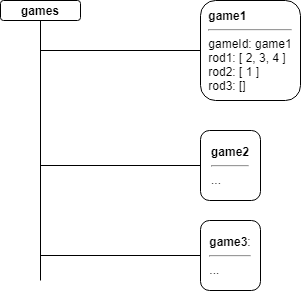
Tower of Hanoi Server Design

# Basics

* Used Google Firebase cloud functions with Typescript to build the game engine that exposes a RESTful API
* Used Google Firestore (a NoSQL database) to store any game data
* Used Postman for API testing while implementation
* Created a Nano ID (similar to UUID v4) upon game session creation
  + The created UUID is used in the following cases
    - when reading/updating/deleting the existing game state
* Used Ajv to validate JSON input from client side
* Used Jest for unit testing

# Data Model

On Firestore, data is stored under the games collection. Each document represents a game session. The following picture is an example of how the structure looks like:

* games: a collection
* game1: a document with ID “game1”, which contains the following fields
  + gameId: string of a Nano ID, indicating the game ID
  + rod1: number[], indicating the disks it currently holds. An empty means the rod has no disks stack onto it
  + rod2: same as rod1
  + Rod3: same as rod1

Here is how its JSON representation looks like:

{

“games”: {

“gameId”: //some string

“rod1”: [ // an array of numbers, used as a stack ]

“rod2”: [ // an array of numbers, used as a stack ]

“rod3”: [ // an array of numbers, used as a stack ]

}

# REST APIs

All RESTful APIs can be requested by appending to the route **<https://us-central1-tower-of-hanoi-314419.cloudfunctions.net/webApi>**

**POST (api/create)**

* Instead of a traditional POST request that grabs client side input and use the given information upon game creation, the game engine exposes a POST API that takes no input from the client side. The server provides a default game state to the client
* In the POST request, the server generates a Nano ID as a unique identifier for the created game. The ID is stored as well as sent back to the client with the default game state
* The server sends back an *OK (200)* response upon a successful creation. Victory condition check result and the created game state are sent together as response message
* Any unexpected error is sent back to the client as a *Internal Server Error* *(500)*

**GET (api/read/:gameId)**

* The GET request is the same as a traditional one. It requires a gameId as the request parameter which is used for retrieving data from the database
* Before the actual data retrieval, the server checks whether the specified game exists in database. If not, the server sends back a *Not Found (404)* response without performing the operation
* The server sends back an *OK (200)* response upon a successful read. Victory condition check result and the retrieved game state are sent together as response message
* Any unexpected error is sent back to the client as a *Internal Server Error (500)*

**PUT (api/update/:gameId)**

* Instead of a traditional PUT request that takes a complete JSON object matching the data model stored in the database as request body for updating an existing game, the game engine requires the request body in the following syntax
  + { “from”: // a string representing a valid rod, “to”: // a string representing a valid rod }
* The server proceed the operation by removing the top-most disk (remove the number in the beginning of the corresponding number[]) from the “from” rod and add it to the top of “to”rod (insert the number to the beginning of the corresponding number[])
* This proposed PUT request implementation has the following benefits over a traditional one
  + Less syntax validation effort as only two fields are required from clients
  + Enforce the rule that only one disk can be moved at a time
* The server performs the following checks before the actual data update
  + Check data existence. If data does not exist in database, the server sends back a *Not Found (404)* response without performing the operation
  + Validate client’s request body. If the input is invalid (e.g., missing field, invalid rod is provided) the server sends back a *Bad Request (400)* response without performing the operation
  + Validate client’s attempt move. If client attempts an invalid move (i.e., tying to move a disk from an empty rod, or move a larger disk onto a smaller one) the server sends back a *Forbidden (403)* response without performing the operation
* The server sends back an *OK (200)* response upon a successful update. Victory condition check result and the updated game state are sent together as response message
* Any unexpected error is sent back to the client as a *Internal Server Error (500)*

**DELETE (api/delete/:gameId)**

* The DELETE request is the same as a traditional one. It requires a gameId as the request parameter which is used for deleting data from the database
* Before the actual data deletion, the server checks whether the specified game exists in database. If not, the server sends back a *Not Found (404)* response without performing the operation
* The server sends back an *OK (200)* response upon a successful deletion. The deleted gameId is included in the response message
* Any unexpected error is sent back to the client as a *Internal Server Error (500)*

# Alternative Solution

* Store game data in a local database
  + Pros
    - The server has total control over the data
    - Data is easily accessible (no network limitation, e.g., internet outage, network latency)
  + Cons
    - Game state will be cleared when server is down
    - Not easily scale-able
    - Requires more effort on data maintenance (e.g., manual data backup)
* Use the traditional POST request format
  + Pros
    - Provides better flexibility as clients can decide the default game state
  + Cons
    - More error-prone as clients have more control over data
    - More validation effort on the server side
* Use the traditional PUT request format
  + Pros
    - Client is more familiar with the traditional style
  + Cons
    - More error-prone as clients have more control over data
    - More validation effort on the server side
    - More computation required on the server side to enforce the game rules

# Improvements and Optimization

* Complete unit tests and integration tests
* Make data update asynchronous
* Support more than 4 disks. This can be achieved by
  + Add an additional field “numberOfDisks” in the data model
  + Request a client input of a number greater than or equal to 4 upon game creation
  + During game creation, dynamically add disks to initial rod
* Use localization for resources (e.g., string) used in multiple places
* Support multiplayer gaming
* Require user authentication. When storing data to database, add one extra layer of encryption for data protection