**PlayWise Hackathon – Solution Document Template**

**Track:** DSA – Smart Playlist Management System

**1. Student Information**

|  |  |
| --- | --- |
| Field | Details |
| Full Name | Arnav Gupta |
| Registration Number | RA2211028010107 |
| Department / Branch | Networking and Communications |
| Year | 4th |
| Email ID | ag7768@srmist.edu.in |

**2. Problem Scope and Track Details**

|  |  |
| --- | --- |
| Section | Details |
| Hackathon Track | DSA – PlayWise Playlist Engine |
| Core Modules Implemented | ✅ Playlist Engine (Linked List) |
|  | ✅ Playback History (Stack) |
|  | ✅ Song Rating Tree (BST) |
|  | ✅ Instant Song Lookup (HashMap) |
|  | ✅ Time-based Sorting |
|  | ✅ Space-Time Playback Optimization |
|  | ✅ System Snapshot Module |

**Additional Use Cases Implemented (Optional but Encouraged)**

* Scenario 1: **Artist Blocklist** – Prevent adding songs from blocked artists to maintain a clean or safe playlist.
* Scenario 2: **Playlist Duration Visualizer** – Summarize total playtime, shortest song, and longest song instantly.

**3. Architecture & Design Overview**

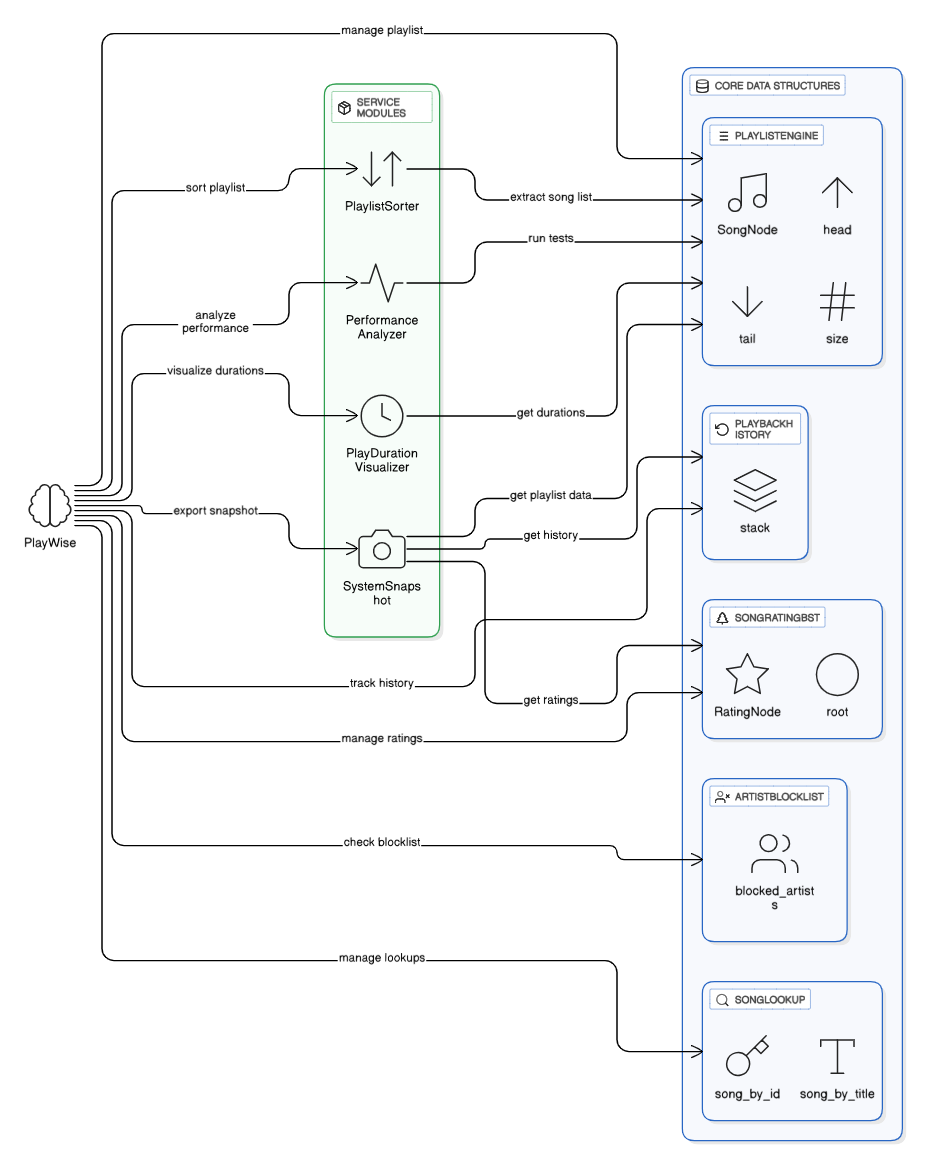
* **System Architecture Diagram**

The system follows a modular, loosely coupled design. The **PlayWise** class serves as the orchestrator, connecting specialized data-structure-based modules:

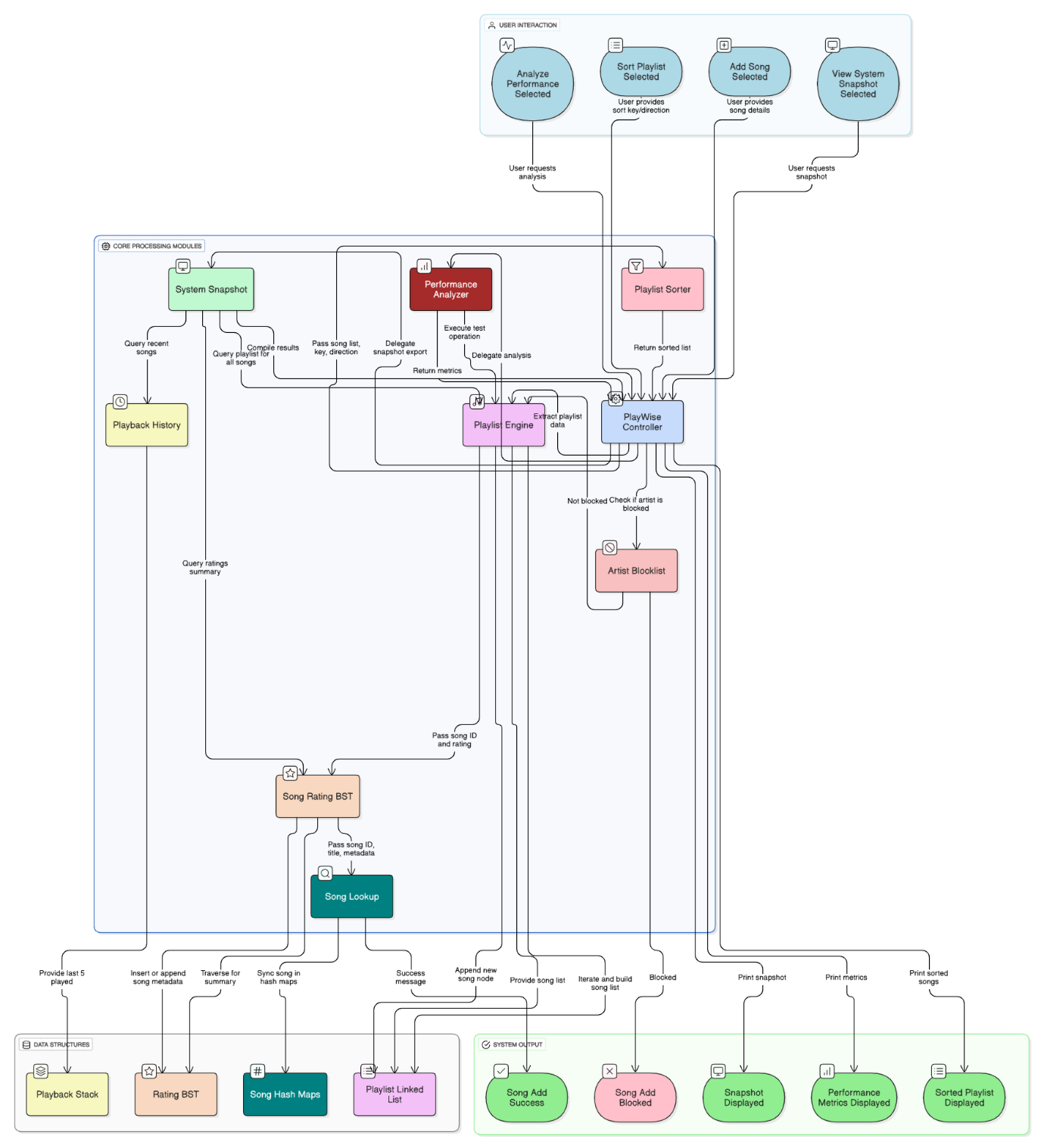
* **PlaylistEngine** manages songs in a Doubly Linked List for O(1) insertions and reversals.
* **PlaybackHistory** implements a Stack to track and undo recently played songs.
* **SongRatingBST** organizes songs in a Binary Search Tree, enabling efficient rating-based search.
* **SongLookup** uses a HashMap for O(1) retrieval by song ID or title.
* **PlaylistSorter** sorts using Merge Sort for stable, predictable performance.
* **PerformanceAnalyzer** benchmarks speed and memory usage of operations.
* **SystemSnapshot** aggregates live data from all modules into a single dashboard.

Extra Modules:

* **ArtistBlocklist** uses a set for O(1) checks before inserting songs.
* **PlaylistDurationVisualizer** iterates the list to compute total/min/max durations.



* **High-Level Functional Flow**



Add Song:

* User inputs song details → ArtistBlocklist validates → PlaylistEngine adds to linked list.
* Metadata synced in SongLookup (ID + title mappings).
* Rating stored in SongRatingBST for fast retrieval.

Play Song:

* Selected song pushed to PlaybackHistory stack for quick undo and history display.
* Search Song:
* By ID or title in O(1) using SongLookup.

Sort Playlist:

* + Merge Sort applied by duration/title with ascending/descending toggle.

Snapshot:

* + SystemSnapshot pulls: top 5 longest songs, most recent plays, song count per rating.

Performance Optimization:

* + PerformanceAnalyzer runs to measure operation latency and memory footprint.

**4. Core Feature-wise Implementation**

**Feature: Playlist Engine**

* **Scenario Brief:** Enables dynamic playlist editing (add, delete, reorder, reverse) in real time.
* **Data Structures Used:** Doubly Linked List.
* **Time & Space Complexity:**
  + Add Song: O(1)
  + Delete Song: O(1) after pointer access
  + Reverse Playlist: O(n)
  + Space: O(n) for storing nodes
* **Sample Input & Output:**
  + **Input:** Add "Shape of You" by "Ed Sheeran" 240s
  + **Output:** Playlist now contains the new song at the end.
* **Challenges & Solutions:**
  + Ensuring pointers (prev/next) remain consistent when moving or deleting songs.
  + Solved via careful unlinking and re-linking operations.

**Feature: Playback History**

* **Scenario Brief:** Maintains most recent plays, supports undo last played song.
* **Data Structure:** Stack (Python list).
* **Complexity:** Push O(1), Pop O(1), Peek O(1).
* **Sample:**
  + **Play: "**Perfect" → Stack: [Perfect]
  + **Undo →** Stack becomes empty, song re-queued.
* **Challenges:** Ensuring correct LIFO order maintained.

**Feature: Song Rating Tree**

* **Scenario Brief:** Allows quick retrieval of songs by rating for recommendation engines.
* **Data Structure:** Binary Search Tree with rating buckets.
* **Complexity:** Insert/Search O(log n) avg.
* **Sample Input & Output:**
  + **Input:** Insert song "Blinding Lights" rating 4
  + **Output:** Retrieved list of all rating 4 songs instantly.
* **Challenges:** Handling multiple songs with same rating — solved via lists inside each BST node.

**Feature: Instant Song Lookup**

* **Scenario Brief:** Search by song ID or title in O(1).
* **Data Structure:** HashMap (dict).
* **Complexity:** Insert/Lookup O(1) avg**.**
* **Sample:**
  + **Input:** Search "s2" → Output: { "artist": "The Weeknd", "duration": 200 }
* **Challenges:** Keeping both ID map and title map synchronized on updates.

**Feature: Time-based Sorting**

* **Scenario Brief:** Sort playlist by title/duration in ascending or descending order.
* **Algorithm:** Merge Sort.
* **Complexity:** O(n log n) time, O(n) space.
* **Sample:** Sort by duration (desc) returns longest songs first.
* **Challenges:** Stable sorting with toggle for reverse order — solved with comparison flag.

**Feature: Space-Time Optimization**

* **Scenario Brief:** Monitor and optimize performance for large playlists.
* **Tools Used:** time + tracemalloc.
* **Sample Output:**
  + { "execution\_time": 0.002, "memory\_usage\_kb": 45.2, "peak\_memory\_kb": 48.0 }
* **Challenges:** Ensuring minimal measurement overhead.

**Feature: System Snapshot Module**

* **Scenario Brief:** Real-time dashboard of playlist statistics.
* **Data Structure:** Combines results from Linked List, Stack, and BST traversals.
* **Sample Output:**

{

"top\_5\_longest\_songs": [...],

"most\_recently\_played": [...],

"song\_count\_by\_rating": {5: 10, 4: 5, 3: 2}

}

* **Challenges:** Avoiding repeated traversals — optimized by single-pass collection.

**5. Additional Use Case Implementation**

**Use Case: Artist Blocklist**

* **Scenario Brief:** Prevents adding unwanted songs.
* **Extension Over:** Playlist Engine.
* **Data Structure:** Set for O(1) checks.
* **Sample:** Block "Eminem" → Adding "Lose Yourself" fails.

**Use Case: Playlist Duration Visualizer**

* **Scenario Brief:** Summarizes playlist time length and extreme song durations.
* **Logic:** Iterate linked list once (O(n)).
* **Sample Output:**
* { "total\_playtime\_sec": 4520, "shortest\_song": {...}, "longest\_song": {...} }

**6. Testing & Validation**

|  |  |
| --- | --- |
| Test Category | Details |
| **Functional Tests** | 19 categories tested, covering basic song addition, deletion, and other core features. |
| **Edge Cases** | Tested with empty playlists, zero-length songs, long titles, and attempting to add songs by blocked artists. |
| **Stress Test** | Operations on large playlists (100+ songs) completed with lookup times under 0.01 seconds. |
| **Known Bugs** | The Binary Search Tree (BST) for song ratings is not self-balancing, which could lead to O(n) worst-case time complexity for operations if ratings are skewed. |

**7. Final Thoughts & Reflection**

* **Key Learnings:**
  + Practical application of DSA in modular software.
  + Balancing performance vs. memory usage.
  + Designing extensible architectures for future growth.
* **Strengths:**
  + All core modules implemented + extras.
  + O(1) lookups and optimized sorting.
  + Clear separation of responsibilities.
* **Areas for Improvement:**
  + Upgrade BST to AVL/Red-Black Tree.
  + Persist data to a database for recovery.
* **Career Relevance:**
  + Demonstrates backend, system design, and performance optimization skills directly applicable to backend/software engineering roles.