Front End/User Interface:
The Front end is split into two components:
Navigation bar:

- This is where the statuses will be updated such as:
    - whose turn it currently is
    - Strengths of two territories that are currently engaged
    - If player is attacker, how many die to choose
    - Attack confirmation
    - results of die role
- Below will be the card a player has

Map:

- Using AMMaps library to simulate the map
- Territories:
    - represented by circle
    - Each lat/long coordinate is in an array:
        - Ex: const CHINA = [36.897568, 103.375284];
    - Each “territory data structure” is a dictionary
        - Ex:

```
let CHINADATA =   {
        "latitude": CHINA[0],
        "longitude":  CHINA[1],
        "type": "circle" ,
        "color": "#CC0000",
        "scale": 0.5,
        "label": "China 5",
        "labelShiftY": 2,
         "zoomLevel": 5,
        "title": "China",
        "id": "China",
         "description": "Occupied by Player 1. Soldiers: 5"
};
```

        - This is then put into an array in which is then attached to the variable “image” in the AMCharts.createChart function
- Borders and adjacency:
    - AMMaps has a line attribute
        - It’s an array in which each index contains a dictionary with two arrays
        - The first array is associated with lat start and lat end which the second array is associated with long start and long end
        - Each dictionary is assigned to a variable
        - Example: of one index in the line array

```
let AB-OT = {
        "latitudes": [ ALBERTA[0], ONTARIO[0]],
        "longitudes": [ ALBERTA[1], ONTARIO[1] ]
```

“color”: blue

}

- Reflecting changes
    - Each gui element will have a function that reflects its changes
    - Each territory will have a click event and execute certain elements based on the condition: (ex: player is currently attack vs not attacking)
    - Since whether or not these onclick function works is whether the button appears or not, we won’t have to worry about players clicking
    - Each map sub data structure mentioned above will be changed when changes attack and be replaced in the map data structure
        - Lines: potential route of attack will change to attacker’s color
        - Territory color will change when territory is taken over by another player
        - Label: (territory followed by number of soldiers)
    - Essentially we will have a bunch of change<element> functions
    - These methods will be called based on the update object received from the backend via sockets (see networking).

Networking:

Package Client:

The Client package manages organization of player sessions, and communication between the back end and the front end. It works by taking in move information from a player session, and passing it to the back end. The back will then a return a game update. This update will return whether or not the move requested by the front end was valid, the results of that move and what needs to be displayed on the board as a result, and what moves are now available for the player who just moved.

- Matches Class:
    - Manages and organizes all sessions into game lobbies. Holds onto player sessions, and match classes, the internal representation of a lobby.
    - Connected Method: Add this new session our sessions queue, and give it a unique id. Send update messages to this session informing it of available match lobbies that it needs to display.
    - Closed Method: Remove this session from our sessions queue, and update the lobby menu to no longer list that session as being present.
    - Message Method: Respond to different message from the GUI based on the message type. Can make calls to join_player, start_game, and create_lobby.
    - Join_player Method: Store a player id in the selected lobby. Update other lobbies in case this player changed the lobby they wanted to be in. Asks the GUI to see if the selected lobby is now full, and a match can begin.
    - Start_game Method: Begin a match and do not allow any new sessions to join the lobby.

        - Create_lobby Method: Creates a new lobby with the size and name inputted by the player.

- Match Class
    - Manages and individual game of risk. Sends and receives messages to update the GUI and ask for move updates from the back end.
    - Has getters for the match name, match id, current number of members in lobby, maximum lobby size, and player ids. Can add and remove players.
    - Start Method: Causes the match to no longer add or remove players. Creates a risk game, and asks it for an update for the start of the match. As this is the first stage of the game, this update will be along the lines of “player x can place an army”. The match begins accepting messages for player moves. When a message is received, a move is sent to the back end, which will return a game update for the GUI.

Game Design:

Package Risk:

The risk package contains the data structures most fundamental to the operation of the risk game. The package is divided between classes related to execution of the game (actual gameplay) and classes controlling the logic, flow of control, and rules of the game.

-Class RiskBoard: models the game board. The board knows about the territories on it, knows which territories are in which continents, and also knows the connections between territories.

-Class RiskGame: stores the data structures, including RiskBoard. RiskGame is where gameplay occurs. The referee class regulates gameplay.

-Class RiskPlayer: each playerId generated by the network is assigned a RiskPlayer. RiskPlayers know which territories they control and which risk cards they have.

-Class Die: has a method that simulate a regular 6 sided die roll.

-Class Turn: risk is a turn based game. People alternate roles according to a formulaic procedure. The Turn class is designed to capture this pattern. A Turn object knows who owns it (the RiskPlayer) and also knows what the phase of the game is (i.e, handin cards, reinforce, attack, defend, etc.). At any point in the game, the Turn class controls what the next move should be. If someone has decided to attack, the Turn will be modified to defend and change ownership to the defender. The turn cycles based on the rules of the game and user input.

-Class Referee: The referee class works with turn to develop legal moves for the players. The referee determines the state of the turn and decides via a switch which moves are available for

which players. For example, if someone has decided to attack, the next move should be a defend move assigned to the defender. The purpose of referee is double verification of user input. The referee modifies the GameUpdate class, which sends information to the GUI about what should be allowed. A GameUpdate object might specify that only the attacker should be able to access certain input buttons, and the other players should have their input disabled. The GameUpdate preemptively forces the user to select correct input based on the rules of the game. The goal is to set restrictions so the user must choose a legal number of dice to roll or troops to move, for example. When the user actually selects input in the GUI, that information is sent to the backend where it is verified. This ensures a second check on user input before changes are made to the game state.

Package Move:

The move package contains the data structures necessary to define each possible type of move during risk. There are two subtypes of move classes. One type defines the actual move of the player, such as the attacking territory and the number of die rolled. The second type of move class defines all of the valid actions that one type of move can take at a given time.

Enum MoveType: This enum is used to give an id to each type of move: reinforce, turn in a card, attack, defend, claim territory and move troops.

Interface Move: Each move class implements this interface, has two methods: getPlayerId() and getMoveType(). getMoveType will be used by the networking side of the project to know which type of object it needs to communicate to the frontend.

Class ValidReinforceMove: This class defines the valid reinforcing moves. It contains the player id of the current player, the MoveType, a set of territories the player owns and the number of troops the player must reinforce with. It has a validate move method which can take in a ReinforceMove and determine if the move is within the bounds defined by this class.

Class ReinforceMove: This class defines how the player reinforced his troops. It contains the player id, and a map of territories to integers to define where and how many troops the player put on the map.

Class ValidCardTurnInMove: This class defines what cards the player can turn in and where the player can place troops on the map. It has a validate move method which validates a CardMove is within the bounds defined by this class.

Class ValidAttackMove: This class defines what attacks are valid for a player. It contains the player id, the set of territories the player owns that can attack and against whom as well as the max number of die that can be rolled for each territory that can attack. It has a method that can take in an AttackMove object and determine if it is within the bounds defined by this class.

Class AttackMove: This class defines how the player chose to attack. It contains which territory the player is attacking from and which territory the player is attacking and the number of die the player would like to roll.

Class ValidDefendMove: This class defines how many die a defending player can roll. It can validate a DefendMove object based on the bounds defined by this class.

Class DefendMove: This class defines how the player chose to defend. It contains the number of die the player would like to roll.

Class ValidClaimTerritoryMove: This class defines how an attacker can claim a territory he or she has defeated. It contains the attacking territory, the claimed territory and the max number of troops the attacker can move to his or her new territory. It can validate a ClaimTerritory object is within the bounds defined by this class.

Class ClaimTerritoryMove: This class defines how a player chose to claim a territory. It contains the number of troops the player would like to move into his or her new territory.

Class ValidMoveTroopsMove: This class defines how a player can move his troops during the end of his turn. It contains information detailing which territories are reachable from each other and the total number of troops that can be moved from one territory to another. This class can validate if a MoveTroopsMove is within the bounds defined by this class.

Class MoveTroopsMove: This class defines how the player chose to move his troops during the end of his turn. It contains the starting territory, the ending territory and the total number of troops moved.