Aufgabe 2

In dieser Aufgabe wird es darum gehen, folgende Komponenten zu bauen:

- die Game-Engine, also die fachliche Implementierung der Spielregeln
- einen RESTful Controller, der Funktionen der Engine über ein RESTful-Interface verefügbar macht
- Persistenz
  - Eine Entity-Klasse, um Spiele via JPA zu speichern
  - Eine Repository-Klasse, die CRUD-Operationen ermöglicht

Kotlin Doku

https://kotlinlang.org/docs/reference/basic-syntax.html

GameEntity

In der Datei `GameEntity.kt` soll eine Klasse `GameEntity` angelegt werden.

Die Klasse soll eine JPA-Entity sein und über folgende Properties verfügen:

- `id` vom Typ `String`
- `initialItems` vom Typ `Int`
- `remainingItems` vom Typ `Int`
- `finished` vom Typ `Boolean`
- `nextPlayer` vom Typ `Players`
- `winner` vom Typ `Players`-Enum (nullable)

Die Klasse soll einen Konstruktor bekommen, in dem ein Wert für die Property `initialItems` gesetzt werden
kann.

GameRepository

In der Datei `GameRepository.kt` soll ein Interface `GameRepository` angelegt werden.

Dieses Interface soll ein JPA-Repository für die Entity `GameEntity` sein.

GameEngine

In der Datei `GameEngine.kt` soll eine Klasse `GameEngine` als Spring-Service angelegt werden.

Tipp: Logging mit KLogging

In diesem Projekt ist das Logging mit dem Framework KLogging verfügbar.

Durch diese Anweisung:

```
class GameEngine {

    /**
     * define KLogging() as companion object
     */
    companion object : KLogging()
```

Wird der Logger folgendermaßen verfügbar gemacht:

```
    // ...
    logger.debug {"This is a log message"}
    // ...
```

Datenklasse ValidationResult

In der Datei `GameEngine.kt` soll weiterhin eine Datenklasse `ValidationResult` angelegt werden.

`ValidationResult` soll eine Funktion `isValid()` anbieten, deren Rückgabewert den Typ Boolean hat.

Die Klasse soll ein Set (`MutableSet` von `String`) enthalten. In diesem Set sollen Fehlermeldungen als Strings
gespeichert werden, die nach einer Validierung eines Spielzugs anfallen.

`isValid()` soll den Wert `true` zurückgeben, wenn das o.g. Set leer ist.

Validierung von Spielzügen

Ein Spielzug ist nicht valide wenn:

- er auf einem Spiel ausgeführt werden soll, `game.finished` den Wert `true` hat
- ein Spieler einen Spielzug macht, der gemäß der Property `game.nextPlayer` nicht an der Reihe ist
- durch einen Spielzug mehr Hölzer genommen werden sollen, als in `gameEntity.remainingItems` verfügbar sind.

Für jede fehlgeschlagene Validierung soll eine Fehlermeldung in das Set in `ValidationResult`

Funktion play()

Diese Klasse `GameEngine` soll genau eine Funktion `play()` implementieren. Deren Parameter sollen sein:

- `game` vom Typ `GameEntity`
  - dieser Parameter wird den Zustand eines Spiels vor dem aktuellen Spielzug enthalten.
- `numberOfItems` vom Typ `Int`
  - dieser Parameter wird angeben, wieviele Hölzer im aktuellen Spielzug antnommen werden
- `player` vom Typ `Players`
  - dieser Parameter wird angeben, welcher Spieler gerade spielt.

Der Rückgabewert von `pair()` ist `Pair<GameEntity, ValidationResult>`

Constraints

- `play()` muss dafür sorgen, dass
  - die Property `gameEntity.remainingItems` um den Wert `numberOfItems` subtrahiert wird.
  - die Property `game.nextPlayer` beim Verlassen der Funktion den Wert des nächsten Spielers hat. Dieser Wert muss
    ungleich dem Wert `game.nextPlayer` zum Aufrufzeitpunkt der Methode sein.
  - die Property `game.finished` auf `true` gesetzt wird, sofern die Substraktion von `gameEntity.remainingItems` um den Wert `numberOfItems` zum Ergebnis `1` oder `0` geführt hat. In diesem Fall muss auch
    die Property `gameEntity.winner` korrekt gesetzt werden.

GameRestController

In der Datei `GameRestController.kt` soll eine Klasse angelegt werden, die vom Interface `org.example.demo.rest.api.GameApi` erbt und alle davon abgeleiteten Methoden implementieren

Die Klasse soll ein Spring Rest-Controller sein.

Über Dependency-Injection sollen die Klassen `GameRepository` und `GameEngine` injziert werden.

Es müssen die drei Funktionen des Interfaces implementiert werden.

getGames()

Diese Funktion soll alle vorhandenen Entitäten vom Typ `GameEntity` aus der Datenbank abfragen. Danach sollen
sie auf den Typ `GameDto` konvertiert werden und als `ResponseEntity<List<GameDto>>` zurückgegeben werden.

Tipp: Konvertieren mit .apply()

```
gameRepository // gibt GameEntity zurück in der Variable "it"
               .findAll()
               .map {
                   GameDto().apply { // Zugriff auf das neuw GameDto mit "this"
                       this.id = it.id
                       this.initialItems = it.initialItems
                       // ..
                   }
               }
```

createGame()

Diese Funktion soll ein neues Spiel anlegen. Aus dem Funktionsparameter vom Typ `NewGameDto` soll eine
Instanz von `GameEntity` erzeugt und persistiert werden.

Als Rückgabewert die Game ID zurückgeliefert werden. Außerdem soll im `Location` Header die relative URL auf das Spiel gesetzt werden. Beispiel:

```
        // Create uri to game and return it
        val headers = HttpHeaders().apply {
            location = URI("/game/${savedGame.id}")
        }

return ResponseEntity(savedGame.id, headers, HttpStatus.CREATED)
```

play()

Diese Funktion führt einen Spielzug durch. Als Parameter werden die Spiel-ID und die Daten des durchzuführenden Spielzugs übergeben.

Zunächst muss anhand der Spiel-ID das existierende Spiel als `GameEntity` aus der Datenbank geladen werden.

Danach wird die `GameEntity` zusammen mit der Anzahl der genommenen Hölzer und dem Spielertyp an die
Funktion `GameEngine.play()` übergeben.

Bei einem validen Spielzug soll der neue Spielzustand persistiert und als `GameDto` zurück gegeben werden.

Bei einem nicht validen Spielzug soll sich der Spielzustand nicht ändern und stattdessen eine HTTP-Response
mit Fehlerstatuscode 400 (Bad Request) zurückgegeben werden.

Bei einem Spielzug für ein nicht existierendes Spiel (Spiel-ID in DB unbekannt) soll eine HTTP-Response
mit Fehlerstatuscode 404 (Not Found) zurückgegeben werden.

Tipp: Werfen von Exceptions, um in REST-Controllern HTTP-Fehlercodes zurückzugeben

In der Datei `GameRestController.kt` kann folgende Exception definiert werden.

```
@ResponseStatus(value = HttpStatus.NOT_FOUND, reason = "Game does not exist.")
class GameNotFoundException : RuntimeException()
```

Wird die `GameNotFoundException` in einer Funktion geworfen, die auf einen HTTP-Endpunkt gemappt ist,
so gibt dieser Endpunkt als Response den hier definierten HTTP-Fehlercode zurück.