Catan Server Web API Documentation

The syntactic details for interfacing with the Catan server web API are provided on the server’s Swagger page. Swagger is a tool for documentating and interactively invoking web API methods. The Swagger page may be accessed by running the server, opening a web browser on the URL http://localhost:8081/, and clicking on the “Interactive Server API” link.

This document contains additional details about the server’s various web API methods beyond that provided by the server’s Swagger page. Many of the methods have pre and post-conditions. The pre-conditions specify conditions that should be true before calling the method. If the caller meets the pre-conditions, the method should meet the post-conditions. However, if the caller fails to meet the pre-conditions, the method is not guaranteed to work. You are responsible to make sure you call methods only when the pre-conditions have been met.

For all methods we assume that the request message sent by the client meets the syntax requirements defined by the server’s Swagger page, and we do not specifically list this as a pre-condition (it is just assumed). However, be aware that if a request has invalid syntax, the method call will fail.

Disclaimer: If you find errors in the method documentation, please notify the TAs. The intent is that each method be called only when allowed by the game rules. If you find pre-conditions that are missing on any of the methods, let us know. If you call a method when that operation is not allowed by the game rules (for example, allowing a player to make a move when it is not their turn), bad things may happen. We need to tighten up the error handling in the server so it behaves better when pre conditions are not met, but this is a work in progress.
Non-Move APIs

/user/login

**Description:**
Logs the caller in to the server, and sets their catan.user HTTP cookie. (See the document titled "How the Catan Server Uses HTTP Cookies" for more details on cookies.)

**Pre-conditions:**
username is not null
password is not null

**Post-conditions:**
If the passed-in (username, password) pair is valid,
1. The server returns an HTTP 200 success response with “Success” in the body.
2. The HTTP response headers set the catan.user cookie to contain the identity of the logged-in player. The cookie uses "Path="/", and its value contains a url-encoded JSON object of the following form: { “name”: STRING, “password”: STRING, “playerID”: INTEGER }. For example, { “name”: “Rick”, “password”: “secret”, “playerID”: 14 }.

If the passed-in (username, password) pair is not valid, or the operation fails for any other reason,
1. The server returns an HTTP 400 error response, and the body contains an error message.

**Notes:**
The passed-in username and password may correspond to the credentials of any registered user. The server starts with four users: Sam, Brooke, Pete and Mark. Their passwords are sam, brooke, pete and mark respectively. Any additional registered users should also work with this call.

/user/register

**Description:**
This method does two things:

1) Creates a new user account
2) Logs the caller in to the server as the new user, and sets their catan.user HTTP cookie.
(See the document titled “How the Catan Server Uses HTTP Cookies" for more details on cookies.)
**Pre-conditions:**
username is not null  
password is not null  
The specified username is not already in use.

**Post-conditions:**
If there is no existing user with the specified username,
   1. A new user account has been created with the specified username and password.  
   2. The server returns an HTTP 200 success response with “Success” in the body.  
   3. The HTTP response headers set the catan.user cookie to contain the identity of the logged-in player. The cookie uses ”Path=/”, and its value contains a url-encoded JSON object of the following form: { “name”: STRING, “password”: STRING, “playerID”: INTEGER }. For example, { “name”: “Rick”, “password”: “secret”, “playerID”: 14 }.

If there is already an existing user with the specified name, or the operation fails for any other reason,
   1. The server returns an HTTP 400 error response, and the body contains an error message.

**Notes:**
You should be able to register any username via this call, unless that username is already registered with another user. There is no method for changing passwords.

**/games/list**

**Description:**
Returns information about all of the current games on the server.

**Pre-conditions:** None

**Post-conditions:**
If the operation succeeds,
   1. The server returns an HTTP 200 success response.  
   2. The body contains a JSON array containing a list of objects that contain information about the server’s games

If the operation fails,
   1. The server returns an HTTP 400 error response, and the body contains an error message.
Output JSON format:
The output is a JSON array of game objects. Each game object contains the title and ID of a game, and an array of four player objects containing information about players who have joined the game. Each player object contains the color, name and ID of a player who has joined the game. Players who have not yet joined the game are represented as empty JSON objects. The id's are integers, and colors are one of the following values: red, green, blue, yellow, puce, brown, white, purple, orange.

```json
[
  {
    "title": "Game Name",
    "id": 0,
    "players": [  
      {
        "color": "orange",
        "name": "Player Name",
        "id": 0
      },  
      ...
    ]
  },
  ...
]
```

/games/create

Description:
Creates a new game on the server.

Pre-conditions:
- name != null
- randomTiles, randomNumbers, and randomPorts contain valid boolean values

Post-conditions:
If the operation succeeds,
1. A new game with the specified properties has been created
2. The server returns an HTTP 200 success response.
3. The body contains a JSON object describing the newly created game

If the operation fails,
1. The server returns an HTTP 400 error response, and the body contains an error message.
Output JSON format:
The output is a JSON object containing information about the newly created game, including its title, ID, and an array of four empty player objects.

```json
{
    "title": "Game Name",
    "id": 3,
    "players": [
        {},
        {},
        {},
        {}
    ]
}
```

/games/join

Description:
Adds the player to the specified game and sets their catan.game cookie. (See the document titled “How the Catan Server Uses HTTP Cookies” for more details on cookies.)

Pre-conditions:
1. The user has previously logged in to the server (i.e., they have a valid catan.user HTTP cookie).
   2. The player may join the game because
      2.a They are already in the game, OR
      2.b There is space in the game to add a new player
   3. The specified game ID is valid
   4. The specified color is valid (red, green, blue, yellow, puce, brown, white, purple, orange)

Post-conditions:
If the operation succeeds,
1. The server returns an HTTP 200 success response with “Success” in the body.
   2. The player is in the game with the specified color (i.e. calls to /games/list method will show the player in the game with the chosen color).
      3. The server response includes the “Set-cookie” response header setting the catan.game HTTP cookie

If the operation fails,
1. The server returns an HTTP 400 error response, and the body contains an error message.

/games/save

Description:
This method is for testing and debugging purposes. When a bug is found, you can use the /games/save method to save the state of the game to a file, and attach the file to a bug report. A developer can later restore the state of the game when the bug occurred by loading the previously saved file using the /games/load method. Game files are saved to and loaded from the server's saves/ directory.

Pre-conditions:
1. The specified game ID is valid
2. The specified file name is valid (i.e., not null or empty)

Post-conditions:
If the operation succeeds,
1. The server returns an HTTP 200 success response with “Success” in the body.
2. The current state of the specified game (including its ID) has been saved to the specified file name in the server’s saves/ directory

If the operation fails,
1. The server returns an HTTP 400 error response, and the body contains an error message.

/games/load

Description:
This method is for testing and debugging purposes. When a bug is found, you can use the /games/save method to save the state of the game to a file, and attach the file to a bug report. A developer can later restore the state of the game when the bug occurred by loading the previously saved file using the /games/load method. Game files are saved to and loaded from the server’s saves/ directory.

Pre-conditions:
1. A previously saved game file with the specified name exists in the server’s saves/ directory.

Post-conditions:
If the operation succeeds,
1. The server returns an HTTP 200 success response with “Success” in the body.
2. The game in the specified file has been loaded into the server and its state restored (including its ID).

If the operation fails,

1. The server returns an HTTP 400 error response, and the body contains an error message.

/game/model?version=N

Description:
Returns the current state of the game in JSON format.

In addition to the current game state, the returned JSON also includes a “version” number for the client model. The next time /game/model is called, the version number from the previously retrieved model may optionally be included as a query parameter in the request (/game/model?version=N). The server will only return the full JSON game state if its version number is not equal to N. If it is equal to N, the server returns “true” to indicate that the caller already has the latest game state. This is merely an optimization. If the version number is not included in the request URL, the server will return the full game state.

Pre-conditions:

1. The caller has previously logged in to the server and joined a game (i.e., they have valid catan.user and catan.game HTTP cookies).
2. If specified, the version number is included as the “version” query parameter in the request URL, and its value is a valid integer.

Post-conditions:

If the operation succeeds,

1. The server returns an HTTP 200 success response.
2. The response body contains JSON data
   a. The full client model JSON is returned if the caller does not provide a version number, or the provide version number does not match the version on the server
   b. “true” (true in double quotes) is returned if the caller provided a version number, and the version number matched the version number on the server

If the operation fails,

1. The server returns an HTTP 400 error response, and the body contains an error message.
The format of the returned JSON can be found on the server’s Swagger page, or in the document titled “Client Model JSON Documentation”.

/game/reset

Description:
Clears out the command history of the current game.

For the default games created by the server, this method reversts the game to the state immediately after the initial placement round. For user-created games, this method reversts the game to the very beginning (i.e., before the initial placement round).

This method returns the client model JSON for the game after it has been reset.

You must login and join a game before calling this method.

Pre-conditions:
1. The caller has previously logged in to the server and joined a game (i.e., they have valid catan.user and catan.game HTTP cookies).

Post-conditions:
If the operation succeeds,
1. The game’s command history has been cleared out
2. The game’s players have NOT been cleared out
3. The server returns an HTTP 200 success response.
4. The body contains the game’s updated client model JSON

If the operation fails,
1. The server returns an HTTP 400 error response, and the body contains an error message.

Note:
When a game is reset, the players in the game are maintained.

/game/commands [GET]

Description:
Returns a list of commands that have been executed in the current game.
This method can be used for testing and debugging. The command list returned by this method can be passed to the /game/command (POST) method to re-execute the commands in the game. This would typically be done after calling /game/reset to clear out the game's command history. This is one way to capture the state of a game and restore it later. (See the /games/save and /games/load methods which provide another way to save and restore the state of a game.)

For the default games created by the server, this method returns a list of all commands that have been executed after the initial placement round. For user-created games, this method returns a list of all commands that have been executed since the very beginning of the game (i.e., before the initial placement round).

You must login and join a game before calling this method.

**Pre-conditions:**
1. The caller has previously logged in to the server and joined a game (i.e., they have valid catan.user and catan.game HTTP cookies).

**Post-conditions:**
If the operation succeeds,
1. The server returns an HTTP 200 success response.
2. The body contains a JSON array of commands that have been executed in the game. This command array is suitable for passing back to the /game/command [POST] method to restore the state of the game later (after calling /game/reset to revert the game to its initial state).

If the operation fails,
1. The server returns an HTTP 400 error response, and the body contains an error message.

**/game/commands [POST]**

**Description:**
Executes the specified command list in the current game.

This method can be used for testing and debugging. The command list returned by the /game/command [GET] method is suitable for passing to this method.

This method returns the client model JSON for the game after the command list has been applied.

You must login and join a game before calling this method.
**Pre-conditions:**

1. The caller has previously logged in to the server and joined a game (i.e., they have valid catan.user and catan.game HTTP cookies).

**Post-conditions:**

If the operation succeeds,

1. The passed-in command list has been applied to the game.
2. The server returns an HTTP 200 success response.
3. The body contains the game’s updated client model JSON

If the operation fails,

1. The server returns an HTTP 400 error response, and the body contains an error message.

**/game/listAI**

**Description:**

Returns a list of supported AI player types.

Currently, LARGEST_ARMY is the only supported type.

**Pre-conditions:** **None**

**Post-conditions:**

If the operation succeeds,

1. The server returns an HTTP 200 success response.
2. The body contains a JSON string array enumerating the different types of AI players. These are the values that may be passed to the /game/addAI method.

**/game/addAI**

**Description:**

Adds an AI player to the current game.

You must login and join a game before calling this method.

**Pre-conditions:**

1. The caller has previously logged in to the server and joined a game (i.e., they have valid catan.user and catan.game HTTP cookies).
2. There is space in the game for another player (i.e., the game is not “full”).
3. The specified “AIType” is valid (i.e., one of the values returned by the /game/listAI
   method).

Post-conditions:
If the operation succeeds,
   1. The server returns an HTTP 200 success response with “Success” in the body.
   2. A new AI player of the specified type has been added to the current game. The server
      selected a name and color for the player.

If the operation fails,
   1. The server returns an HTTP 400 error response, and the body contains an error
      message.

/util/changeLogLevel

Description:
Sets the server's logging level.

Pre-conditions:
   1. The caller specifies a valid logging level. Valid values include: SEVERE, WARNING,
      INFO, CONFIG, FINE, FINER, FINEST

Post-conditions:
If the operation succeeds,
   1. The server returns an HTTP 200 success response with “Success” in the body.
   2. The Server is using the specified logging level

If the operation fails,
   1. The server returns an HTTP 400 error response, and the body contains an error
      message.

Move APIs

This section describes all the moves that can be made in a game. When a player makes a move in
a game, the client calls one of these methods to execute the move on the server. Each kind of
move has a “type” string that serves as a name for that kind of move (e.g., “sendChat”,
“finishTurn”, “offerTrade”, etc.). There is a different web method for each different type of
move (e.g., “/moves/sendChat”, “/moves/finishTurn”, “/moves/offerTrade”, etc.). The input to
each /moves/* method is a JSON object contained in the HTTP request body that describes the move’s parameters. We call this JSON object a “command object”, because it describes the move command to be executed on the server.

The following sections describe each of the available move methods. The pre- and post-conditions are given for each method. They are grouped in sections that contain common pre-conditions (e.g., it must be your turn). If pre-conditions on a specific move are omitted, the only preconditions it has are the ones for its section.

**Common Data Types**

Legal values for the various command object properties are given, unless they are obvious from the property’s name (i.e. the `playerIndex` property must be a `playerIndex`, `resource1` must be a `Resource`, etc.) Many of the command object properties are of the following data types.

- **playerIndex** - An integer representing a player’s position in the game’s turn order. It has one of the following values: 0, 1, 2, or 3.

- **EdgeLocation** - Represents the location of an edge on the game map. The x and y properties are integers in the range \([-3, 3]\) that represent a hex location. The direction property is one of the following strings: `[NW, N, NE, SE, S, SW]`.

- **VertexLocation** - Represents the location of a vertex on the game map. The x and y properties are integers in the range \([-3, 3]\) that represent a hex location. The direction property is one of of the following strings: `[NW, NE, E, SE, SW, W]`.

- **Resource** - A resource type is one of the following strings: `[Brick, Wood, Sheep, Wheat, Ore]`

- **ResourceHand** - A JSON object that represents a collection of resource cards. It has five properties: `[brick, wood, sheep, wheat, ore]`. Each has an integer value. 0 means the card amount doesn’t change. A negative value means the player is gaining one or more of that card. A positive number means the player is losing one of more of that card.

**Universal Command Properties**

All move commands objects contain at least the following properties. These properties are described once here and should be assumed for each move type.

1. “type”: `string` [the name of the move being executed (“sendChat”, “finishTurn”, “offerTrade”, etc.]
2. “playerIndex”: `integer` in the range \([0-3]\) [the player’s position in the game’s turn order]
Universal Pre-Conditions
All /move/* methods also have a common pre-condition in that they assume that the caller has already logged in to the server and joined a game. This pre-condition is not repeated on each move type, but should be assumed.

Anytime Commands

➢ sendChat
  ○ Properties:
    ■ content: string [the message you want to send]
  ○ Preconditions
    ■ None (this command may be executed at any time by any player)
  ○ Postconditions
    ■ The chat contains your message at the end

Miscellaneous Commands

➢ acceptTrade
  ○ Properties
    ■ willAccept: boolean [Whether or not you accept the offered trade]
  ○ Preconditions
    ■ You have been offered a domestic trade
    ■ To accept the offered trade, you have the required resources
  ○ Postconditions
    ■ If you accepted, you and the player who offered swap the specified resources
    ■ If you declined no resources are exchanged
    ■ The trade offer is removed

➢ discardCards
  ○ Properties
    ■ discardedCards: ResourceHand [the cards you are discarding]
  ○ Preconditions
    ■ The status of the client model is 'Discarding'
    ■ You have over 7 cards
    ■ You have the cards you're choosing to discard.
  ○ Postconditions
    ■ You gave up the specified resources
    ■ If you're the last one to discard, the client model status changes to 'Robbing'
'Rolling' Commands

➢ rollNumber
  ○ Properties
    ■ *number*: integer in the range 2-12 [the number you rolled]
  ○ Precondition
    ■ It is your turn
    ■ The client model’s status is ‘Rolling’
  ○ Postconditions
    ■ The client model’s status is now in ‘Discarding’ or ‘Robbing’ or ‘Playing’

'Playing' Commands

General preconditions:
  ● It is your turn
  ● The client model’s status is 'Playing'

➢ buildRoad
  ○ Properties
    ■ *free*: boolean [whether or not you get this piece for free (i.e., in setup)]
    ■ *roadLocation*: EdgeLocation [the new road’s location]
  ○ Preconditions:
    ■ The road location is open
    ■ The road location is connected to another road owned by the player
    ■ The road location is not on water
    ■ You have the required resources (1 wood, 1 brick; 1 road)
    ■ Setup round: Must be placed by settlement owned by the player with no adjacent road
  ○ Postconditions:
    ■ You lost the resources required to build a road (1 wood, 1 brick; 1 road)
    ■ The road is on the map at the specified location
    ■ If applicable, “longest road” has been awarded to the player with the longest road

➢ buildSettlement
  ○ Properties
    ■ *free*: boolean [whether or not you get this piece for free (i.e. in setup)]
    ■ *vertexLocation*: VertexLocation [the location of the settlement]
  ○ Preconditions:
    ■ The settlement location is open
- The settlement location is not on water
- The settlement location is connected to one of your roads except during setup
- You have the required resources (1 wood, 1 brick, 1 wheat, 1 sheep; 1 settlement)
- The settlement cannot be placed adjacent to another settlement
- **Postconditions:**
  - You lost the resources required to build a settlement (1 wood, 1 brick, 1 wheat, 1 sheep; 1 settlement)
  - The settlement is on the map at the specified location

**buildCity**

- **Properties**
  - `vertexLocation`: `VertexLocation` [the location of the city]
- **Preconditions**
  - The city location is where you currently have a settlement
  - You have the required resources (2 wheat, 3 ore; 1 city)
- **Postconditions**
  - You lost the resources required to build a city (2 wheat, 3 ore; 1 city)
  - The city is on the map at the specified location
  - You got a settlement back

**offerTrade**

- **Properties**
  - `offer`: `ResourceHand` [negative numbers mean you get those cards]
  - `receiver`: `playerIndex` [the recipient of the trade offer]
- **Postconditions**
  - The trade is offered to the other player (stored in the server model)

**maritimeTrade**

- **Properties**
  - `ratio`: `integer` [2, 3, or 4]
  - `inputResource`: `Resource` [what you are giving]
  - `outputResource`: `Resource` [what you are getting]
- **Preconditions**
  - You have the resources you are giving
  - For ratios less than 4, you have the correct port for the trade
- **Postconditions**
The trade has been executed (the offered resources are in the bank, and the requested resource has been received)

robPlayer

- Properties
  - location: HexLocation [the new robber location]
  - victimIndex: playerIndex, or -1 if you are not robbing anyone [the player you are robbing]

- Preconditions
  - The robber is not being kept in the same location
  - If a player is being robbed (i.e., victimIndex != -1), the player being robbed has resource cards

- Postconditions
  - The robber is in the new location
  - The player being robbed (if any) gave you one of his resource cards (randomly selected)

finishTurn

- Properties
  - None (except the universal properties)

- Precondition
  - None (except the preconditions for this section)

- Postcondition
  - The cards in your new dev card hand have been transferred to your old dev card hand
  - It is the next player’s turn

buyDevCard

- Properties
  - None (except the universal properties)

- Preconditions
  - You have the required resources (1 ore, 1 wheat, 1 sheep)
  - There are dev cards left in the deck

- Postconditions
  - You have a new card
    - If it is a monument card, it has been added to your old devcard hand
    - If it is a non-monument card, it has been added to your new devcard hand (unplayable this turn)
**Dev Card Commands**

General Preconditions:
- It is your turn
- The client model status is 'Playing'
- You have the specific card you want to play in your old dev card hand
- You have not yet played a non-monument dev card this turn

➢ Soldier
   - Properties
     - location: HexLocation [the new robber location]
     - victimIndex: playerIndex, or -1 if you are not robbing anyone [the player you are robbing]
   - Preconditions
     - The robber is not being kept in the same location
     - If a player is being robbed (i.e., victimIndex != -1), the player being robbed has resource cards
   - Postconditions
     - The robber is in the new location
     - The player being robbed (if any) gave you one of his resource cards (randomly selected)
     - If applicable, “largest army” has been awarded to the player who has played the most soldier cards
     - You are not allowed to play other development cards during this turn (except for monument cards, which may still be played)

➢ Year_of_Plenty
   - Properties
     - resource1: Resource [the first resource you want to receive]
     - resource2: Resource [the second resource you want to receive]
   - Preconditions
     - The two specified resources are in the bank
   - Postconditions
     - You gained the two specified resources

➢ Road_Building
   - Properties
     - spot1: EdgeLocation
     - spot2: EdgeLocation
- **Preconditions**
  - The first road location (spot1) is connected to one of your roads.
  - The second road location (spot2) is connected to one of your roads or to the first road location (spot1)
  - Neither road location is on water
  - You have at least two unused roads
- **Postconditions**
  - You have two fewer unused roads
  - Two new roads appear on the map at the specified locations
  - If applicable, “longest road” has been awarded to the player with the longest road

- **Monopoly**
  - **Properties**
    - resource: **Resource** [the resource being taken from the other players]
  - **Preconditions**
    - None (except the general Preconditions for this section)
  - **Postconditions**
    - All of the other players have given you all of their resource cards of the specified type

- **Monument**
  - **Properties**
    - None (except the universal properties)
  - **Preconditions**
    - You have enough monument cards to win the game (i.e., reach 10 victory points)
  - **Postconditions**
    - You gained a victory point