Design Specifications Document

Ticket to Ride

Introduction

Ticket to Ride is a railway-themed board game originally published in 2004. The game has won numerous awards, including Game of the Year in its launch year. The popularity of this game has continued to grow over the years; the October 2014 report shows that over 3 million copies have been sold.

In this project, you will create an online version of Ticket to Ride. Your Ticket to Ride game will be based off of the American version of the board game. Implementation details are up to you, but the following design principles will need to be incorporated into your final product:

1. Singleton Class
2. Command Pattern
3. Plugin Pattern
4. Observer Pattern
5. Proxy/Adapter Classes
6. Facade Pattern
7. State Pattern
8. Factory Pattern
9. Chain of Responsibility
10. Visitor
11. Composition

Ticket to Ride — A Quick Overview

There are three main views for the game: (1) login/register, (2) game lobby, and (3) gameplay. Each will need to be connected to a backend server to populate the view with the correct data. The details of this backend interaction will be described later.

The following sections specify the appearance of each main view and should be recognizable in the final product. Although the images included are mockups, they contain all the crucial components, and those components must be included in the product.

Login/Register Screen (F.1.1 and F.1.2[1])

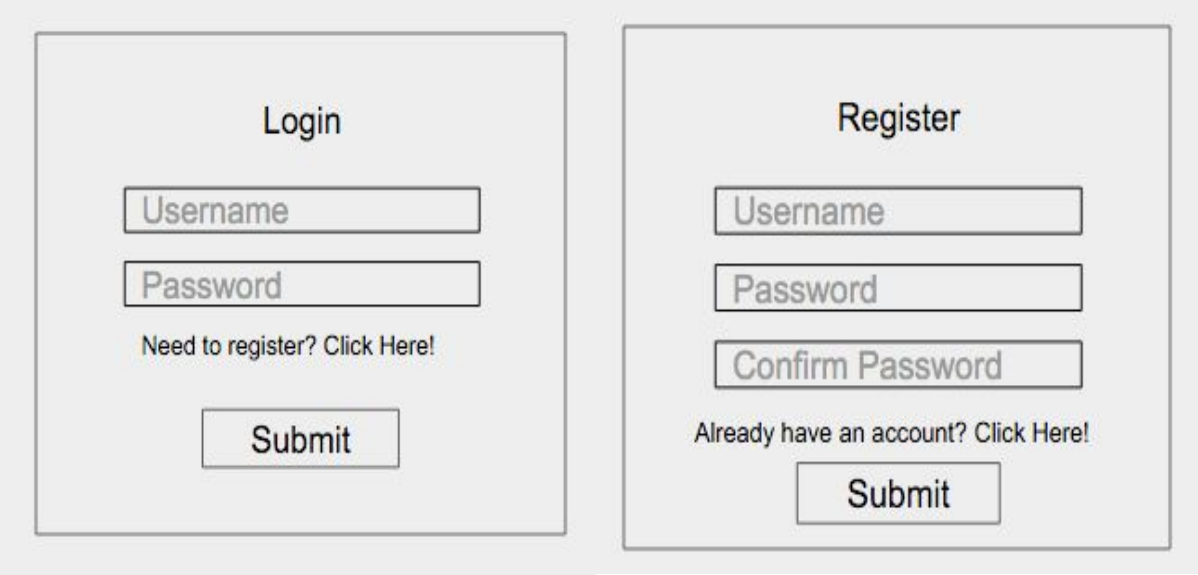


Figure 2. Login and register views

The login should only accept valid username and password combinations from users who are already registered. If incorrect information is submitted, red text will appear signifying the error, and will prompt for resubmission. This text may either appear underneath the text field, above the text fields, or in a pop-up dialogue (in the case of a dialogue, the text need not be red, but some indication of the offending text must be given).

Registering a new user should reject the registration if the username is already taken or if the password is less than 5 characters long or consists of non-allowed characters[2]. If an error occurs, red text will be shown prompting the user to use correct information and re-submit. This error text must follow the same specifications as mentioned in the paragraph above.

Once a user has logged in or registered successfully, they should be sent to the game lobby screen.

---

[1] These alphanumeric listings refer to the Ticket To Ride requirements document and specify which sections are being met in the design.

[2] See the requirements document sections F.1.1 and F.1.2 for non-allowed characters.

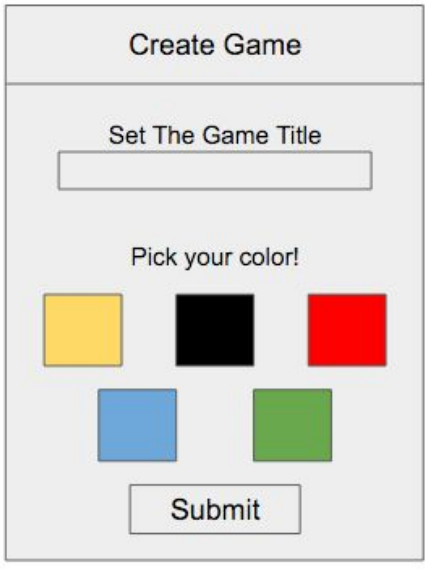
Game Lobby Screen (F.2)

Figure 3. Game lobby view

The game lobby displays a list of the current available games and the players connected with it. As long as a spot is available, the game will display the remaining colors that can be claimed while joining. To join, a user will simply click the color he/she wishes to join as. Any games that you have already joined will also be displayed to allow for switching between games. You also have the option of creating a new game by pressing the "Create Game" button.

Creating a Game (F.2.1)

Pressing the "Create Game" button will bring up an overlay that allows you to name the game and choose the color you wish to join as. After clicking submit, the game will be added to the list of available games for all other players to join.



GAME BOARD (F.3)

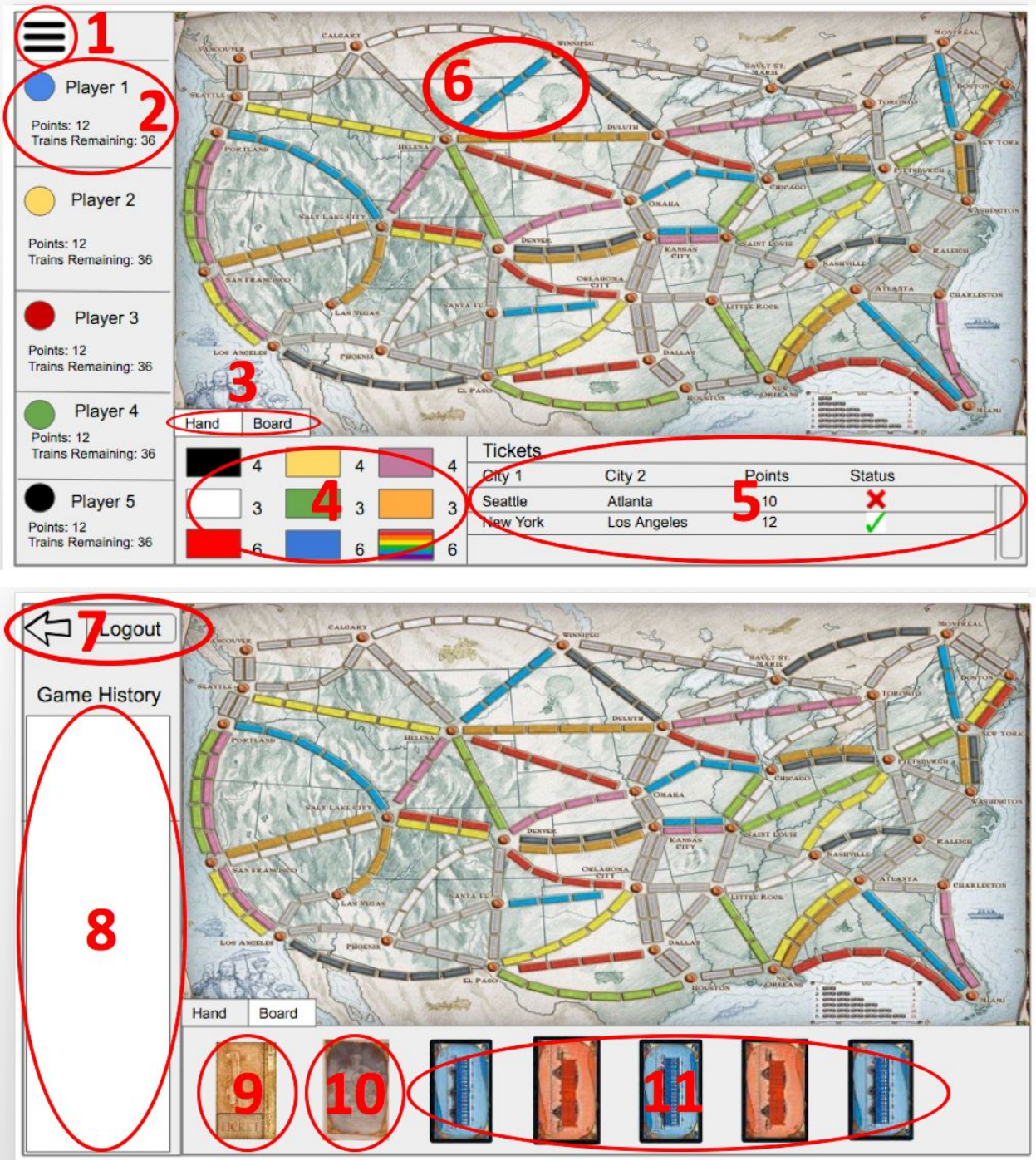


Figure 5. Game board

The main game board can be split into several major functional components. These components are:

1. **Game information menu**: The left-hand strip of the view. The game information menu contains all player summaries (next section) in an ordered list, and a button that, when clicked, replaces the game information menu with a logout/game history pane (sections 7 and 8).

2. **Player summary:** Presented in the game information menu as an ordered list. Each box contains one player summary. The player summary reports (1) The player title, being either player numbers or usernames, (2) points earned so far, and (3) the number of remaining trains for the player.
3. **Game play tabs:** Allows switching between the hand and board interaction modes. In hand mode, player train cards (section 4) and player destinations (section 5) are displayed. In board mode, the destination deck (section 9), train deck (section 10), and train yard (section 11) are displayed.
4. **Player trains:** Shows the number of train cards currently held by the player according to color.
5. **Player destinations:** Shows the destination cards currently held by the player. Each destination card includes information on start/end city, points the card is worth, and status (whether or not the destination route is completed).
6. **Purchase route:** Interactive map component that allows players to buy routes.
7. **Logout:** Logs the user out of the current game, issues the appropriate commands to the server.
8. **Game history:** Shows a text summary of the recent moves in a game.
9. **Destination deck:** Allows a user to draw new destination cards[3]. We suggest using a modal for interaction with the destination deck (see later section).
10. **Train deck:** Allows the user to draw new train cards. Cards from the train deck are face-down and unknown to all players.
11. **Train yard:** Allows the user to draw new train cards. Cards from the train yard are face-up and known to all players.

Additional modals can be used for a more detailed interaction. The use of modals is suggested for the selection of destination routes and purchasing of a city-to-city route on the board.

[3] Rules for drawing destination cards, train car cards, etc. are available in the game rules.

Selecting Ticket/Route Cards (F.3.6)

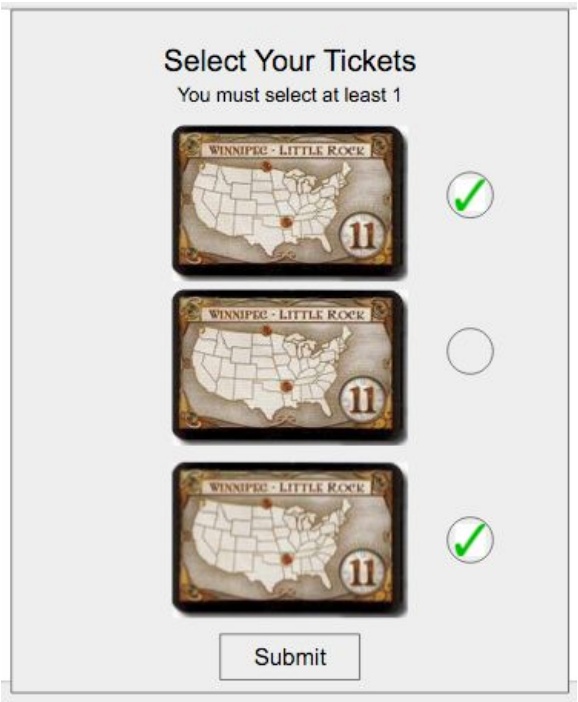


Figure 6. Select destination cards overlay.

Players can select ticket/destination route cards at two gameplay stages: (1) at the beginning phase of the game and (2) after electing to select new destination routes during his or her turn.

To begin the game, all players are presented with three destination cards. At least two of these cards must be selected before the turns can advance. All cards not selected are returned to the destination card deck.

If a player elects to select new destination routes during his or her turn, the same choice is presented with a minimum of one card. You may choose whether to require selection of the card before allowing the next turn to begin or allowing the turns to proceed until the next turn in which the player is required to make the selection before beginning his/her own turn. All cards not selected must be returned to the destination card deck.

Ticket to Ride — Buying a Route (F.3.5)

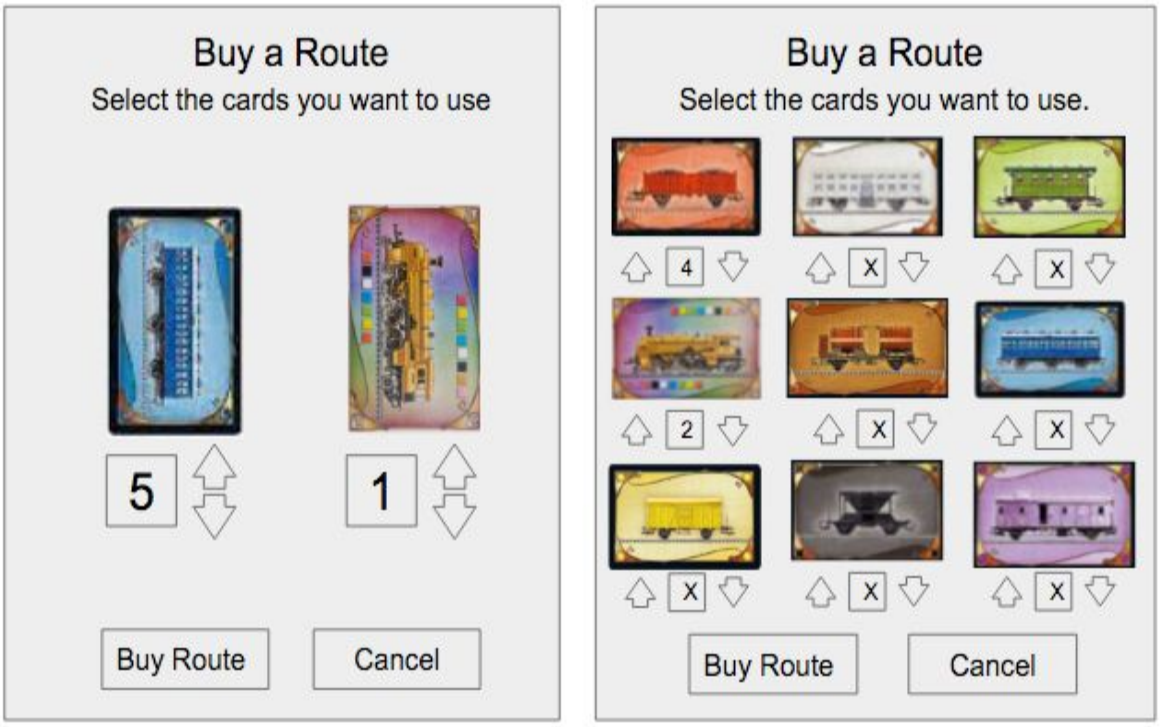


Figure 7. Buy route modal options.

The two modals shown in figure 7 are two possibilities for allowing users to purchase their route. The first option requires some logic to happen before display, but would be used for all non gray routes as they only allow for one color and wild cards. The second modal is for gray routes as they can be selected from any color. Once a user starts using a color to purchase a gray route, as shown in figure 7 with red, the other color options besides "wild" are blocked. They are blocked because gray routes can be bought by any one color, not by many colors.

Server Architecture

The server side of the Ticket to Ride game consists of two main parts: the connection components and the game model.

Server Connections

The server utilizes two different connections to communicate to the clients. Primary communication between the server and the clients occurs through a WebSocket connection. This connection handles authentication (login, register, logout), matchmaking (joining a game, creating a game, etc.), and gameplay (drawing a card, ending a turn). Due to the use of an upgraded HTTP connection, the WebSocket can employ two-way communication and therefore allow the server to push updates to the clients without prior request.

Secondary communication with the server happens through a simple HTTP REST server. This communication primarily handles distributing static information to clients upon request (such as requesting an image). These types of request

do not require the upgraded connection the WebSocket provides and allows the WebSocket to focus solely on conveying dynamic information.

WebSocket (T.1)

The WebSocket server handles connections through a custom WebSocket handler. The handler stores a synchronized static HashMap of sessions with userId keys. When a client connects to the server, the WebSocket handler stores the session in a local variable. When that session sends a login command, the handler adds that local session to the HashMap with the returned userId. Because of this, a user cannot simultaneously log into the server from multiple locations as only one session will be stored per user.

Any message sent to the server will be translated into a command object that, when executed, performs all the logic necessary to change the server appropriately. The Response objects created by these commands are sent to all pertinent users (for example, all the users in a game if a user ends his turn). This allows all users in a game to stay up-to-date with the game state.

Commands

When a message arrives at the WebSocket handler, commands are instantiated through a double factory pattern using GSON. All incoming requests have a body similar to the following:

```
{
      "command": "login",
      "parameters":
      {
            "username": "Bob",
            "password": "bob"
      }
}
```

The CommandFactory instantiates a CommandParser object using GSON's fromJson() on the entire JSON string. The CommandParser stores the command string and the entire JSON object for parameters. The parseCommand() in the CommandParser uses reflection to create the appropriate command object from the command string, once again using GSON's fromJson() on the parameters to instantiate the command with the appropriate member variables set. This pattern allows direct invocation of GSON's fromJson() without having to create custom deserializers.

Invoking the execute() requires the userID of the user invoking the command. This userID is not used in the Login and Register commands due to the userID not having been decided before the execute() call.

Each command object interacts with the server model through the ServerFacade. After performing any necessary logic, the command returns a ResponseWrapper object.

Responses

Every command returns a ResponseWrapper object from its execute(). This object takes a Response object, serializes it through GSON's toJson(), and stores it as a message. Additionally, this list holds the ids of the users who need to receive the message (usually the users in a particular game).

The Response objects themselves are Data Transfer Objects (DTOs). They are only used to represent the message that will be sent to the client. The Response class and all of its subclasses are directly serializeable through GSON's toJson() without the need for a custom serializer. The Response class itself contains several static methods to generate standard success, invalid input, and server error responses.

Other Handler Logic

The userID is not known before a user logs in, so special logic needs to occur to inform the handler of the appropriate ID when the login/register occurs. Since the commands are generated generally, the most convenient way to do this is to use the ResponseWrapper object returned from the login/register commands.

On command generation, the WebSocketHandler checks to see if the generated command is a login or register command. If so, the responseWrapper contains the userID of the session as the only entry in its target userIDs field. The handler can then extract the userID and use it to index the session in the static HashMap.

REST HTTP (T.2)

The HTTP server uses a REST architecture. On receiving a GET request, the attached handler returns the requested file. As mentioned, this is intended to primarily service image requests. No POST or other requests will be accepted.

Server Model

Any information in the Server not related to connections is stored as two member variables of the ServerFacade class: the user list and the game list.The user list holds a list of User objects containing:

- Username
- Hashed Password
- Unique User ID
- A list of games joined
- A boolean for logged in state

An auto-incremented ID is assigned to each user upon registration. This userID is used to identify the user across the server.

The game list consists of Game objects, each containing a PlayerManager and a Gameboard, as well as a history of all messages sent to the players of that game.

PlayerManager

The PlayerManager class keeps track of a list of Players in the game. Aside from this, the PlayerManager adds some functionality to the model, such as buying tracks.

The Player class itself keeps track of the state of the player at any time. Each of these Player objects is composed of a User object, allowing each player to be identified uniquely. This functionality is essential in assisting the WebSocketServer in sending messages to all players in a game. Aside from this User object, the Player object also keeps track of points scored, tracks left, player color, destination routes, current cards, and whether the player has built the longest track. This class is fully serialized when sending the full model.

GameBoard

The GameBoard class is responsible for any remaining game state not covered by the PlayerManager. Notably this includes the state of every route in the game, which train cards are visible and which are in the deck, and the deck of destination cards still left.

Of note is how the GameBoard manages the state of the routes in the game. Every route is included in a list that is not modified. The order of this list is synchronized with the client, allowing route references to be passed between the server and client as an index into this list. Aside from this, every route is also found in a HashMap of Route lists with playerID keys. This HashMap stores the routes according to ownership. Every unpurchased route left in the game is stored under the id -1. After this, every other route (the ones the players have bought) are stored under the player ID of the corresponding user. A purchase of a route results in the relevant route being removed from the still-unclaimed routes and into the list of the player who bought it.

Another interesting feature of the GameBoard is the decks of destination cards. Because of the rules of the game, all destination cards that were dealt but not selected are placed into a discard list. Once the destination card deck is empty or extremely low, the discarded destination cards are shuffled and then reinserted into the main deck. This preserves the random nature mentioned in the official rules.

Finally, the train cards are managed by the GameBoard to assure compliance with the rules of the game. All five visible cards are stored in an array and the rest of the cards are stored in a separate list. Upon receiving a DrawTrainCard command, the server performs whatever logic is needed and, in the response, returns the value of the new cards visible on the table. This allows the server the ability to automatically redraw cards if one of the rules is broken. For example, if selecting a card reveals a third rainbow train card, the server can reshuffle the visible cards into the deck and return the new visible cards.

Server API (T.4)

Pre-Game

Login:

input: {username: string, password: string}

output: {description: string}

This will check the provided credentials to allow access to the game hub. If the user is authenticated the description value in the JSON is “success”. If either the user name isn’t found or the password for that user isn’t correct the description value in the JSON is “invalid input”. If there is any internal server error the description value in the JSON is “server error”. All passwords should be salted and hashed and then compared. Only allow a user to login

one time. If a user is currently logged in and they attempt to login again the description value in the JSON is “already logged in”

Logout:

input: {}

output: {description: string}

Calling this command the server will see which socket the user is connected to and close that connection. Note that this function should not cause any issues if the user was not already logged in.

Register:

input: {username:string, password:string, email:string}

output:{description:string}

This will attempt to register the provided details into the server. The password should be hashed and the hash should be saved, not the plain text password. There should also be created a salt for that user. If everything was successful the description value in the JSON is “success”. If either the email or username provided are already registered on the server the description value in the JSON is “invalid input”. The server will also return “invalid input” if the input is less than 4 characters long and contains any non alphanumeric characters. If any other server errors occur the description value in the JSON is “server error”.

GAME HUB

SendChat:

input:{gameId:int, message:string}

output:{sender:string, message:string}

This will send a chat message to a specific game from a specific person.

UpdateJoinableGames:

input:{}

output:{games:[{gameId:int, gameName:string, players:[{username:string, color:string, playerOrder:int},{}...]},{}..], description:string}

This will get all the games on the server that are not yet started and are not yet full. That is, any game any other users can join. This does not show the games the particular user is already in. When CreateGame is called on the

server this API will also push the new game to all other connected clients, updating them of the existence of the newly created game.

UpdateUserGames:

input: {}

output:{games:[{gameId:int, gameName:string, players:[{username:string, color:string, playerOrder:int},{}...]},{}..], description:string}

Once a user logs in this API is called and returns all the games the current user is playing in. This also includes games that are not yet started, but the user is already a part of. When a user joins a game this API should continue to be pushed from server to client for any updates. This should continue to be called even when a player is in the middle of a game. (Thus someone can be sitting waiting for more players to join a game, and be playing in another game). If everything completes correctly the description value in the JSON is “success”. If there are any internal server problems the description value in the JSON is “server error”.

JoinGame:

Input:{gameId:int, color:string}

output: {description:string}

This should be called once a user clicks on a color to join an available game. This should also trigger an update to be sent to all other players in the game (by sending a UpdateUserGames) alerting everyone else that a new player has entered the game. If the player who has joined if the 5th player the game automatically begins. If all was good the description value in the JSON is “success”. If there are any internal server problems the description value in the JSON is “server error”.

CreateGame:

Input: {gameName:string, color:string}

output: {description:string, gameId:int}

This API is called when a user creates a new game. This call should also join that user to the newly created game and set their color. It will also update all other clients of the existence of the newly joinable game by sending an UpdateJoinableGames. If all was good the description value in the JSON is “success”. If there are any internal server problems the description value in the JSON is “server error”.

LeaveGame: (future)

input: {gameId:int}

output:{description:string}

If a user leaves a game (as in they say, I don't want to be in this game anymore. This does not mean they logged off), this will remove the user from the game. This is only possible when the game has not yet started. It will update all other clients by sending a UpdateUserGames as well as a UpdateJoinableGames (so everyone else knows available colors and number of people in each game).

StartGame:

Input: {gameId:int}

output {description:string, gameId:int, initialTrainCards:[string,string,....], initialDestination:[{start:string, end:string, points:int},{},....]}

This is only allowed from the person who created the game. The game should have at least two players in the game to begin. This also returns the initial train cards in a player's hand - these strings are just colors- and initial destination cards for the user to select. If all went well the description JSON value is "game has started". This is also repeated to every other user in the game. If there are not enough people the JSON value for description is "insufficient number of players" and is only sent to the person who sent the initial request. If there was any other internal server error the JSON value for description is "server error".

AddAI: (future)

input:{gameId:int, difficulty:string}

output:{description:string}

This will add an AI to the game specified. The game can not be full, and can not be started yet. If the game is full the JSON value for description should be "game is full". If the game has already started the JSON value for description should be "game already started". If there was any other server error the JSON value for description should be "server error". If everything went ok, and the AI was added then the JSON value for description should be "success"

SendClientModelInformation:

input:{gameId:int}

output:{players:[{username:string, order:int, trainsLeft:int, color:string, cards:[{color:string, amount:int},{}...]},{}...], routes:[int,int,...], cities:[{to:string, from:string, points:int},{}...]}

This is sent to update the client on login, and periodically as necessary indicating the current state of each of the user's games. The input gameId int is the game Id for the model the client wants, and the output is a game model.

GAME COMMANDS

BuyRoute:

input:{gameId:int, routeIndex:int, trainColor:string, numberOfWilds:int}

output: {description: string, gameId:int, playerIndex:int, routeIndexPurchased:int, trainsLeft:int, pointTotals:[{playerId:int, points:int}...]}

This is called when a user intends to buy a route on the map. The gameId is the game index of the game where the user wants to buy a route, the routeIndex is the index into the routes array of the route the user wants to buy. The train color is the color of the route the user intends to buy, and the number of wildcards is the number of wilds the user is using to buy the route. The route must be available for purchase, the user must have enough cards, including the wilds, to make the purchase and they must have enough trains available. Every time this is called the server should check if the player has completed a route. If so it should send out a NotifyDestinationRouteCompleted command. If the route index provided is not valid the description JSON value is “invalid route location”. If the user does not have enough trains the description JSON value is “insufficient trains”. If the route intending to be bought can’t be bought with the color intended the description JSON value is “invalid train color”. If it’s not the players turn who sent in the request the description JSON value “not your turn”. If there are any other internal server errors the description JSON value “server error”. If everything is ok and the route is bought then this API is pushed to every other client in the game.

DrawTrainCard:

input: {gameId: int, cardLocation: int}

output:{description: string, gameId:int, playerIndex:int, cardDrawn:string, availableTrainCards:[{color:string, index:int},{}...], canDrawWild:boolean}

A player calls this API to draw a train card from the available train cards. CardLocation can be an int between 0 and 5. A 5 for cardLocation indicates the top of the deck, not one of the upward facing cards. CardDrawn is the color that has been selected, and should be added the to players hand. The canDrawWild indicates with a wild can be drawn and is toggled based on if the player drew a card from the front facing cards. If all went well the JSON value for description is “success”. If the deck is exhausted the JSON value for description is “no more cards to draw”. If the player is not authorized to draw a card the JSON value for description is “draw again not allowed”. If the player attempts to draw a color card, and then attempts to draw a wild card the JSON value for description is “wild card not allowed on draw again”. And if it is not the players turn the JSON value for description is “not your turn”.

NotifyDestinationRouteCompleted:

input: N/A

output:{gameId: int, playerIndex:int, route:{start:string, end:string, points:int} }

This is pushed from the server to notify the client that a destination route card has been completed. This

should be created and sent out to the player who has completed a route, if buying a route completes a destination route.

GetDestinations:

input: {gameId: int}

output: {description: string, destinationCards:[{cityIndex1:int, cityIndex2:int, points int},{}....]}

If a player wants to get new destination cards on their turn this API is called. It should return either 3 cards or the number of destination cards left, the lesser of the two. The city index is referred to by the city to index map created. If all went well the JSON value for description is “success”. If there was any type of internal server error the JSON value for description is “server error”. If it is not the players turn the JSON value for description is “not your turn”.

SelectDestinations:

input:{gameId:int, destinationsSelected:[int,int...]}

output:{description:string, gameId:int, playerIndex:int}

The destinations selected is an array of ints that correspond to the index of the list from calling GetDestinations API. They are the destinations that the player has selected. If the game is just starting that array must have at least two elements, if the game is on going that array must have at least one element. The array can never have more than 3 elements and the elements should only ever have the numbers 0, 1, or 2. (Since the array give only had index choices of 0, 1, or 2 the client shouldn’t ever send anything but those indexes back). If all went well the JSON value for description is “success”. If there are not enough elements selected the JSON value for description is “have to select more”. If the index(s) provided are not valid the JSON value for description is “not a valid index”. If it is not that player's turn the JSON value for description is “not your turn”. If any other internal server error occurs the JSON value for description is “server error”.

TurnStartedNotification:

input: N/A

output:{gameId:int, playerIndex:int, lastRound:boolean}

When a player ends their turn this API is sent to everyone in the game indicating whose turn it currently is. It also indicated if the last round has started (triggered by someone buying a route and having 2 or fewer train cars remaining).

GameEnded:

input: N/A

output:{gameId:int, players:[{playerIndex:int, points:int, longestRouteRecipient: boolean},{}....]}

Once a game has ended this is sent to every player in the game indicating the results of the game.

GAME TEST COMMANDS:

LoadGameState:

input:{path:string}

output:{status:string}

This will take a path as a string and load that file, which is a game model in json form. If all goes will it will return a string saying “success” or if there was an error it will say “error”.

SaveGameState:

input:{gameId:int}

output:{path:string}

This will take a game id as an int and will serialize that game to json and save it on the server. The path string will be the path to the json file on the server.