Katana Backend Programming Test

Scenario

You intend to create card games like Poker and Blackjack. The goal is to create an API to handle decks and cards to be used in any game like these.

Instructions

You are required to provide an implementation of REST API to simulate a deck of cards.

You need to provide a solution written in NodeJS Typescript. If you do not feel too comfortable with the language, it is OK to research a little before starting writing the API. Feel free to use any NodeJS Typescript framework in the market or simply use a custom build.

The API should have the following methods to handle cards and decks:

- Create a new **Deck**
- Open a **Deck**
- Draw a **Card**

Requirements

Create a new Deck

The solution provides an endpoint that would create the standard 52-card deck of French playing cards, it includes all thirteen ranks in each of the four suits: spades (), clubs (), diamonds (), and hearts (). You don’t need to worry about Joker cards for this assignment.

There should be a way that allows the created deck to be shuffled.

When user has specified that they want the deck created deck to be short deck (SHORT), it should only include 32 cards (lacking cards from 2 to 6).

Request

The request should allow following specifications for the deck:

```
{
   "type": "FULL",
   "shuffled": true
}
```

Response

The response from the endpoint should include the following:

- The deck id (**UUID**)
- The deck type: FULL/SHORT (**string**)
- The deck properties like shuffled (**boolean**)
- Total cards remaining in this deck (**integer**)

```
{
   "deckId": "521b0293-01f7-44c2-9990-27079eb2352d",
   "type": "FULL",
   "shuffled": true,
   "remaining": 52
}
```

Open a Deck

It would return a given deck that matches the specified UUID. This method will “open the deck”, meaning that it will list all cards by the order they were added to the deck (which might or might not be shuffled based on configuration used to create the deck). If the deck UUID is not provided or is not a valid UUID, the endpoint should return a corresponding error.

The list should be shuffled when created deck included {shuffled: true}. Note that for a given deck, the shuffling should remain consistent (should not differ every time the deck/open is called).

The response needs to return a JSON that would include:

- The deck id (**UUID**)
- The deck type: FULL/SHORT (**string**)
- The deck shuffled state (**boolean**)
- Total cards remaining in the deck (**integer**)
- All the remaining cards (**card object**)

```
{
    "deckId": "521b0293-01f7-44c2-9990-27079eb2352d",
    "type": "FULL",
    "shuffled": true,
    "remaining": 3,
    "cards": [
      {
        "value": "ACE",
        "suit": "SPADES",
        "code": "AS"
      },
      {
        "value": "KING",
        "suit": "HEARTS",
        "code": "KH"
      },
      {
        "value": "10",
        "suit": "SPADES",
        "code": "10S"
      },
      {
        "value": "8",
        "suit": "CLUBS",
        "code": "8C"
      }
   ]
}
```

Draw a Card

When you draw a card(s) of a given Deck, if the deck was not passed over or invalid it should return an error. A count parameter needs to be provided to define how many cards to draw from the deck. The cards should be drawn from the top of the deck (for both non-shuffled and shuffled decks).

The response needs to return a JSON that would include:

- All the drawn cards (**card object**)

```
{
    "cards": [
        {
           "value": "ACE",
           "suit": "SPADES",
           "code": "AS"
        },
        {
           "value": "KING",
           "suit": "HEARTS",
           "code": "KH"
        },
    ]
}
```

What are we looking for

We are interested in

- **How you structure your code.** Code should be:
  - Well tested
  - Easy to extend (think of other card games you want to create)
  - Easy to modify
  - Easy to understand by others
  - Complies with coding best practices
- Evidence of your Back-End development knowledge
- Evidence of testing (TDD, BDD)

We are not expecting a polished solution, so please do not spend time working outside of the required scope of this exercise.

Please upload your solution to GitHub with a clear step-by-step readme file explaining how to build and run your submission, build scripts, and any tests you have written.

The solution you create will serve as a context for later technical conversation that you'll be having with engineers from Katana (responsibility segregation, REST, SOLID and scalability).

Bonus points

These points are optional but would be great if you:

- Use docker
- Use any other database than "in-memory"
- Provide unit and integration tests (at least few from both)

We expect this test to take approximately 5 hours.