Football Data Analysis System Report

Index Table

Introduction

[no marks] – This should give the marker a guide as to what to expect from your report. Please make the marker's work easier by being honest. [This part needs more information to be completed]

Task 1: Data Storage Design

Solution:

Design and its motivations:

I decided to create six tables in the PostgreSQL database to store the more static data that doesn't change frequently. The tables are designed to store information about clubs, competitions, nations, player statistics, players, and users of the web statistics soccer page. This structure allows for efficient storage and retrieval of data that is less dynamic. The use of a relational database like PostgreSQL is suitable for this purpose due to its robustness, support for complex queries, and data integrity features.

PostgreSQL Tables:

- **Clubs:**

```
CREATE TABLE public.clubs (
    club_id integer NOT NULL,
    club_code character varying(255),
    name character varying(255),
    domestic_competition_id character varying(50),
    total_market_value character varying(50),
    squad_size integer,
    average_age numeric(3,1),
    foreigners_number integer,
    foreigners_percentage numeric(4,1),
    national_team_players integer,
    stadium_name character varying(255),
    stadium_seats integer,
    net_transfer_record character varying(50),
    coach_name character varying(255),
    last_season integer,
    url character varying(1024)
);
```

- **Competitions:**

```
CREATE TABLE public.competitions (
  competition_id character varying(50) NOT NULL,
  competition_code character varying(255),
  name character varying(255),
  sub_type character varying(100),
  type character varying(100),
  country_id character varying(100),
  country_name character varying(100),
  domestic_league_code character varying(50),
  confederation character varying(50),
  url character varying(1024)
);
```

- **Nations:**

```
CREATE TABLE public.nations (
  name character varying(255) NOT NULL,
  sig text
);
```

- **Players Stat:**

```
CREATE TABLE public.players_stat (
  player_id integer,
  name text,
  yellow_cards integer,
  red_cards integer,
  goals integer,
  assists integer,
  minutes_played integer,
  current_club_id integer,
  country_of_birth text,
  city_of_birth text,
  country_of_citizenship text,
  date_of_birth date,
  foot text,
  height_in_cm integer,
  highest_market_value_in_eur integer,
  image_url text,
  url text,
  partite integer,
  age integer
);
```

- **Players:**

```
CREATE TABLE public.players (
  player_id integer NOT NULL,
  first_name character varying(255),
  last_name character varying(255),
  name character varying(255),
  last_season integer,
  current_club_id integer,
  player_code character varying(255),
  country_of_birth character varying(255),
  city_of_birth character varying(255),
  country_of_citizenship character varying(255),
  date_of_birth date,
  sub_position character varying(255),
  position character varying(255),
  foot character varying(255),
  height_in_cm integer,
  market_value_in_eur integer,
  highest_market_value_in_eur integer,
  contract_expiration_date date,
  agent_name character varying(255),
  image_url character varying(1024),
  url character varying(1024),
  current_club_domestic_competition_id character varying(255),
  current_club_name character varying(255)
);
```

- **Users:**

```
CREATE TABLE public.users (
  email character varying(255) NOT NULL,
  password character varying(60) NOT NULL
);
```

The more dynamic data, which changes frequently, is stored in a MongoDB database called SoccerDB. This NoSQL database is used to store collections of documents for appearances, careers, club games, game events, game lineups, games, and rankings. The use of MongoDB is advantageous for handling large volumes of unstructured data and for its flexibility in dealing with dynamic schemas.

- **Appearances:**

```
{
"game_id": Int32,
"player_id": Int32,
"player_club_id": Int32,
"player_current_club_id": Int32,
"date": Date,
"player_name": String,
"competition_id": String,
"yellow_cards": Int32,
"red_cards": Int32,
"goals": Int32,
"assists": Int32,
"minutes_played": Int32
}
```

- **Career:**

```
{
"player_id": Int32,
"club_id": Int32,
"game_id": Int32,
"player_name": String,
"competition_id": String,
"yellow_cards": Int32,
"red_cards": Int32,
"goals": Int32,
"assists": Int32,
"minutes_played": Int32,
"season": Int32,
"club_name": String,
"competition_name": String
}
```

- **Club Games:**

```
{
"game_id": Int32,
"club_id": Int32,
"own_goals": Int32,
"own_manager_name": Int32,
"opponent_id": Int32,
"opponent_goals": Int32,
"opponent_manager_name": String,
"hosting": String,
"is_win": Int32
}
```

- **Game Events:**

```
{
"date": Date,
"game_id": Int32,
"minute": Int32,
"type": String,
"club_id": Int32,
"player_id": Int32,
"description": String,
"player_assist_id": String,
"player_in_id": String,
"player_name": String,
"assist_name": String
}
```

- **Game Lineups:**

```
{
"game_id": Int32,
"club_id": Int32,
"type": String,
"number": Int32,
"player_id": Int32,
"player_name": String,
"team_captain": Int32,
"position": String
}
```

- **Games:**

```
{
"game_id": Int32,
"competition_id": String,
"season": Int32,
"round": String,
"date": Date,
"home_club_id": Int32,
"away_club_id": Int32,
"home_club_goals": Int32,
"away_club_goals": Int32,
"home_club_position": Int32,
"away_club_position": Int32,
"home_club_manager_name": String,
"away_club_manager_name": String,
"stadium": String,
"attendance": Int32,
"referee": String,
"url": String,
"home_club_name": String,
"away_club_name": String,
"aggregate": String,
"competition_type": String,
"competition_name": String
}
```

- **Rankings:**

```
{
"competition_id": String,
"season": Int32,
"club_id": Int32,
"club_name": String,
"club_goals_home": Int32,
"win_home": Int32,
"loss_home": Int32,
"points_home": Int32,
"home_games": Int32,
"club_goals_away": Int32,
"win_away": Int32,
"loss_away": Int32,
"points_away": Int32,
"away_games": Int32,
"tie": Int32,
"total_goals": Int32,
"total_win": Int32,
"total_loss": Int32,
"total_points": Int32,
"total_games": Int32
}
```

Issues:

Introduce the task:

This section refers to the challenge of storing dynamic and static football data efficiently. The primary challenge is to identify which data is more dynamic and which is less so, and to decide the appropriate storage method for each type. Dynamic data, which changes frequently, needs to be stored in a way that allows for fast updates and retrieval, while static data can be stored in a more traditional relational database.

Requirements:

The design of the data storage system complies with the requirements specified in the original assignment sheet by ensuring efficient storage and retrieval of both dynamic and static data. The use of PostgreSQL for static data such as clubs, competitions, players, nations, player statistics, and user information ensures data integrity and supports complex queries. MongoDB is used for dynamic data like appearances, careers, club games, game events, game lineups, games, and rankings, providing flexibility and efficiency in handling frequently changing data.

- **Static Data (PostgreSQL):**
  - **Tables:** clubs, competitions, players, nations, player statistics, users
  - **Motivation:** PostgreSQL is robust and supports complex queries, making it suitable for static data that requires strong consistency and integrity.
- **Dynamic Data (MongoDB):**
  - **Collections:** appearances, careers, club games, game events, game lineups, games, rankings
  - **Motivation:** MongoDB's flexibility and efficiency in handling large volumes of unstructured data make it ideal for dynamic data that changes frequently.

This division ensures that the system meets all specified requirements, providing a fast and reliable solution for storing and accessing both static and dynamic football data.

Limitations:

Several considerations were taken into account to address potential limitations and ensure the extensibility of the solution:

1. **Identifying Dynamic and Static Data:** The first problem was to determine which data is more dynamic and which is less. For example, user data, which consists of usernames and passwords for JFT Soccer users, is relatively static since it is created once and stored for the necessary duration. Similar considerations were made for other data types.
2. **Data Organization:** The original data from Transfermarkt includes clubs, competitions, players (in PostgreSQL), and appearances, careers, club games, game events, game lineups, games (in MongoDB). Additional data such as nations (with flags), player statistics, and user information for JFT Soccer were also included. This organization was based on the need for these datasets in the main Express server.
3. **Adding Tables and Collections:** For MongoDB, a rankings collection was created to store statistical data for each club, including metrics like wins, losses, goals, etc. The decision to create additional tables and collections rather than relying on more complex queries was made to simplify data retrieval, given that the data is not updated in real-time.
4. **Data Cleaning:** Using pandas and Python, the data was cleaned to address issues like incomplete, damaged, or empty rows. Only the most complete data rows were retained, ensuring data quality and integrity.

Extensibility and Adaptability:

- The solution is designed to be extensible and easily adaptable to other requirements. For example, new tables or collections can be added as needed without disrupting the existing structure.
- The use of PostgreSQL and MongoDB allows for flexibility in handling both structured and unstructured data, making it easier to scale and adapt to future needs.

Potential Limitations:

- **Data Updates:** The static nature of some data might require manual updates, which could be a limitation if more real-time data is needed in the future.
- **Complex Queries:** While the current design simplifies queries, more complex queries might be necessary as the system grows, potentially impacting performance.

Overall, the design aims to balance efficiency, scalability, and ease of use, acknowledging that no design is flawless but striving to address key challenges and limitations proactively.

Task 2: Main Express Server Design

Solution:

Design and its motivations:

The main Express server is designed to handle the front-end operations of the JFT Soccer website. It serves static assets such as HTML, CSS, and JavaScript files to the client and communicates with other backend servers to fetch and send data. The design of this server focuses on simplicity, speed, and efficiency, ensuring a smooth user experience.

Key Features:

- **Static Asset Serving:** The server delivers static files like HTML, CSS, and JavaScript to the client.
- **API Proxy:** Routes are set up to fetch data from the secondary Express server (connected to MongoDB) and the Spring Boot server (connected to PostgreSQL).
- **Client-Side Libraries:** Utilizes Bootstrap 5 for responsive design, jQuery 3.7.1 for DOM manipulation, and Axios for making HTTP requests.

Advantages:

- **Separation of Concerns:** By isolating the front-end server, it allows independent scaling and maintenance of the user interface without affecting the backend logic.
- **Performance:** Lightweight server setup ensures quick response times and minimal latency in serving static assets.
- **Modularity:** The client-side code is divided into multiple files, each handling specific tasks, enhancing maintainability and readability.

Disadvantages:

- **Complexity in Management:** Requires careful coordination between multiple servers, which can complicate deployment and debugging processes.
- **Potential Bottlenecks:** As the number of user interactions increases, the server might face challenges in managing concurrent requests efficiently.

Issues:

Introduce the task:

This section refers to the challenge of designing a central server (Express) that is fast and can serve thousands of users. The primary challenge is to deliver static assets quickly and handle user interactions efficiently while acting as an intermediary between the client and backend servers.

Requirements:

The design of the main Express server meets the requirements by ensuring efficient delivery of static assets and facilitating communication with backend services. The server setup includes:

- **Route Handling:**
  - Example: Fetching club names from the backend.

```
router.get('/api/clubs-names', async (req, res, next) => {
  try {
    const response = await fetch('http://localhost:8082/clubs-names');
    const data = await response.json();
    res.json(data);
  } catch (error) {
    next(error);
  }
});
```

  - Example: Posting competition data to fetch related games.

```
router.post('/api/games_by_champion', async (req, res) => {
  const { competition_id } = req.body;

  try {
    const response = await fetch('http://localhost:3002/api/games_by_competition', {
      method: 'POST',
      headers: { 'Content-Type': 'application/json' },
      body: JSON.stringify({ competition_id })
    });

    if (!response.ok) {
      const errorMessage = `Failed to fetch games: ${response.statusText}`;
      console.error(errorMessage);
      return res.status(response.status).json({ message: errorMessage });
    }

    const data = await response.json();
    res.status(200).json(data);
  } catch (error) {
    console.error('Error forwarding request:', error);
    res.status(500).json({ message: 'Internal Server Error', details: error.message });
  }
});
```

- **Client-Side Integration:**
  - Using Axios for HTTP requests.

```
function postAxiosQuery(url, data) {
    return axios.post(url, data)
        .then(response => {
            return response.data;
        })
        .catch(error => {
            if (error.response) {
                console.error("Error status:", error.response.status);
                console.error("Error data:", error.response.data);
                console.error("Error headers:", error.response.headers);
            } else if (error.request) {
                console.error("Error request:", error.request);
            } else {
                console.error("Error message:", error.message);
            }
            console.error("Error config:", error.config);
        });
}

function getAxiosQuery(url) {
    return axios.get(url)
        .then(response => {
            return response.data;
        })
        .catch(error => {
            if (error.response) {
                console.error("Error status:", error.response.status);
                console.error("Error data:", error.response.data);
            } else if (error.request) {
                console.error("Error request:", error.request);
            } else {
                console.error("Error message:", error.message);
            }
            console.error("Error config:", error.config);
        });
}
```

- **Client-Side Libraries:**
    - **Bootstrap 5:** Utilized for responsive design and ease of styling.
    - **jQuery 3.7.1:** Used for DOM manipulation and handling events.
    - **Axios:** Simplifies HTTP requests to the backend servers.

Limitations:

Potential Limitations:

- **Scalability:** As the user base grows, the server might face challenges in managing a high volume of concurrent requests, necessitating optimization or scaling solutions.
- **Data Consistency:** Ensuring data consistency between multiple backend services can be complex and might require additional synchronization mechanisms.
- **Error Handling:** Robust error handling is required to manage potential failures in communication with backend services.

Extensibility:

- The modular structure of the client-side code and the use of well-defined API routes ensure that the server can be easily extended to accommodate new features and requirements.
- Integration of new client-side libraries or frameworks can be done with minimal disruption to existing functionality.

Overall, the main Express server is designed to provide a robust and scalable solution for serving the front end of the JFT Soccer website, ensuring efficient interaction between the client and backend services.

Task 3: Secondary Express Server Design

Solution:

Design and its motivations:

The secondary Express server is designed to handle the dynamic data stored in MongoDB. This server uses the Model-View-Controller (MVC) architecture to ensure a clear separation of concerns, making the application more modular and maintainable. The server is responsible for managing data related to game lineups, events, careers, games, and rankings, providing an API for the main Express server to interact with.

Key Features:

- **MVC Architecture:** This design pattern divides the application into three interconnected components: Models, Views, and Controllers. It helps organize the code and makes it easier to manage and scale.
- **MongoDB Integration:** The server is connected to MongoDB, which is suitable for handling large volumes of dynamic and unstructured data.
- **API Endpoints:** Provides API routes for CRUD operations on the MongoDB collections.

Advantages:

- **Modularity:** The MVC architecture allows for easy maintenance and scalability by separating the business logic, data access, and presentation layers.
- **Efficiency:** MongoDB's flexibility and schema-less nature make it ideal for storing and retrieving dynamic data efficiently.
- **Clarity:** Clear division of responsibilities within the codebase enhances readability and simplifies debugging.

Disadvantages:

- **Complexity in Query Writing:** Learning to write efficient queries in MongoDB can be challenging initially.
- **Overhead:** Managing multiple servers and ensuring seamless communication between them can introduce additional overhead.

Issues:

Introduce the task:

This section refers to the challenge of designing a secondary server (Express) that is fast and can handle dynamic data stored in MongoDB. The server must efficiently manage and serve thousands of requests related to game data, player careers, and other dynamic content.

Requirements:

The design of the secondary Express server meets the requirements by ensuring efficient management of dynamic data. The server setup includes:

- **Database Connection:**

```
const mongoose = require('mongoose');

const mongoDB = 'mongodb://127.0.0.1:27017/SoccerDB';
mongoose.Promise = global.Promise;
mongoose.connect(mongoDB)
    .then(() => {
        console.log('connection to mongodb worked!');
    })
    .catch((error) => {
        console.log('connection to mongodb did not work! ' + JSON.stringify(error));
    });
```

- **MVC Structure:**
    - **Models:** Define the schema and structure of the data stored in MongoDB.

```
const mongoose = require('mongoose');
const Schema = mongoose.Schema;

const gameLineupSchema = new Schema({
    game_id: { type: Number, required: true },
    club_id: { type: Number, required: true },
    type: { type: String, required: true },
    number: { type: Number, required: true },
    player_id: { type: Number, required: true },
    player_name: { type: String, required: true },
    team_captain: { type: Number, required: true },
    position: { type: String, required: true }
}, {
    collection: 'game_lineups',
    timestamps: true
});

const GameLineup = mongoose.model('GameLineup', gameLineupSchema);
module.exports = GameLineup;
```

- **Controllers:** Handle the business logic and data manipulation.

```
const GameLineup = require('../models/game_lineups');

exports.getGameLineupsByGameId = async (req, res) => {
    const { game_id } = req.body;
    try {
        const gameLineups = await GameLineup.find({ game_id: game_id });
        if (gameLineups.length === 0) {
            return res.status(404).json({ message: 'No game lineups found for this game ID' });
        }
        res.status(200).json(gameLineups);
    } catch (error) {
        res.status(500).json({ message: error.message });
    }
};
```

- **Routes:** Define the API endpoints and map them to the corresponding controller methods.

```
const express = require('express');
const router = express.Router();
const gameLineupsController = require('../controllers/gameLineupsController');

router.post('/lineupbygameid', gameLineupsController.getGameLineupsByGameId);

module.exports = router;
```

- **App Configuration:**

```
const express = require('express');
const path = require('path');
const logger = require('morgan');
const cookieParser = require('cookie-parser');

const app = express();
const gamesRouter = require('./routes/games');
const gameLineupsRouter = require('./routes/gameLineups');
const gameEventsRouter = require('./routes/gameEvents');
const careerRoutes = require('./routes/career');
const rankingsRoutes = require('./routes/rankings');

app.set('views', path.join(__dirname, 'views'));
app.set('view engine', 'ejs');

app.use(logger('dev'));
app.use(express.json());
app.use(express.urlencoded({ extended: false }));
app.use(cookieParser());
app.use(express.static(path.join(__dirname, 'public')));

app.use('/', require('./routes/index'));
app.use('/api', require('./routes/users'));
app.use('/api', gamesRouter);
app.use('/api', gameLineupsRouter);
app.use('/api', gameEventsRouter);
app.use('/api', careerRoutes);
app.use('/api', rankingsRoutes);

module.exports = app;
```

The server runs on `localhost:3002` and connects to MongoDB at `mongodb://127.0.0.1:27017/SoccerDB`.

Limitations:

Potential Limitations:

- **Scalability:** As the user base grows, the server might face challenges in managing a high volume of concurrent requests, necessitating optimization or scaling solutions.
- **Data Consistency:** Ensuring data consistency between multiple backend services can be complex and might require additional synchronization mechanisms.
- **Error Handling:** Robust error handling is required to manage potential failures in communication with backend services.

Extensibility:

- The modular structure of the client-side code and the use of well-defined API routes ensure that the server can be easily extended to accommodate new features and requirements.
- Integration of new client-side libraries or frameworks can be done with minimal disruption to existing functionality.

Overall, the secondary Express server is designed to provide a robust and scalable solution for managing dynamic data related to football statistics, ensuring efficient interaction between the main Express server and MongoDB.

Task 4: Spring Boot Server Design

Solution:

Design and its motivations:

The Spring Boot server is designed to manage the static data stored in PostgreSQL. This server uses a package structure based on the different data entities it handles, such as clubs, competitions, and nations. Each package contains the necessary components for that entity: controller, service, repository, and model. This structure ensures a clear separation of concerns and makes the application more modular and maintainable.

Key Features:

- **Package Structure:** Organizes code by data entities, making it easier to manage and scale.
- **Spring Data JPA:** Provides an abstraction over the data access layer, making database interactions more efficient and less error-prone.
- **RESTful APIs:** Exposes endpoints for CRUD operations on the data entities.

Advantages:

- **Modularity:** The package structure allows for easy maintenance and scalability by separating the business logic, data access, and presentation layers.
- **Efficiency:** Spring Boot and Spring Data JPA streamline the development process and improve performance.
- **Clarity:** Clear division of responsibilities within the codebase enhances readability and simplifies debugging.

Disadvantages:

- **Learning Curve:** Understanding Spring Boot and JPA can be challenging for beginners.
- **Overhead:** Managing multiple services and ensuring seamless communication between them can introduce additional overhead.

Issues:

Introduce the task:

This section refers to the challenge of designing a server (Spring Boot) that efficiently handles static data stored in PostgreSQL. The server must manage data related to clubs, competitions, and nations, providing an API for the main Express server to interact with.

Requirements:

The design of the Spring Boot server meets the requirements by ensuring efficient management of static data. The server setup includes:

- **Package Structure:**
    - **Clubs:**
        - **ClubsController:** Handles HTTP requests and responses.
        - **ClubsRepository:** Interfaces with the database.
        - **ClubsService:** Contains the business logic.
        - **Clubs:** Represents the data model.
    - **Competitions:**
        - **CompetitionsController:** Handles HTTP requests and responses.
        - **CompetitionsRepository:** Interfaces with the database.
        - **CompetitionsService:** Contains the business logic.
        - **Competitions:** Represents the data model.
    - **Nations:**
        - **NationsController:** Handles HTTP requests and responses.
        - **NationsRepository:** Interfaces with the database.
        - **NationsService:** Contains the business logic.
        - **Nations:** Represents the data model.

  Players:
    - **PlayersController:** Handles HTTP requests and responses.
    - **PlayersRepository:** Interfaces with the database.
    - **PlayersService:** Contains the business logic.
    - **Players:** Represents the data model.

  Players Statistic:
    - **PlayersStatController:** Handles HTTP requests and responses.
    - **PlayersStatRepository:** Interfaces with the database.
    - **PlayersStatService:** Contains the business logic.
    - **PlayersStat:** Represents the data model.

  Users:
    - **UsersController:** Handles HTTP requests and responses.
    - **UsersRepository:** Interfaces with the database.
    - **UsersService:** Contains the business logic.
    - **Users:** Represents the data model.
- **Example: Nations Package**
    - **Nations Model:**

```
package com.iumtweb.spring_server.nations;
import jakarta.persistence.*;

@Entity
@Table(name = "nations")
public class Nations {

    @Id
    @Column(name = "name", nullable = false)
    private String name;

    @Column(name = "sig", columnDefinition = "TEXT")
    private String sig;

    // Default constructor
    public Nations() {
    }

    // Constructor with parameters
    public Nations(String name, String sig) {
        this.name = name;
        this.sig = sig;
    }

    // Getters and setters
    public String getName() {
        return name;
    }

    public void setName(String name) {
        this.name = name;
    }

    public String getSig() {
        return sig;
    }

    public void setSig(String sig) {
        this.sig = sig;
    }
}
```

- **NationsController:**

```
package com.iumtweb.spring_server.nations;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.web.bind.annotation.GetMapping;
import org.springframework.web.bind.annotation.RestController;
import java.util.List;

@RestController
public class NationsController {

    @Autowired
    private NationsService nationsService;

    @GetMapping("/soccer-nations")
    public List<SoccerNations> getAllSoccerNations() {
        return nationsService.getSoccerNations();
    }
}
```

- **NationsRepository:**

```
package com.iumtweb.spring_server.nations;

import org.springframework.data.jpa.repository.JpaRepository;
import org.springframework.data.jpa.repository.Query;
import java.util.List;

public interface NationsRepository extends JpaRepository<Nations, String> {

    @Query("SELECT n.name as name, n.sig as sig FROM Nations n WHERE EXISTS (SELECT 1 FROM Competitions c WHERE c.country
    List<SoccerNations> findAllSoccerNations();
}
```

- **NationsService:**

```
package com.iumtweb.spring_server.nations;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.stereotype.Service;
import java.util.List;

@Service
public class NationsService {

    @Autowired
    private NationsRepository nationsRepository;

    public List<SoccerNations> getSoccerNations() {
        return nationsRepository.findAllSoccerNations();
    }
}
```

- **SoccerNations Interface:**

```
package com.iumtweb.spring_server.nations;

public interface SoccerNations {
    String getName(); // coming from Competitions
    String getSig(); // coming from Nations
}
```

- **Application Properties:**

```
server.port=8082
# DataSource settings: PostgreSQL
spring.datasource.url=jdbc:postgresql://localhost:5432/postgres
spring.datasource.username=massimoredomi
spring.datasource.password=

# Specify the JDBC driver for PostgreSQL
spring.datasource.driver-class-name=org.postgresql.Driver

# Hibernate properties
# You can configure additional Hibernate properties here if needed
spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.PostgreSQLDialect

# Hibernate ddl auto (create, create-drop, validate, update)
spring.jpa.hibernate.ddl-auto=validate

# Other configurations if required
spring.jpa.show-sql=true

logging.level.org.springframework.jdbc.core=DEBUG
logging.level.org.hibernate.SQL=DEBUG
logging.level.root=INFO
```

The server runs on `localhost:8082` and connects to PostgreSQL at `jdbc:postgresql://localhost:5432/postgres`.

Limitations:

Potential Limitations:

- **Scalability:** As the user base grows, the server might face challenges in managing a high volume of concurrent requests, necessitating optimization or scaling solutions.
- **Data Consistency:** Ensuring data consistency between multiple backend services can be complex and might require additional synchronization mechanisms.
- **Error Handling:** Robust error handling is required to manage potential failures in communication with backend services.

Extensibility:

- The modular structure of the code and the use of well-defined API routes ensure that the server can be easily extended to accommodate new features and requirements.
- Integration of new libraries or frameworks can be done with minimal disruption to existing functionality.

Overall, the Spring Boot server is designed to provide a robust and scalable solution for managing static data related to football statistics, ensuring efficient interaction between the main Express server and PostgreSQL.

Task 5: Chat System Implementation

Solution:

Design and its motivations:

The chat system is implemented using socket.io to facilitate real-time communication among fans and pundits. Socket.io is chosen for its ability to handle bi-directional communication between the server and clients, providing a seamless and interactive chat experience.

Key Features:

- **Real-Time Communication:** Enables real-time, bidirectional communication between web clients and servers.
- **Room-Based Chats:** Users can join specific chat rooms to discuss topics related to their favorite teams or competitions.
- **User-Friendly Interface:** The chat interface is designed to be intuitive, allowing users to easily send and receive messages.

Advantages:

- **Efficiency:** Socket.io efficiently manages real-time communication, ensuring minimal latency in message delivery.
- **Scalability:** The system can handle multiple chat rooms and a large number of users simultaneously.
- **Interactivity:** Provides a dynamic and interactive experience for users, enhancing engagement on the platform.

Disadvantages:

- **Complexity:** Implementing real-time communication introduces additional complexity in terms of connection management and message handling.
- **Learning Curve:** Understanding the workings of socket.io and integrating it with the main Express server can be challenging.

Issues:

Introduce the task:

This section refers to the challenge of implementing a chat system among fans and pundits using socket.io. The chat system needs to handle real-time communication, manage multiple chat rooms, and ensure a smooth user experience.

Requirements:

The design of the chat system meets the requirements by providing a robust solution for real-time communication. The setup includes:

- **Client-Side Implementation:** The client-side code initializes the chat interface, connects to the chat namespace on the server, and handles user interactions such as sending messages and joining rooms.
- **Server-Side Implementation (Main Express Server):** The server-side code sets up socket.io, handles user connections, manages chat rooms, and broadcasts messages to all users in a room.

```
exports.init = function(io) {
  const chat = io.of('/chat').on('connection', function (socket) {
    try {
      socket.on('create or join', function (room, userId) {
        socket.join(room);
        chat.to(room).emit('joined', room, userId);
      });

      socket.on('chat', function (room, userId, chatText) {
        chat.to(room).emit('chat', room, userId, chatText);
      });

      socket.on('disconnect', function(){
        console.log('someone disconnected');
      });
    } catch (e) {
      console.error('Socket error:', e);
    }
  });
}
```

Limitations:

Potential Limitations:

- **Scalability:** As the number of users and chat rooms increases, the server may face challenges in handling a high volume of concurrent connections, requiring load balancing and scaling solutions.
- **Network Reliability:** The performance of real-time communication heavily depends on network reliability. Unstable network conditions can affect the user experience.
- **Security:** Ensuring secure communication and preventing unauthorized access to chat rooms is crucial.

Extensibility:

- The chat system is designed to be extensible, allowing for the addition of new features such as private messaging, user authentication, and chat history.
- The modular structure of the code ensures that new functionalities can be integrated with minimal disruption to existing features.

Overall, the chat system implemented using socket.io provides a robust and scalable solution for real-time communication among fans and pundits, enhancing the interactivity and engagement on the JFT Soccer platform.

Task 6: Web Interface for Data Query

Solution:

Design and its motivations:

The web interface for data query was designed to allow users to easily and efficiently access and analyze football data stored in the databases. The interface is built using a combination of Bootstrap for responsive design, jQuery for DOM manipulation, and Axios for HTTP requests.

The main Express server serves the front-end assets and acts as an intermediary between the client and the back-end servers. This setup ensures a seamless user experience by providing quick and easy access to the data.

Key Features:

- **Bootstrap:** Provides a responsive and visually appealing design, ensuring the interface looks good on all devices.
- **jQuery:** Simplifies DOM manipulation, event handling, and AJAX requests.
- **Axios:** Used for making HTTP requests to the back-end servers, providing a promise-based API that works in the browser and Node.js.
- **Express Server:** Acts as a proxy, handling API requests from the client and forwarding them to the appropriate back-end server (either the secondary Express server connected to MongoDB or the Spring Boot server connected to PostgreSQL).

Advantages:

- **Responsive Design:** Using Bootstrap ensures that the interface is accessible on both desktop and mobile devices.
- **Efficient Data Handling:** Axios and jQuery make it easy to fetch and display data dynamically without requiring page reloads.
- **Scalability:** The separation of concerns between the front end and back end allows for easy scaling and maintenance.

Disadvantages:

- **Complexity:** Managing multiple libraries and ensuring they work together seamlessly can add complexity to the project.
- **Initial Load Time:** Serving static assets and initializing scripts may take some time, especially for first-time users.

Issues:

Introduce the task:

This section refers to the challenge of enabling the user to query the database via a web interface. Users need to be able to easily access and filter the football data stored in the databases through a user-friendly interface.

Requirements:

The design complies with the requirements by providing a robust and efficient interface for querying the football data. The interface allows users to:

- View and search for clubs, competitions, and player statistics.
- Filter data based on various criteria such as competition, club, game or player.
- Display the results in a user-friendly format, with the ability to drill down into detailed views.

Limitations:

Potential Limitations:

- **Performance Issues:** As the amount of data grows, the performance of the interface might degrade, requiring optimization and possibly pagination or lazy loading techniques.
- **User Input Handling:** Ensuring that all user inputs are properly sanitized to prevent SQL injection or other security vulnerabilities.
- **Data Consistency:** Ensuring that the data displayed in the interface is always up to date with the latest data from the databases.

Extensibility:

- The modular design of the interface allows for easy addition of new features or data queries.
- The use of well-known libraries and frameworks ensures that other developers can easily understand and extend the codebase.
- Future enhancements could include advanced filtering options, integration with other data sources, or improved visualization tools.

Experience and Refactoring:

For this project, the most challenging part for me was understanding how to build the web interface, particularly the JavaScript and its capabilities. I went through three major refactors to improve the code structure and maintainability.

1. **Initial Approach:** Initially, I created a large function with different cases inside for each event triggered. This "father function" managed all the web interface interactions, calling the main function with different values. Depending on these values, a switch case was triggered to call the main component that needed to change. However, this approach resulted in a massive, hard-to-maintain function that was difficult to debug and extend.

2. **First Refactor:** I divided the main function into smaller functions, each responsible for a macro area such as nations, clubs, or games. While this improved readability, it still had issues. Different components managing the same data sometimes required different formatting styles and displayed different aspects of the same data.

3. **Second Refactor:** I then refactored the code to think of each component as an object that interacts with other objects. Each object had its own data-action function, performing specific tasks and calling other objects if necessary. This object-oriented approach made the code more modular and maintainable. After this refactor, I started studying React, as it follows a similar component-based approach, but I didn't have time to implement it in this project. I plan to use React in my next project.

Interactivity and Usability:

I aimed to make the JFT Soccer web page highly interactive, with many components that not only display information but are also clickable and interactive. Users can access the same information from different components or through different selections of components. Additionally, a search bar allows users to search by competition, club, or nation.

For example:

- **Nation:** Clicking on a nation displays all the champions, games, and players from that nation.

- **Competition:** Clicking on a competition displays the nation, games, and players of that competition.
- **Club:** Clicking on a club displays the nation of the club, the competition of the club, and the players of the club.

This interconnected design ensures that users have multiple ways to interact with the data, enhancing the overall user experience.

Task 7: Jupyter Notebooks for Data Analysis

Solution:

Design and its motivations:

In this project, I utilized Jupyter Notebooks to analyze the football data provided. Jupyter Notebooks are an excellent tool for data analysis due to their interactive nature, allowing me to write and execute code in a cell-based format, visualize results instantly, and document the analysis process alongside the code. For this analysis, I used pandas for data manipulation and analysis, and various other libraries such as matplotlib and seaborn for creating charts and graphics.

Key Features:

- **Interactive Analysis:** Jupyter Notebooks provide an interactive environment where code, text, and visualizations can be combined.
- **Data Manipulation:** Pandas is used extensively for cleaning, transforming, and analyzing the data.
- **Visualization:** Libraries like matplotlib and seaborn are used to create detailed charts and graphs to visualize the data insights.
- **Documentation:** The notebook format allows for thorough documentation of the analysis process, making it easy to follow and understand.

Advantages:

- **Ease of Use:** The interactive nature of Jupyter Notebooks makes it easy to test and iterate on code, immediately seeing the results of data transformations and visualizations.
- **Rich Visualizations:** Using matplotlib and seaborn allows for the creation of a wide variety of visualizations to explore and present data insights effectively.
- **Documentation:** Combining code with markdown cells enables clear documentation, making the analysis easy to understand and reproduce.

Disadvantages:

- **Performance:** Jupyter Notebooks can become slow with very large datasets or complex computations.
- **Version Control:** Managing version control with notebooks can be challenging compared to traditional scripts due to their cell-based structure.

Issues:

Introduce the task:

This section refers to the challenge of creating Jupyter Notebooks that enable analysis of the football data. The task involved cleaning and processing the data, performing exploratory data analysis (EDA), and creating visualizations to uncover insights.

Requirements:

The design of the Jupyter Notebooks complies with the requirements by providing a comprehensive and interactive analysis of the football data. The notebooks include:

- **Data Cleaning and Preprocessing:** Steps to clean and preprocess the data, handling missing values, and transforming data into a usable format.
- **Exploratory Data Analysis (EDA):** Detailed analysis to explore the data, including summary statistics and initial visualizations.
- **Advanced Analysis:** In-depth analysis to uncover trends and patterns in the data, supported by various visualizations.
- **Documentation:** Thorough documentation of each step of the analysis process, ensuring that the methodology is clear and reproducible.

Limitations:

Potential Limitations:

- **Performance Issues:** Handling very large datasets can be slow and memory-intensive, which might require optimizing the code or using more powerful computational resources.
- **Complex Visualizations:** Creating very complex visualizations might require additional libraries or more advanced coding techniques.
- **Collaboration:** While Jupyter Notebooks are excellent for individual analysis, collaborating on notebooks can be challenging due to their cell-based structure and potential merge conflicts in version control.

Extensibility:

- The modular nature of the analysis allows for easy addition of new analysis steps or visualizations as new data becomes available or as new questions arise.
- The use of popular libraries such as pandas, matplotlib, and seaborn ensures that the notebooks can be extended with additional functionalities provided by these libraries.

Experience and Challenges:

Creating the Jupyter Notebooks for this project was both a rewarding and challenging experience. One of the primary challenges was understanding how to effectively clean and preprocess the data. The datasets contained a lot of missing values and inconsistencies, which required extensive use of pandas to handle.

Initially, I struggled with how to structure the notebook. I started with a linear approach, writing all the code sequentially. However, this quickly became unmanageable as the analysis grew in complexity. I realized the importance of breaking down the analysis into smaller, manageable sections, each focusing on a specific aspect of the data.

Another significant challenge was the visualization of the data. I initially used basic matplotlib plots, but they did not convey the insights as effectively as I wanted. I then explored seaborn, which provided more advanced and aesthetically pleasing visualization options. Learning how to use seaborn effectively took some time, but it was worth the effort as it greatly enhanced the clarity of the visualizations.

Additionally, integrating different data sources and ensuring data consistency across these sources was a complex task. I had to write custom functions to merge and align the data properly, which required a deep understanding of both the data and the pandas library.

Despite these challenges, the process of iterating on the analysis, visualizing the data, and documenting my findings in the notebook was highly educational. It allowed me to see the immediate impact of my code changes and adjust my approach in real-time.

Looking forward, I plan to continue using Jupyter Notebooks for data analysis projects and to explore more advanced features and libraries to further enhance my analysis capabilities. This experience has also inspired me to learn more about machine learning and how I can apply it to further analyze and predict trends in football data.

Conclusions

Throughout the process of designing and implementing the various components of this project, several key conclusions and lessons emerged. This project provided valuable insights into the complexities and challenges of developing a comprehensive data-driven web application.

Key Conclusions:

1. **Modular Design and Scalability:**

- Dividing the application into distinct tasks, each handled by specialized servers or components, proved to be highly effective. The main Express server, secondary Express server connected to MongoDB, and Spring Boot server connected to PostgreSQL each handled specific responsibilities. This modular design not only improved scalability but also made the system more manageable and maintainable.

2. **Interactive and User-Friendly Web Interface:**

- The web interface was designed to be highly interactive and user-friendly, allowing users to easily query and navigate through the football data. The use of Bootstrap, jQuery, and Axios enabled the creation of a responsive and dynamic interface that provided a seamless user experience.

3. **Real-Time Communication with Socket.io:**

- Implementing the chat system using socket.io demonstrated the importance of real-time communication in enhancing user engagement. Despite the initial complexity, socket.io proved to be an excellent choice for handling real-time, bidirectional communication between the server and clients.

4. **Data Analysis with Jupyter Notebooks:**

- Using Jupyter Notebooks for data analysis provided a powerful and flexible tool for exploring and visualizing the football data. The combination of pandas for data manipulation and matplotlib/seaborn for visualizations enabled a comprehensive analysis, uncovering valuable insights from the data.

5. **Challenges and Learning Opportunities:**

- Each component of the project presented its own set of challenges. From understanding and implementing socket.io for real-time communication, to managing data consistency across multiple databases, to structuring and refactoring the web interface code, each challenge provided a learning opportunity that contributed to the overall success of the project.