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     }'
```

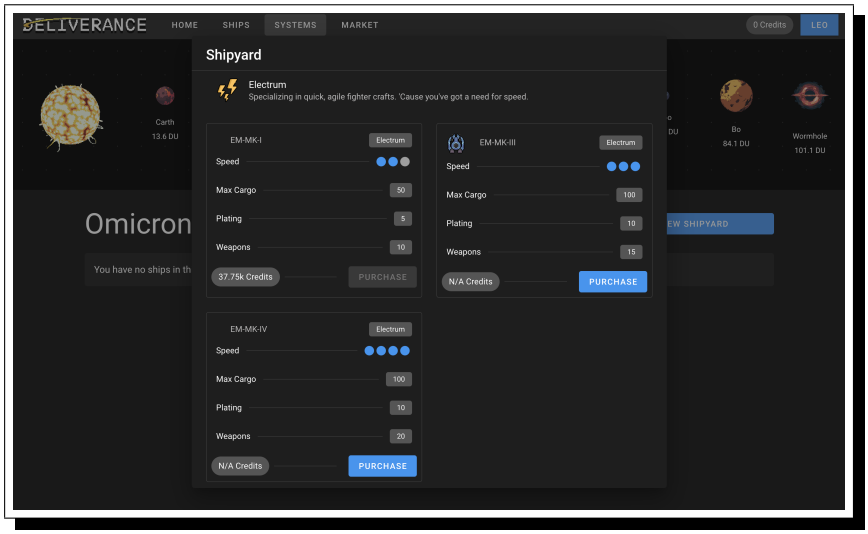
However, there are 2 versions of the SpaceTraders API in production, the complete V1, and the alpha V2 release. The former is more stable due to its maturity, and has a wider

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.

    1.1 The CLI allow the user to input commands and manage their fleet, as well as displaying their respective output in a neatly organised manner. An example use of how one might use the CLI to direct a ship with `ship_id: 2` to `waypoint: X1-QT13-H53` would be `ship fly --ship_id 2 --waypoint X1-QT13-H53`

    1.2 The output of commands will either be communicated textually through the console, or by updating the graphical display. An example command of the former type may be listing all ships present in a particular shipyard, and for the latter – moving between systems will update the star map.

2. The ability to register or login to an account using an access token.

    2.1 The user will be welcomed with a prompt to either login to an existing account (by providing their access token), or to create a new account.

2.2 Should they opt to create a new account, they will be presented with another prompt to enter their desired agent symbol, and the case wherin a symbol has already been taken will be handled.

3. The ability to purchase ships and manage one's fleet through various commands including:

    3.1 Displaying a list of waypoints or systems within reach of a ship - taking into consideration the ship's speed and fuel level.

    3.2 Navigating between available systems and waypoints, this will take into consideration the fuel of the ship and will update the graphical display to show both the current system/waypoint and previously visited waypoints.

    3.3 The ability to view the contents of a shipyard, and purchase the desired ship with credits.

4. The ability to take out and repay loans.

    4.1 With a command, you will be able to view all availabe loans, their expiration date, resource required, and credits rewarded.

    4.2 Similarly, you will also be able to accept loans and view your current loan through the graphical interface.

5. The ability to manage one's mining operations.

    5.1 You will be able to view the cargo of any paritcular ship, and the being used to carry it.

    5.2 Ship's will be able to mine resources from asteroids and add the resource to their cargo – the ship exceeds its weight limit.

    5.3 You will be able to sell cargo at various markets throughout the universe for market prices and collect the credits.

6. The ability to visually display a map of the star system.

    6.1 The graphical display will show the current system and all of its waypoints, as well as indicating the locations of any ships in your fleet and previously visited locations.

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?
   "I have played through the tutorial, to get an idea of what the game is. But I'm not a regular player. I still don't understand how most of the systems actually work.

2. Which features of your client(s) did you like?
   "I'm already pretty used to a command line, so I liked how quick it was to anything done. There was a command for everything, and they displayed the results in a nice and organised manner (2.2, 2.3). I also thought the design suited nature of the game pretty well – like one of those 80's text adventure games with the green and black terminal font. (1.1.1)"

3. Which features of your client(s) did you dislike?
   "I didn't like how difficult it was to get started, the HELP command didn't even list all the available commands, I needed to have the documentation open on the side to get anywhere! I also didn't like how it made you input your token for every single request. (2.1.1)"

4. Which part of the game did you find the most difficult to understand? And what could have helped you understand it better?
   "I still don't understand how complicated it is to get anywhere, you have to set a flight mode, dock your ship – and it becomes tedious to get anywhere in a reasonable time. (2.3.1)"

5. Did you find a graphical interface improved your understanding of the game?
   "I liked seeing the place's I'd been, and it gave a nice visual way of seeing how far i'd come since I started. Although I found that just the starmap was a little dull, especially when it was nearly empty. I liked how the view changed based on different commands, like to show the available ships or loans. (1.3)"

6. Would you have found a tutorial system beneficial to your enjoyment when first starting out with Space Traders?
   "I think SpaceTraders own tutorial is pretty overwhelming, but if the frontend is well intuitive its pretty maneagable. I think the main thing is, especially when you're using a CLI, having a quick breakdown of all the comamnds you can use, and the parameters they require goes a long way."

7. Which of the above clients appeals the most to your tastes? And what did you like about it?
   "I mean, I used the second one because I liked the way it looked. But it's quite unstable and opaque. I think if i'd started with the first one I might've understood the game a little better."

1.6 Objectives

1. **Interface & Interaction**

   1.1 The program should run as a Terminal User Interface (TUI). (*1.4 Existing Systems*)

      1.1.1 The program should be designed in the style of an 80's text adventure game with textual interfaces.

1.2 A CLI console should be used to enter commands and manage one's fleet. (*1.4 Existing Systems*)

1.2.1 The program should be able to take, sanitise and process user input, distilling it into a series of http requests required to complete the desired task.

1.2.2 The program should be able to serialise the results of such requests and communicate any relevant information textually to the user through the console.

1.3 Information regarding the star system and ongoing operations should be communicated both textually and through a graphical interface. (*1.5.1 Client Interview*)

1.3.1 The graphical display should show a map of visited waypoints, as well as information about the current waypoint.

1.4 The program should support the SpaceTraders V2 API specification (*1.3 API*)

2. **Functionality**

2.1 Users should be able to register a new agent, or login to an existing agent with an API token. (*1.1 Problem Defenition*)

2.1.1 Should the user register a new agent, the program should store their API token to streamline future requests.

2.1.2 A prompt should be shown for the user to enter their agent symbol and the program should handle the case wherin a symbol is already taken.

2.2 Users should be able to takeout and repay their loans as well as viewing their existing loans through the console. (*1.1 Problem Defenition*)

2.2.1 When queried, a textual list of all available loans with their respective reward and resource required should be shown. `loans --list`

2.2.2 The user should be able to take out a loan, and repay such a loan through a series of console commands.

2.2.3 The resource required, and reward for completion of the current loan should be displayed to the user with a single command. `loans repay --loan_id 0 -- resource IRON_ORE --ammount 5`

2.3 Users should be able to manage their fleet, browse, and purchase ships as well as carrying out mining operations. (*1.1 Problem Defenition*)

2.3.1 Users should be able to navigate & dock their ships to a waypoint as well as viewing a list of all waypoints within warping distance with the current fuel level through commands such as `ship set-flight CRUISE --ship_id 2` or `waypoints --list`.

2.3.2 Users should be able to list the contents of a particular shipyard when docked there – and view with a brief overview of the price, speed, defences, weapons and capacity of any ships.

2.3.3 Users should be able to purchase a ship from a shipyard and manage the state of all the ships in their fleet.

2.4 Users should be able to view a visual map of the star system & visited waypoints. (*1.5.1 Client Interview*)

2.4.1 The display must show the state of your current system with a bolder marker used to indicate a waypoint with a ship present, a dim grey marker an unexplored waypoint, and a lighter grey waypoint a previously visited one.

1.7 Data Modelling

Maintaining program state through a series of compound data structures – representing various aspects of the system such as an agent, ship or loan - is vital to creating a program that achieves the objectives outlined above. JSON responses from http queries, must be parsed into organised structs with which the program can interact through a series of predefined methods & public attributes. Structs must be created for Loans, Ships, Agents, Waypoints, & Systems, the struct for the Agent class might be as described below:

```
Agent:
    * The agent's API access token.
    * An array of the agent's previously visited waypoints.
    * The agent's current loan ID \& resource required for
        ↪ completion.
    * The ID's of all the agent's owned ships.
    * The number of credits possessed by an agent.
```

A similar description for the ship, and waypoint class would be as follows:

```
Ship:
    * The ship's flight mode (CRUISE, BURN, DRIFT, STEALTH).
    * The ship's waypoint marker.
    * The ship's current fuel level.
    * The ship's current cargo.
    * The ship's maximum weight, and the current weight of
        ↪ cargo.

Waypoint:
    * The waypoint's x,y coordinates.
    * The waypoints symbolic name.
    * Traits for that particular waypoint e.g. (MARKETPLACE,
        ↪ SHIPYARD).
```

There will be a `GameState` struct containing references to the objects defined above, and will be passed by reference as a parameter to functions, in turn, managing the state of the program through the centralised structure.

2 Design

The system will be a frontend for a user to interact with the SpaceTraders API through a console prompt and graphical interface. And is described by the structure chart below:

Prompt user to register or login with an API token
User Input
Navigate
List available waypoints
Query waypoints from API
Display the name, coordinates and notable features of the waypoints.
Navigate to a waypoint
Get waypoint coordinates from user input
Send http request to maneuvre the ship
Extract Resources
Survey waypoints for extractable resources
Dock at a location given by user input
Perform a http post request to extract resources
Market
List market prices
Query market prices from the API
Display the list of prices to the user through the console
Sell Cargo
Get resources & their quantities to sell from user input
Send http post request to sell user cargo
Update ship weight & user credits
Loans
List loans
Query loan data from the API
Display the list of available loans, their reward and resource through the console
Accept Loan
Get LoanID from user input
Send http request to accept the loan
Update the current loan shown through the interface

2.1 Data Design

Data entering the system will be queried from the SpaceTraders API, and the JSON responses from http requests will have to be parsed into a series of structs which can be passed as parameters to maintain state. A `GameState` class will hold references to the other classes required for API calls, or communicating information to the user and will be defined as below. The waypoints vector will be used for displaying a graphical star map on the right of the screen, and the Agent reference, for storing the symbolic name and token key for future API requests.

```
1 struct GameState {
2     agent: &Agent,
3     waypoints: Vec<Waypoint>,
4 }
```

The Agent class will store all account information that may be displayed through the interface or required for API calls, including: the agent's symbolic name, API token, current loan, and credits. It also contains a vector of Ships composing the fleet, thus mitigating the need for querying the API every time information is requested – and the structs only have to be rebuilt when performing modifying requests – thus reducing latency by minimising requests.

```
1 struct Agent {
2     token: &str,
3     symbol: &str,
4     credits: u32,
5     curr_loan: &Loan,
6     fleet: Vec<Ship>,
7 }
```

The Agent class contains references to both the `Ship`, and `Loan` classes defined by the specification below. Both will be created to parse the json response of a request into an organised structure, and stored to later be queried when users request information on their fleet or loans.

```
1 enum FlightMode {
2     CRUISE,
3     BURN,
4     DRIFT,
5     STEALTH,
6 }
7
8 struct Ship {
9     flight_mode: FlightMode,
```

```
    curr_waypoint: &Waypoint,
    fuel: i32,
    curr_weight: u32,
    max_weight: u32,
}

enum Resource {
    IRON_ORE,
    COPPER_ORE,
}

struct Loan {
    reward: i32,
    resource: Resource,
    expiration_date: DateTime,
}
```

A similar interface definition of the `Waypoint` struct can be created. Instances of these will be stored in a vector attribute of the `GameState` struct, and used to visually display a map of the star system, as well as to be referenced by ships to indicate their current position.

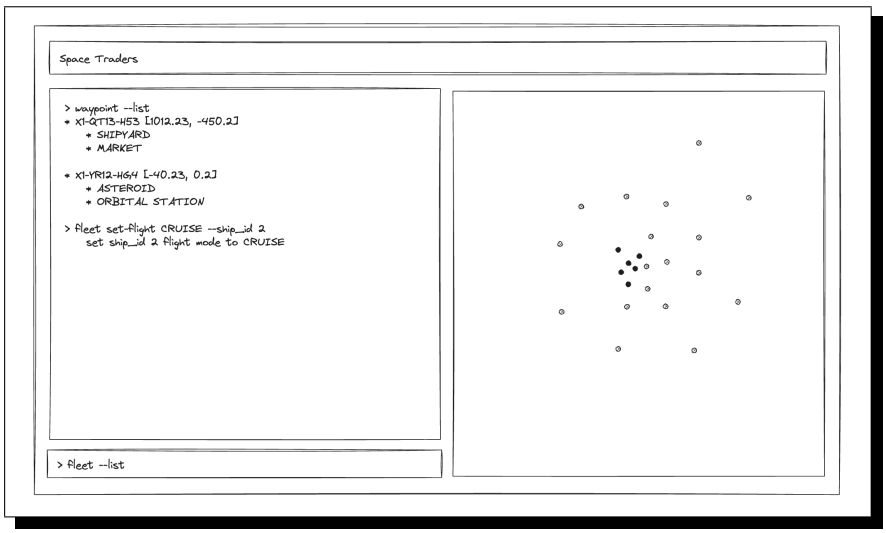
```
enum Trait {
    SHIPYARD,
    MARKET,
    ASTEROID
}

struct Waypoint {
    symbol: &str,
    coordinate: [i64; 2],
    trails: Vec<Trait>,
}
```

These data structures will store the parsed JSON from the SpaceTraders requests, and will be passed around as state through references from the `GameState` struct which will display the required data contained within these structs to the user.

2.2 Screen Layouts

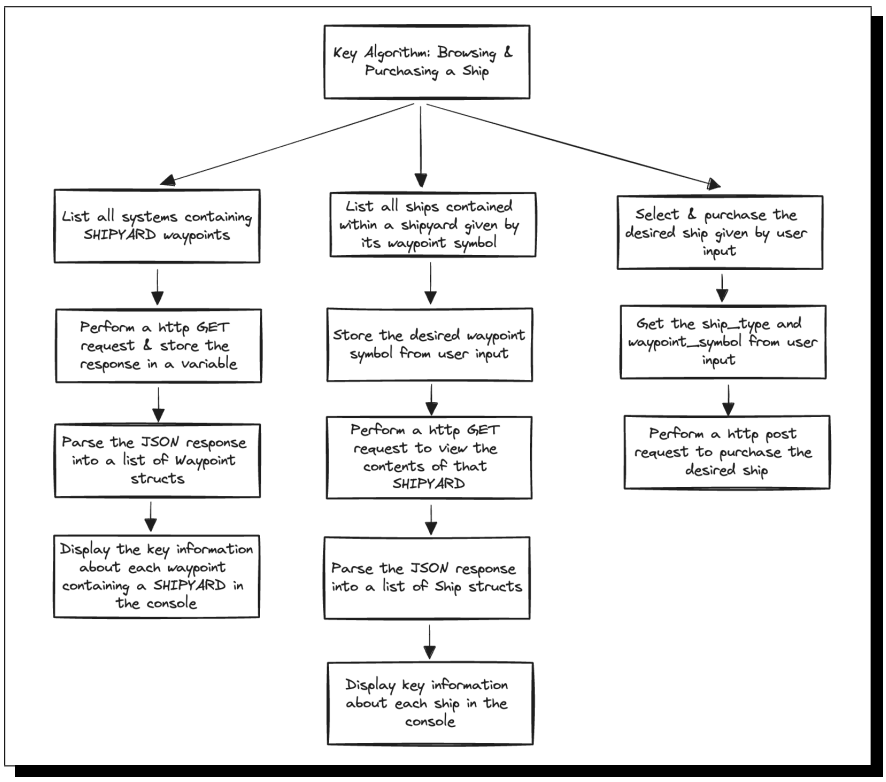
The Terminal User Interface (TUI) will be split into 4 main components, the header – containing the application title, agent symbol, and current loan; the console - containing a history of previous commands and their textual output; a prompt - where the user will enter commands; and the graphical display - where information regarding the SpaceTraders star system such as waypoints and ship locations can be communicated.



2.3 Key Algorithms

2.3.1 Managing & Purchasing a Ship

The first algorithm will handle the process of purchasing a ship - and consist of locating nearby shipyards, listing the ships contained within a particular shipyard given by its waypoint symbol, and purchasing the desired ship from said shipyard. The algorithm can be decomposed into the following tasks:



2.3.2 Listing Waypoints containing a Shipyard

The first part of the algorithm consists of performing a parameterised get request, setting `shipyard` as a trait required for the JSON of such a waypoint to be returned. The algorithm

will then parse this data into a structure, and print it to the console. The format wil be the waypoint's type, symbol, coordinates and then a list of traits e.g. (SHIPYARD, VOLCANIC, etc..). The following pseudocode will represent this algorithm:

```
PROCEDURE ListWaypointsWithTrait(STRING trait)
    response <- GET https:/api.spacetraders.io/v2/systems/X1-
        ↪ ZB92/waypoints?traits={trait}

    FOR waypointJSON IN response DO
        waypoint = Waypoint::FROM(waypointJSON)

        OUTPUT "{waypoint.symbol} - {waypoint.type}, ({
            ↪ waypoint.x}, {waypoint.y}):"
        FOR trait in waypoint.traits DO
            OUTPUT "*{trait.symbol}\n"
        END
    END
ENDPROCEDURE
```

2.3.3 Listing the contents of a Shipyard

The second part of the algorithm will list the contents of a particular shipyard (given by user input), and will involve the same process of parsing the JSON repsonse into a series of structs, with which an easy to digest decomposition of all the ships and their features can be displayed through the console.

```
FUNCTION ListShipyardContents() RETURNS Ship[]
    waypoint <- INPUT "Enter a waypoint symbol:"
    response <- GET https://api.spacetraders.io/v2/systems/X1
        ↪ -ZB92/waypoints/{waypoint}/shipyard'

    Ships <- []
    FOR shipJSON in response["ships"] DO
        ship <- Ship::FROM(shipJSON)
        OUTPUT """
        {ship.type} {ship.name} {ship.price}
        {ship.description}
        """
        Ships.push(ship)
    END

    RETURN Ships
ENDFUNCTION
```

2.3.4 Purchasing a ship from a Shipyard

Finally, after the the contents of a particular shipyard has been listed and the user has made a decision on which to purchase, they will be prompted for the type of the ship they wish to purchase. Then before the post request is made, the user's credits will be checked against the price of the ship - and a suitable message will be output if they do not have the funds. Else, a http post request will be sent to the SpaceTraders API to purchase the ship.

```
1  PROCEDURE PurchaseShip(GameState)
2      ShipType <- INPUT "Enter the ship you wish to purchase:"
3      IF GameState.agent.credits < ships["ship_type"].price
4          OUTPUT "You do not have sufficient funds to purchase
               ↪ this ship"
5      ELSE
6          POST --url 'https://api.spacetraders.io/v2/my/ships'
7          --header 'Content-Type: application/json'
8          --data '{
9             "shipType": "{ShipType}",
10            "waypointSymbol": "{Waypoint}"
11          }'
12     ENDIF
13 ENDPROCEDURE
```

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).