Leo De Silva

A Level Computer Science

Mini Project: SpaceTraders

A TUI client for the SpaceTraders API

2024, St Albans School

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.

To play Space Traders you must first register yourself a callsign e.g. (L3O, TR4D3R etc..) which identifies an agent. Contracts, ships, and credits are all associated with an agent identity. The Space Traders universe is composed of systems and waypoints. Waypoints are locations within a system such as a planet, moon or asteroid, and consist of a type, x-y coordinates, and set of features such as a shipyard or marketplace. Ships can navigate between waypoints or warp between systems. An agent earns credits by taking contracts with a deadline and requirements for completion e.g. (deliver "205 IRON_ORE" to destination: "X1-QT13-H53" by "2024-03-15T12:04:25.707Z").

To complete a mining contract, you must locate an asteroid containing the required ore, and send your purchased mining droid to mine the resource. Cargo can then be sold at marketplaces to earn credits and offload cargo, with variable market rates for each resource depending on the volume being traded; which affects both the stability and magnitude of the market price.

Since SpaceTraders lacks a frontend for users to interact with, actions are performed via http requests sent to the Space Traders server. For example, this request to navigate to the waypoint X1-QT13-H53:

```
curl --request POST \
 --url 'https://api.spacetraders.io/v2/my/ships/L30_DE-1/
    ↪ navigate' \
 --header 'Authorization: Bearer eyJ[...]pQw' \
 --header 'Content-Type: application/json' \
 --data '{
    "waypointSymbol": "X1-QT13-H53"
  }'
```

My project will be to produce a frontend client to interact with the Space Traders API in a user-friendly manner, and will be targeted towards someone with at least a basic understanding of the command line – thus comfortable using commands to manage their fleet. However new to the game of Space Traders.

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.

1.2.1 Prototyping

The prototypes below go through the process of registering an agent, and handling any potential http errors – or the case wherin an agent symbol has already been taken. The agent data is stored in a HashMap with attributes corresponding to their respective json keys, and which is later serialised into the body of a http POST request. Both languages use asynchronous programming techniques to ensure the response has been recieved before processing resumes, (`async` in rust, and `defer` in go). The response is then deserialised into a HashMap and is checked for errors (in which case the applicable error is neatly printed) – else the agent's token is output to the console and the agent has been registered successfully.

```
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.",
```

```
            res["data"]["agent"]["symbol"]);
        println!("{}", res["data"]["token"]);
    }

    Ok(res["data"]["token"].to_string());
}

#[tokio::main]
async fn main() {
    let _ = register().await.unwrap();
}
```

```
$ http_prototype git:(master): cargo run
    Compiling http_prototype v0.1.0
      Finished dev [unoptimized] target(s) in 0.51s
      Running `target/debug/http_prototype`

Congratulations, "L30_DESILVA". You have been registered.
Please note your token:
"eyJhbGciO[...]QhEdECLg"

$ http_prototype git:(master): cargo run
    Finished dev [unoptimized] target(s) in 0.07s
     Running `target/debug/http_prototype`

"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.

```
package main

import (
    "bytes"
    "encoding/json"
    "fmt"
    "net/http"
)

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

```
14
15 func main() {
16     body, err := json.Marshal(map[string]string{
17         "symbol": "TRUCKER",
18         "faction": "COSMIC",
19     })
20
21     if err != nil {
22         panic(err)
23     }
24
25     req, err := http.NewRequest(
26         "POST",
27         "https://api.spacetraders.io/v2/register",
28         bytes.NewBuffer(body)
29     )
30
31     if err != nil {
32         panic(err)
33     }
34
35     req.Header.Add("Content-Type", "application/json")
36     client := &http.Client{}
37     res, err := client.Do(req)
38     if err != nil {
39         panic(err);
40     }
41
42     defer res.Body.Close()
43
44     data := Register{}
45     err = json.NewDecoder(res.Body).Decode(&data)
46     if err != nil {
47         panic(err)
48     }
49
50     if len(data.Error) != 0 {
51         fmt.Println(data.Error["message"])
52     } else {
53         var agent map[string]interface{}
54         agentJson, err := json.Marshal(data.Data);
55         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     }'
```

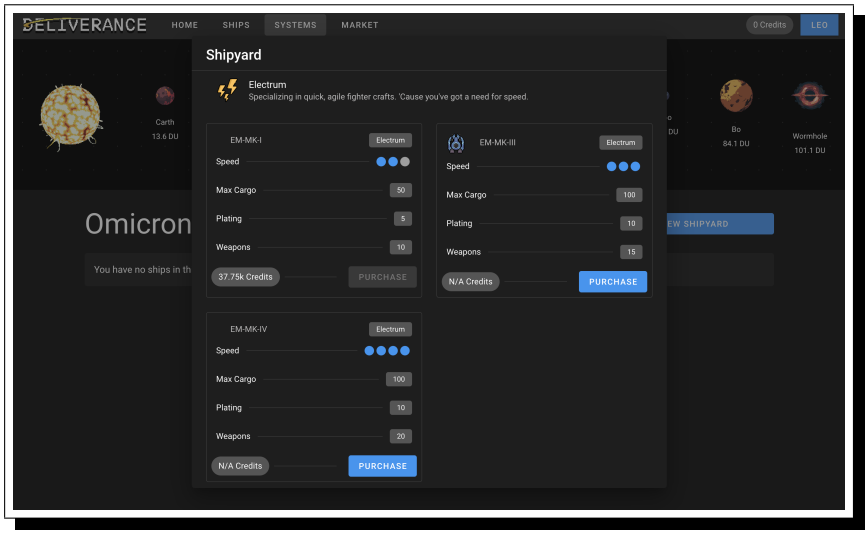
However, there are 2 versions of the SpaceTraders API in production, the complete V1, and the alpha V2 release. The former is more stable due to its maturity, and has a wider

variety of available resources: including frontend clients and tutorials to learn from. Yet it lacks much of the expanded functionality of the second release and is no longer actively maintained. Whereas V2 consists of a larger universe, a wider breadth of expanded features, and more 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 thus utilising the respective advantages of both GUIs and CLIs: allowing for both a simple and easy to understand visualisation of your fleet, and complex commands that can be entered with comparative ease. Furthermore this creates a more 'retro' aesthetic that suits the nature of the game.

1.5 Client Proposal

My client for this project will be someone with at least a basic understanding of the command line, and new to the game of SpaceTraders. The set of features I will propose is as following:

1. A Terminal User Interface (TUI) comprising of a command line and graphical interface.
2. The ability to register or login to an account using an access token.
3. The ability to purchase ships and manage one's fleet.
4. The ability to take out and repay loans.
5. The ability to manage one's mining operations.
6. The ability to visually display a map of the star system.
7. The ability to query & display data logically *(e.g. sort available ships by price)*

My system will use integrate aspects of both a CLI and GUI. The console permits the flexibility of input that is limited by a GUI, and the graphical interface can display the results of such commands or other relevant information to the user, including but not limited too: a map of the star system, the progress of ongoing mining operations, any pending loans etc... My system will include means of querying data from the API which can be sorted and

displayed according to the user's requirements, such as ordering loans by their reward, or displaying all unsold ships in the system.

1.5.1 Client Interview

I will interview my client about their experience playing Space Traders, and gather their thoughts on the frontends they have used. Here are the questions I intend to ask:

1. Have you ever played a game of Space Traders?
2. Which features of your client(s) did you like?
3. Which features of your client(s) did you dislike?
4. Which part of the game did you find the most difficult to understand? And what could have helped you understand it better?
5. Did you find a graphical interface improved your understanding of the game?
6. Would you have found a tutorial system beneficial to your enjoyment when first starting out with Space Traders?

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).