NBA Data Analysis Project
Design Document

List of Contributors

- **Tunahan Oğuz** (Backend Developer / Tester)
- **Beyzanur Zeybek** (Requirements Analyst / Frontend Developer)
- **Alkım Doryan** (Project Manager / Tester / Scrum Master)
- **Ali Eren Kurt** (Frontend Developer / Product Owner)

Task Matrix

Task ID	Task Description	Responsible	Completion Status	Notes
D1	Draft overall structure & sections	Beyzanur, Alkım	Completed	Followed PA2 guidelines for structure
D2	Compile system overview from project documents	Tunahan, Ali Eren	Completed	Summarized scope, architecture, tech stack
D3	Detail implementation details & codebase layout	Tunahan, Beyzanur	Completed	Included folder structures, key modules
D4	Map use cases & requirements to the design	All team members	Completed	Chose 4 primary use cases from the Requirements Doc
D5	Write design decisions & technology comparisons	Alkım	Completed	Documented alternatives & justification
D6	Format final Design Document	Beyzanur, Alkım	Completed	Ensured consistent layout & references
D7	Final review & approval	All team members	Completed	Verified alignment with project documents

1. System Overview..... 2
1.1 Brief Project Description..... 2
1.2 System Architecture..... 2
1.3 Technology Stack..... 2
2. Implementation Details..... 3
2.1 Codebase Structure..... 3
2.2 Key Implementations..... 3
2.3 Component Interfaces..... 4
2.4 Visual Interfaces..... 4
3. Use Case Support in Design..... 4
3.1 Use Case Selection..... 4
3.2 Requirement Mapping..... 5
3.3 Use Case Design..... 5
3.4 Demo Requirement..... 6
4. Design Decisions..... 6
4.1 Technology Comparisons..... 6
4.2 Decision Justifications..... 6

1. System Overview

1.1 Brief Project Description

The NBA Data Analysis Project is a web-based platform that aggregates and visualizes historical NBA data from Kaggle. Users can explore player statistics, compare performances, generate custom reports, and view team analytics. The goal is to create an intuitive and engaging interface for basketball enthusiasts, analysts, and casual fans to gain insights into player and team performance.

1.2 System Architecture

The system follows a multi-tier (layered) architecture:

- **Frontend (Presentation Layer)**: A React.js (or similar JavaScript framework) client that renders charts and tables, provides filters, and handles user interaction.
- **Backend (Application Layer)**: A Python-based server (using Flask or Django) that processes requests, orchestrates data queries, and applies business logic.
- **Database Layer**: A relational database (e.g., MySQL or PostgreSQL) holding cleaned and processed NBA data.
- **Data Processing**: Python libraries (Pandas, NumPy) used for data cleaning and transformation; integrated into either the backend layer or a separate data preprocessing pipeline.

1.3 Technology Stack

- **Frontend**: React.js, HTML5, CSS3, JavaScript
- **Backend**: Python (Flask or Django), possibly a lightweight REST API architecture
- **Database**: MySQL or PostgreSQL

- **Visualization**: Plotly, Matplotlib, or Chart.js integrated with React
- **Data Processing**: Pandas, NumPy
- **Version Control**: GitHub for source control and team collaboration

2. Implementation Details

2.1 Codebase Structure

A typical repository structure might look like this:

```
/NBA-Data-Analysis
  ├── backend
  │   ├── app.py          # Flask/Django entry point
  │   ├── requirements.txt
  │   ├── data_processing/
  │   │   ├── cleaning.py
  │   │   └── transformations.py
  │   └── services/
  │       ├── ingestion_service.py
  │       └── visualization_service.py
  ├── frontend
  │   ├── package.json
  │   ├── src/
  │   │   ├── components/
  │   │   ├── pages/
  │   │   └── services/
  │   └── public/
  ├── docs
  │   └── (Design Document, QA Plan, etc.)
  ├── tests
  │   ├── backend_tests/
  │   ├── frontend_tests/
  │   └── integration_tests/
  └── README.md
```

- **backend/** contains application logic (routes, data processing, business logic).
- **frontend/** contains React components, pages, and styling.
- **docs/** contains project documentation.
- **tests/** holds unit tests and integration tests for both frontend and backend.

2.2 Key Implementations

1. **Data Ingestion & Cleaning**
    - Python scripts (inside data_processing/) for reading CSV files from Kaggle, cleaning missing values, and standardizing data formats.
    - These scripts feed the cleaned dataset into the database.
2. **Data Integration**
    - A schema that defines relationships across tables (e.g., player_id in multiple tables).

- An ORM (Object-Relational Mapping) or direct SQL queries for joining data as needed.

3. **Dashboard & Visualization**
    - Dynamic charts in the frontend (React + Plotly/Chart.js) that fetch data via REST endpoints.
    - Real-time filtering, sorting, and searching.
4. **Reporting**
    - Backend endpoints to generate PDF/CSV/HTML reports using libraries like pdfkit or WeasyPrint.
    - Automated generation triggered either by user request or scheduled job.
5. **Query Functionality**
    - A user interface that allows custom filters (date ranges, team, player) and triggers REST calls to the backend for aggregated results.

2.3 Component Interfaces

1. **Backend Endpoints**:
    - GET /api/players: Returns a list of players, possibly with pagination.
    - GET /api/players/<id>: Fetches details for a single player.
    - GET /api/teams: Returns a list of teams.
    - GET /api/reports/<type>: Generates and returns specified report (PDF, CSV, etc.).
    - POST /api/query: Accepts a JSON payload specifying filters/queries and returns results.
2. **Frontend Services** (React):
    - fetchPlayers(): Calls GET /api/players.
    - fetchTeams(): Calls GET /api/teams.
    - generateReport(type): Calls GET /api/reports/<type>.
    - runCustomQuery(filters): Calls POST /api/query.

2.4 Visual Interfaces

Wireframes (High-Level Description):

1. **Homepage**: Displays high-level stats (top scorers, top rebounders, etc.) with quick links.
2. **Player Dashboard**: Search bar to find a player, dynamic charts for performance (points, assists, rebounds).
3. **Team Dashboard**: Filter by season, display aggregated team stats (wins, losses, average points).
4. **Comparison View**: Compare two or more players side by side.
5. **Report Generation**: Export button that triggers PDF/CSV generation.

3. Use Case Support in Design

3.1 Use Case Selection

From the Requirements Document, we have selected four critical use cases:

1. **Data Ingestion** (Functional Requirement #1)
2. **Visualization Dashboard** (Functional Requirement #5)
3. **Automated Reporting** (Functional Requirement #6)
4. **Data Export** (Functional Requirement #7)

These four use cases showcase the essential functionalities of the system and will be the focus of the final demo.

3.2 Requirement Mapping

1. **Data Ingestion:** Maps to the requirement that the system must import data from multiple CSV/database files.
2. **Visualization Dashboard:** Maps to the requirement for an interactive dashboard that allows dynamic charts and filtering.
3. **Automated Reporting:** Maps to the requirement of generating automated reports (PDF/HTML).
4. **Data Export:** Maps to the requirement for exporting cleaned or processed data to CSV/Excel.

3.3 Use Case Design

1. **Data Ingestion**
    - **Flow**: (1) System reads CSV → (2) Validate & clean → (3) Insert into database.
    - **Architecture**: Implemented primarily in backend/data_processing/.
    - **State changes**: Database gets updated with new or cleaned data.
2. **Visualization Dashboard**

Flow: (1) User loads dashboard → (2) Frontend calls backend APIs → (3) Receives data → (4) Charts render.

- **Architecture**: React components + REST endpoints.
- **Interactions**: User manipulates filters, triggers new data fetch.

3. **Automated Reporting**

- **Flow**: (1) User clicks “Generate Report” → (2) Backend compiles relevant data → (3) Renders data in PDF/HTML → (4) Returns file to the user.
- **Architecture**: Python library (e.g., pdfkit).
- **Data Flow**: Collates results from multiple database tables for a comprehensive summary.

4. **Data Export**

- **Flow**: (1) User chooses data subset → (2) System queries database → (3) Formats output in CSV/Excel → (4) Prompts user to download.

- **Architecture**: Shared with the general reporting structure, but with simpler CSV/Excel generation logic.

3.4 Demo Requirement

These four use cases must be **fully implemented and demonstrated** at the end of the semester. During the demo, we will:

- Show the data ingestion workflow and how the database is updated.
- Present the dashboard with real-time filtering and chart updates.
- Generate an automated PDF report containing selected statistics.
- Export the processed data subset to CSV.

4. Design Decisions

4.1 Technology Comparisons

1. **Flask vs. Django (Backend Framework)**

- **Flask**: Lightweight, easy to set up, minimal boilerplate.
- **Django**: Batteries-included, built-in ORM, robust structure for larger projects.

2. **MySQL vs. PostgreSQL (Database)**

- **MySQL**: Very common, slightly simpler for basic setups.
- **PostgreSQL**: Advanced features, strong data integrity, well-suited for complex queries.

4.2 Decision Justifications

1. **Python + Flask**

- Team members have prior experience with Python; it's faster to prototype.
- This project doesn't initially require Django's full-scale features.

2. **MySQL**

- Good for handling complex queries and ensures better data integrity.

3. **React**

- Familiar to the team; easy component-driven design.

4. **Plotly/Chart.js**

- Highly interactive and widely supported with React wrappers.