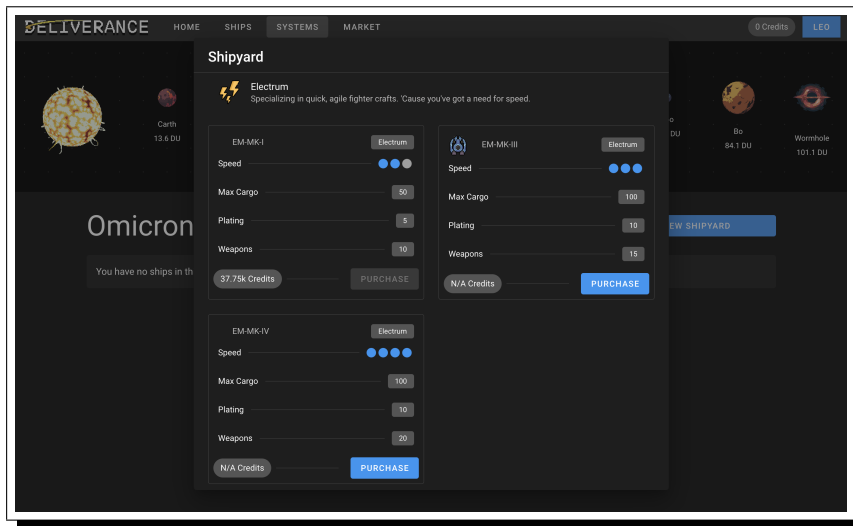
Deliverance (Stumblinbear [3]) took a different approach, opting for a graphical UI, synthesising the http requests required to interact with the API into a series of menus and buttons that are intuitive to work with. Stumblinbear has organised the available operations into 4 categories: Home, Ships, Systems, and Market. This has the advantage of being comparatively easy to work with, however will slow down development due to the resources needed to create a graphical interface and limits the flexibility of the system when adding new functionality.

In conclusion, I will look at approaching my frontend similarly to that of Trade Commander, with an interspersion of textual and graphical elements, creating a more 'retro' aesthetic that suits the nature of the game.

2 Design

3 Technical Solution

4 Testing

5 Evaluation

References

[1] DotEfekts. *TradeCommander*. URL: https://tradecommander.dotefekts.net/. (accessed: 04.03.2024).

[2] SpaceTraders. *SpaceTraders API*. URL: https://spacetraders.io/. (accessed: 04.03.2024).

[3] Stumblinbear. *Deliverance*. URL: https://deliverance.forcookies.dev/. (accessed: 04.03.2024).