AI-Enhanced Music Mashup Studio - Complete Design Document

Visual Design & Component Mapping

Overall Layout Philosophy

A single-page application with a persistent header, a main central workspace showing up to 5 tracks vertically stacked with waveforms, and a collapsible AI suggestions sidebar. Each track can be expanded to show detailed controls, and tracks can be reordered via drag-and-drop.

Screen Layout with Component Names

+---+

| <Header> |

| [Logo] "Summer Mix 2026" [Save] [Load] [Export] [Preview Full Mix] [Profile] |

+---+

| |

| <MainWorkspace> (Center - Scrollable) | <AISidebar> (Collapsible →) |

| | |

| <TrackStack> | <AIPanel> |

| ┌─────────────────────────────────────┐ | 🤖 AI Suggestions |

| │ <TrackCard> │ | |

| │ <TrackHeader> │ | <AIContext> |

| │ "Song Name" - Artist [↑][↓]│ | Based on your current mix: |

| │ 128 BPM | Key: Am [⚙][×] │ | |

```
| │ </TrackHeader>              │  | <SuggestionList>          |
| │                             │  | <SuggestionCard>          |
| │ <WaveformDisplay>           │  | • Track X                 |
| │ [Waveform═══════════════════]│  |  BPM: 130 | Key: Cm      |
| │ </WaveformDisplay>          │  |  Match: 92% ⭐            |
| │                             │  |  "Great harmonic match!"  |
| │ <SettingsToggle>            │  |  [+ Add to Mix]           |
| │ ▼ Settings (Click to expand) │  | </SuggestionCard>         |
| │ </SettingsToggle>           │  |                           |
| │ </TrackCard>                │  | <SuggestionCard>          |
| └─────────────────────────────┘  | • Track Y                 |
|     <DragHandle>                 |  BPM: 126 | Key: G        |
| ┌─────────────────────────────┐  |  Match: 85%               |
| │ <TrackCard expanded>        │  |  "Smooth BPM transition"  |
| │ <TrackHeader>               │  |  [+ Add to Mix]           |
| │ "Song Name" - Artist  [↑][↓]│  | </SuggestionCard>         |
| │ 130 BPM | Key: Cm     [⚙][×] │  | </SuggestionList>         |
| │ </TrackHeader>              │  |                           |
| │                             │  | <RefreshButton>           |
| │ <WaveformDisplay />         │  | [Refresh Suggestions]     |
| │                             │  | </RefreshButton>          |
| │ <TrackSettings>             │  |                           |
| │ ▲ Settings (Expanded)       │  | <CollapseButton>          |
| │ <SegmentManager>            │  | [Collapse ←]              |
| │ ┌──────────────────────────────────────────┐ │  | </CollapseButton>          |
| │ │ <SegmentList>            │ │  | </AIPanel>            |
| │ │ Segments (Can add multiple)  │ │
+────────────────────────────────────────────+
| │ │ <SegmentItem>            │ │
| │ │ Segment 1: Bar 1-32 (32 bars) │ │
| │ │ </SegmentItem>           │ │
| │ │ <SegmentItem>            │ │
| │ │ Segment 2: Bar 48-64 (16 bars) │ │
| │ │ </SegmentItem>           │ │
| │ │ <AddSegmentButton>        │ │
| │ │ [+ Add Segment]          │ │
| │ │ </AddSegmentButton>       │ │
| │ │ </SegmentList>           │ │
| │ └──────────────────────────────────────────┘ │
| │                     │ │
| │ <BasicControls>          │ │
| │ │ Volume: <VolumeSlider />    │ │
| │ │ Fade In: <FadeInput />     │ │
| │ │ Fade Out: <FadeInput />     │ │
| │ </BasicControls>         │ │
| │                     │ │
| │ <AudioAdjustments>         │ │
| │ │ Pitch: <PitchControl />     │ │
| │ │ Speed: <SpeedControl />     │ │
| │ </AudioAdjustments>              │ │
| │                                  │ │
| │ <CollapsibleEQ>                  │ │
| │ │ ▼ EQ (Click to expand)         │ │
| │ │ <EQControls />                 │ │
| │ </CollapsibleEQ>                 │ │
| │                                  │ │
| │ <CollapsibleEffects>             │ │
| │ │ ▼ Effects (Click to expand)    │ │
| │ │ <EffectsPanel />               │ │
| │ </CollapsibleEffects>            │ │
| │ </TrackSettings>                 │ │
| │ </TrackCard>                     │ │
| └──────────────────────────────────────────────┘
|
| <AddTrackButton>
| [+ Add Track (3/5)]
| </AddTrackButton>
| </TrackStack>
| </MainWorkspace>
|
+-----------------------------------------------------------------------------+


<TrackSearchModal> (Opens when Add Track clicked)
```

<SpotifyAuthModal> (Opens on first load if not authenticated)

<LoadPlaylistModal> (Opens when Load clicked)

Complete Component Architecture

1. App Level Components

`App.js`

Purpose: Root component, manages global state and routing
Responsibilities:

- Manages authenticated user state
- Manages current playlist data
- Manages array of up to 5 tracks
- Handles loading states
- Coordinates between Firebase auth and Spotify auth

Child Components:

- `<AuthProvider>`
- `<Header>`
- `<MainWorkspace>`
- `<AISidebar>`

Firebase/Firestore Integration:

- Listens to authentication state changes
- Loads user's playlists on component mount
- Automatically syncs playlist changes to Firestore

Spotify API Integration:

- Initializes Spotify SDK on mount
- Manages access token refresh logic

`AuthProvider.js`

Purpose: Context provider for authentication state across the app
Responsibilities:

- Provides current user information to all child components
- Manages Spotify access token
- Handles authentication status

Exposed Methods:

- `login()` - Handles both Firebase and Spotify OAuth
- `logout()` - Clears both Firebase and Spotify sessions
- `refreshSpotifyToken()` - Refreshes expired Spotify tokens

Firebase/Firestore Integration:

- Uses Firebase auth state listener
- Stores Spotify tokens in Firestore user document

Spotify API Integration:

- Manages OAuth flow to obtain access token
- Handles token refresh when expired

---

2. Header Components

`Header.js`

Purpose: Top navigation bar with playlist controls
Responsibilities:

- Displays current playlist name
- Provides save/load/export functionality
- Shows user profile
- Allows full playlist preview

Child Components:

- `<PlaylistNameEditor>`
- `<SaveButton>`
- `<LoadButton>`
- `<ExportButton>`
- `<PreviewButton>`
- `<UserProfile>`

Firebase/Firestore Integration:

- Save button triggers Firestore write
- Load button opens modal that reads from Firestore

`PlaylistNameEditor.js`

Purpose: Inline editable playlist name
Responsibilities:

- Displays playlist name
- Allows click-to-edit functionality
- Saves changes on blur or enter key

Firebase/Firestore Integration:

- Updates playlist name in Firestore when changed

`SaveButton.js`

Purpose: Save current playlist to database
Responsibilities:

- Validates playlist data before saving
- Shows loading state during save
- Displays success/error messages

Firebase/Firestore Integration:

- Writes entire playlist structure to user's playlist collection

---

`LoadButton.js`

Purpose: Opens modal to load saved playlists
Responsibilities:

- Triggers load playlist modal
- Handles loading state

`ExportButton.js`

Purpose: Export playlist to external format
Responsibilities:

- Generates shareable playlist data
- Future: Export to Spotify, download file, etc.

`PreviewButton.js`

Purpose: Preview entire playlist with transitions
Responsibilities:

- Plays all tracks in sequence
- Shows playback progress
- Applies all audio settings during preview

Spotify API Integration:

- Uses Spotify playback SDK or Web Audio API for playback

`UserProfile.js`

Purpose: User avatar and account menu
Responsibilities:

- Displays user information
- Provides logout option
- Links to settings/preferences

Child Components:

- `<ProfileDropdown>`

Firebase/Firestore Integration:

- Displays user data from Firebase Auth
- Logout triggers Firebase sign out

---

3. Main Workspace Components

`MainWorkspace.js`

Purpose: Central container for all tracks
Responsibilities:

- Manages array of track objects (max 5)
- Coordinates track operations (add, remove, reorder)

Child Components:

- `<TrackStack>`
- `<AddTrackButton>`

Firebase/Firestore Integration:

- Updates Firestore when tracks are reordered or removed

`TrackStack.js`

Purpose: Renders vertical list of tracks with drag-and-drop
Responsibilities:

- Displays all tracks in order
- Enables drag-and-drop reordering
- Manages drag handles between tracks

Child Components:

- `<TrackCard>` (one per track)
- `<DragHandle>` (between cards)

`TrackCard.js`

Purpose: Container for individual track with all controls
Responsibilities:

- Displays track information
- Shows/hides detailed settings
- Manages track playback preview
- Handles track-specific state

Child Components:

- `<TrackHeader>`
- `<WaveformDisplay>`
- `<SettingsToggle>`
- `<TrackSettings>` (conditional on expanded state)

Spotify API Integration:

- Fetches track metadata on mount
- Retrieves audio analysis for waveform

`TrackHeader.js`

Purpose: Track title bar with metadata and controls
Responsibilities:

- Displays song name and artist
- Shows BPM and key
- Provides reorder and remove buttons
- Toggle settings panel

Child Components:

- `<TrackInfo>`
- `<TrackMetadata>`
- `<TrackActions>`

Spotify API Integration:

- Displays data retrieved from Spotify audio features endpoint

---

`WaveformDisplay.js`

Purpose: Visual representation of track audio
Responsibilities:

- Renders waveform visualization
- Highlights selected segments
- Shows playhead during preview
- Enables segment selection by clicking/dragging

Spotify API Integration:

- Fetches audio analysis data
- Uses segment and bar data to generate waveform

`SettingsToggle.js`

Purpose: Button to expand/collapse detailed settings
Responsibilities:

- Toggles visibility of track settings panel
- Shows current state (expanded/collapsed)

`TrackSettings.js`

Purpose: Container for all track editing controls
Responsibilities:

- Houses all audio manipulation controls
- Organizes controls into logical sections
- Only visible when track is expanded

Child Components:

- `<SegmentManager>`
- `<BasicControls>`
- `<AudioAdjustments>`
- `<CollapsibleEQ>`
- `<CollapsibleEffects>`

`SegmentManager.js`

Purpose: Manage multiple segments/sections of a track
Responsibilities:

- Displays list of all segments
- Allows adding new segments
- Coordinates segment editing and deletion

Child Components:

- `<SegmentList>`

- `<AddSegmentButton>`

Firebase/Firestore Integration:

- Updates segment array in track object when changed

`SegmentList.js`

Purpose: List container for all track segments
Responsibilities:

- Renders all segments in order
- Manages segment selection

Child Components:

- `<SegmentItem>` (one per segment)

`SegmentItem.js`

Purpose: Individual segment display and editor
Responsibilities:

- Shows segment bar range
- Allows editing start/end bars
- Provides delete option
- Displays segment duration

`AddSegmentButton.js`

Purpose: Add new segment to track
Responsibilities:

- Creates new segment with default range
- Opens waveform for segment selection

`BasicControls.js`

Purpose: Essential volume and fade controls
Responsibilities:

- Manages overall track volume
- Controls fade in duration
- Controls fade out duration

Child Components:

- `<VolumeSlider>`
- `<FadeInput>` (for fade in)
- `<FadeInput>` (for fade out)

`VolumeSlider.js`

Purpose: Visual slider for volume control
Responsibilities:

- Displays current volume level (0-100%)
- Allows dragging to adjust
- Shows percentage value

`FadeInput.js`

Purpose: Numeric input for fade durations
Responsibilities:

- Accepts bar count for fades
- Validates input values
- Labels fade type (in/out)

`AudioAdjustments.js`

Purpose: Pitch and speed manipulation controls
Responsibilities:

- Manages pitch shift settings
- Manages tempo/speed settings

Child Components:

- `<PitchControl>`
- `<SpeedControl>`

`PitchControl.js`

Purpose: Pitch shift adjustment control
Responsibilities:

- Allows pitch adjustment in semitones (-12 to +12)
- Displays current pitch shift value
- May show musical interval (octave, fifth, etc.)

`SpeedControl.js`

Purpose: Tempo/speed adjustment control
Responsibilities:

- Allows speed adjustment (0.5x to 2.0x)
- Displays current speed multiplier
- Has detents at common values (0.75x, 1.0x, 1.25x, etc.)

`CollapsibleEQ.js`

Purpose: Expandable equalizer section
Responsibilities:

- Toggle visibility of EQ controls
- Shows current EQ state when collapsed

Child Components:

- `<EQControls>` (when expanded)

`EQControls.js`

Purpose: Three-band equalizer controls
Responsibilities:

- Controls low frequency band
- Controls mid frequency band
- Controls high frequency band
- Provides reset to defaults option

Child Components:

- `<EQBand>` (three instances for low/mid/high)
- `<EQVisualizer>` (optional frequency curve display)

`EQBand.js`

Purpose: Individual frequency band control
Responsibilities:

- Adjusts single frequency band (-12dB to +12dB)
- Labels band (Low/Mid/High)
- Shows current value

`CollapsibleEffects.js`

Purpose: Expandable effects section
Responsibilities:

- Toggle visibility of effects panel
- Shows number of active effects when collapsed

Child Components:

- `<EffectsPanel>` (when expanded)

`EffectsPanel.js`

Purpose: Audio effects chain manager
Responsibilities:

- Manages list of active effects
- Allows adding new effects
- Coordinates effect ordering

Child Components:

- `<EffectsList>`
- `<AddEffectButton>`

`EffectsList.js`

Purpose: List of active audio effects
Responsibilities:

- Displays all effects in chain order
- Enables drag-to-reorder effects
- Manages effect removal

Child Components:

- `<EffectItem>` (one per effect)

`EffectItem.js`

Purpose: Individual effect configuration
Responsibilities:

- Displays effect type
- Shows effect-specific parameters
- Allows parameter adjustment
- Provides remove option

Child Components:

- `<EffectTypeSelector>`
- `<EffectParameters>` (dynamic based on effect type)

`AddEffectButton.js`

Purpose: Add new effect to chain
Responsibilities:

- Opens effect type selector
- Creates new effect with default parameters

`DragHandle.js`

Purpose: Visual indicator for drag-and-drop
Responsibilities:

- Provides grab handle between tracks
- Shows hover state
- Integrates with drag-and-drop system

`AddTrackButton.js`

Purpose: Add new track to playlist
Responsibilities:

- Shows current track count (X/5)
- Disabled when max tracks reached
- Opens track search modal

---

4. AI Sidebar Components

`AISidebar.js`

Purpose: Collapsible panel for AI recommendations
Responsibilities:

- Toggles between expanded and collapsed states
- Manages sidebar visibility

Child Components:

- `<CollapseButton>` (when expanded)
- `<ExpandButton>` (when collapsed)
- `<AIPanel>` (when expanded)

`AIPanel.js`

Purpose: Container for all AI suggestion features
Responsibilities:

- Analyzes current playlist
- Fetches AI recommendations
- Displays suggestions with context

Child Components:

- `<AIContext>`
- `<SuggestionList>`
- `<RefreshButton>`

Spotify API Integration:

- Calls recommendations endpoint with current track data
- Retrieves audio features for compatibility analysis

`AIContext.js`

Purpose: Contextual message about suggestions
Responsibilities:

- Displays helpful text explaining suggestions
- Adapts message based on playlist state

`SuggestionList.js`

Purpose: List of recommended tracks
Responsibilities:

- Displays 3-5 AI-suggested tracks
- Sorts by compatibility score

Child Components:

- `<SuggestionCard>` (one per suggestion)

`SuggestionCard.js`

Purpose: Individual track recommendation
Responsibilities:

- Displays track name and artist
- Shows BPM and key
- Displays compatibility score
- Explains why track was suggested
- Provides add-to-playlist action

Child Components:

- `<SuggestionInfo>`
- `<MatchScore>`
- `<AddButton>`

Spotify API Integration:

- Displays track data from Spotify

`MatchScore.js`

Purpose: Visual compatibility score indicator
Responsibilities:

- Displays match percentage (0-100%)
- Uses color coding (green/yellow/red)
- May use star rating or progress bar

`RefreshButton.js`

Purpose: Regenerate AI suggestions
Responsibilities:

- Triggers new recommendation fetch
- Shows loading state during refresh

---

5. Modal Components

`TrackSearchModal.js`

Purpose: Search and select tracks from Spotify
Responsibilities:

- Provides search interface
- Displays search results
- Handles track selection
- Closes on selection or cancel

Child Components:

- `<SearchInput>`
- `<SearchResults>`
- `<CloseButton>`

Spotify API Integration:

- Calls Spotify search endpoint
- Retrieves track metadata

Firebase/Firestore Integration:

- Adds selected track to playlist in Firestore

`SearchInput.js`

Purpose: Search input field with autocomplete
Responsibilities:

- Accepts user search query
- Triggers search on input (debounced)
- Displays search icon/button

`SearchResults.js`

Purpose: Display search results list
Responsibilities:

- Shows matching tracks
- Handles empty state
- Manages loading state

Child Components:

- `<SearchResultItem>` (one per result)

`SearchResultItem.js`

Purpose: Individual search result
Responsibilities:

- Displays album artwork
- Shows track name and artist
- Displays duration
- Shows BPM and key if available
- Handles selection on click

Spotify API Integration:

- May require additional call for audio features

`LoadPlaylistModal.js`

Purpose: Browse and load saved playlists
Responsibilities:

- Fetches user's saved playlists
- Displays playlist list
- Handles playlist selection
- Closes on load or cancel

Child Components:

- `<PlaylistList>`
- `<CloseButton>`

Firebase/Firestore Integration:

- Fetches all playlists from user's Firestore collection
- Loads selected playlist data

`PlaylistList.js`

Purpose: List of saved playlists
Responsibilities:

- Displays all user playlists
- Shows playlist metadata (name, track count, date)

Child Components:

- `<PlaylistListItem>` (one per playlist)

`PlaylistListItem.js`

Purpose: Individual playlist in list
Responsibilities:

- Displays playlist name
- Shows track count
- Shows last modified date
- Provides load action
- Provides delete action (optional)

Firebase/Firestore Integration:

- Delete removes playlist from Firestore

`SpotifyAuthModal.js`

Purpose: Handle Spotify OAuth login
Responsibilities:

- Displays login instructions
- Initiates OAuth flow
- Handles callback
- Shows error states

Spotify API Integration:

- Manages OAuth 2.0 authorization flow
- Exchanges authorization code for access token

Firebase/Firestore Integration:

- Stores access and refresh tokens in Firestore

---

6. Service/Utility Layers

`SpotifyService`

Purpose: Centralized Spotify API integration
Responsibilities:

- Manages all Spotify API calls
- Handles authentication
- Manages token refresh
- Provides helper methods for common operations

Key Methods:

- `searchTracks(query)` - Search for tracks
- `getTrack(trackId)` - Get track details
- `getAudioFeatures(trackId)` - Get BPM, key, energy, etc.
- `getAudioAnalysis(trackId)` - Get detailed waveform data
- `getRecommendations(seedTracks, parameters)` - Get AI suggestions
- `refreshAccessToken()` - Refresh expired token

Spotify API Endpoints Used:

- `/v1/search` - Track search
- `/v1/tracks/{id}` - Track metadata
- `/v1/audio-features/{id}` - Audio features (BPM, key)
- `/v1/audio-analysis/{id}` - Detailed audio analysis
- `/v1/recommendations` - Track recommendations

`FirebaseService`

Purpose: Centralized Firebase/Firestore operations
Responsibilities:

- Manages all database operations
- Handles authentication integration
- Provides CRUD operations for playlists
- Manages token storage

Key Methods:

- `savePlaylist(userId, playlistId, data)` - Save playlist

- `loadPlaylist(userId, playlistId)` - Load playlist
- `getUserPlaylists(userId)` - Get all user playlists
- `deletePlaylist(userId, playlistId)` - Delete playlist
- `saveSpotifyToken(userId, token)` - Store Spotify token
- `getSpotifyToken(userId)` - Retrieve Spotify token

Firestore Collections:

- `/users/{userId}/playlists/{playlistId}` - Playlist documents
- `/users/{userId}/tokens/spotify` - Token storage

`AudioEngine`

Purpose: Web Audio API wrapper for audio manipulation
Responsibilities:

- Loads and decodes audio
- Applies audio effects in real-time
- Manages audio playback
- Coordinates effect chain

Key Methods:

- `loadTrack(audioUrl)` - Load and decode audio
- `applyPitchShift(semitones)` - Shift pitch
- `applySpeedChange(multiplier)` - Adjust tempo
- `applyEQ(low, mid, high)` - Apply equalizer
- `applyEffect(type, parameters)` - Apply audio effect
- `playSegment(startTime, endTime)` - Play track segment
- `applyFade(type, duration)` - Apply fade in/out

`AIRecommendationEngine`

Purpose: AI logic for track compatibility and recommendations
Responsibilities:

- Analyzes track compatibility
- Scores matches based on multiple factors
- Generates explanations for recommendations
- Coordinates with Spotify API for suggestions

Key Methods:

- `analyzeCompatibility(track1, track2)` - Calculate match score
- `generateRecommendations(currentTracks)` - Get suggestions
- `explainRecommendation(track, context)` - Generate explanation
- `scoreByBPM(bpm1, bpm2)` - BPM compatibility scoring
- `scoreByKey(key1, key2)` - Harmonic compatibility scoring
- `scoreByGenre(genres1, genres2)` - Genre similarity scoring

Scoring Factors:

- BPM difference (40% weight)
- Key/harmonic compatibility (40% weight)
- Genre similarity (20% weight)

Firebase/Firestore Data Schema

User Collection Structure

```
/users

 /{userId}

   - email: string

   - displayName: string

   - createdAt: timestamp



   /playlists

     /{playlistId}

       - name: string

       - createdAt: timestamp

       - updatedAt: timestamp

       - tracks: array of track objects
```

```
  /tokens
    /spotify
      - accessToken: string
      - refreshToken: string
      - expiresAt: timestamp
```

Track Object Structure

```
{
  order: number (0-4),
  spotifyTrackId: string,
  trackName: string,
  artistName: string,
  bpm: number,
  key: string,
  segments: array [
    {
      id: string,
      startBar: number,
      endBar: number,
      settings: {
        volume: number (0-100),
        fadeIn: number (bars),
        fadeOut: number (bars),
        pitch: number (semitones),
        speed: number (multiplier),
        eq: {
          low: number (dB),
          mid: number (dB),
          high: number (dB)
        },
        effects: array [
          {
            type: string,
            parameters: object
          }
        ]
      }
    }
  ]
}
```

Spotify API Integration Summary

Authentication Flow

1. User opens app
2. Not authenticated → `SpotifyAuthModal` appears
3. User clicks login
4. Redirected to Spotify OAuth

5. User authorizes app
6. Callback returns with authorization code
7. Exchange code for access token and refresh token
8. Store tokens in Firestore via `FirebaseService`
9. Tokens available throughout app via `AuthProvider`

Track Search Flow

1. User clicks "Add Track"
2. `TrackSearchModal` opens
3. User enters search query
4. `SpotifyService.searchTracks()` called
5. Results displayed in `SearchResults`
6. User selects track
7. `SpotifyService.getAudioFeatures()` fetches BPM/key
8. `SpotifyService.getAudioAnalysis()` fetches waveform data
9. Track added to playlist state
10. `FirebaseService.savePlaylist()` persists to database

AI Recommendation Flow

1. User has tracks in playlist
2. `AIPanel` analyzes current tracks
3. `AIRecommendationEngine.generateRecommendations()` called
4. Engine calls `SpotifyService.getRecommendations()` with:
   - Seed tracks (last 1-2 tracks)
   - Target BPM (from last track)
   - Target key (from last track)
5. Spotify returns 10-20 recommendations
6. For each recommendation:
   - Fetch audio features
   - Calculate compatibility score
   - Generate explanation
7. Sort by score, take top 5
8. Display in `SuggestionList`

Audio Playback Flow

1. User clicks preview on track
2. `AudioEngine.loadTrack()` fetches audio
3. Apply all settings (pitch, speed, EQ, effects)
4. `AudioEngine.playSegment()` plays selected segment

5. Waveform shows playhead position
6. User can stop/restart preview

Component Data Flow Diagram

```
App.js (Root State)
│
├── Authentication Flow
│   └── AuthProvider
│       ├── Firebase Auth State
│       └── Spotify Token Management
│           └── Firestore Token Storage
│
├── Playlist Management
│   ├── Header
│   │   ├── Save → FirebaseService → Firestore
│   │   ├── Load → FirebaseService → Firestore
│   │   └── Preview → AudioEngine
│   │
│   ├── MainWorkspace
│   │   └── TrackStack
│   │       ├── Track Add → TrackSearchModal
│   │       │   └── SpotifyService.searchTracks()
│   │       │
│   │       └── TrackCard (x5)
│   │         ├── WaveformDisplay
│   │         │   └── SpotifyService.getAudioAnalysis()
│   │         │
│   │         ├── TrackSettings
│   │         │   ├── Segment Management → Firestore
│   │         │   ├── Audio Controls → AudioEngine
│   │         │   └── Effects → AudioEngine
│   │         │
│   │         └── Reorder → Firestore Update
│   │
│   └── AISidebar
│       └── AIPanel
│           ├── Current Tracks → AIRecommendationEngine
│           ├── AIRecommendationEngine → SpotifyService
│           ├── SpotifyService.getRecommendations()
│           └── Suggestions → SuggestionList
│               └── Add Track → Playlist State
│
└── Modals
    ├── TrackSearchModal → SpotifyService
    ├── LoadPlaylistModal → FirebaseService
    └── SpotifyAuthModal → Spotify OAuth
```

Key Design Decisions Explained

Why Vertical Stacking?

- Matches natural playlist reading flow (top to bottom)
- Familiar to users from every music app
- Easy to scan entire mix without horizontal scrolling
- Simple reordering with drag-and-drop

Why Collapsible Controls?

- Prevents UI overwhelming for beginners
- Progressive disclosure shows complexity only when needed
- Keeps frequently-used controls (volume, fades) always visible
- Advanced users can access deep editing without cluttering interface

Why Segment-Based Editing?

- Enables creative workflows (use intro + outro, skip middle verse)
- Mirrors how DJs actually work with tracks
- More flexible than simple trim
- Allows multiple sections of same song in one "track slot"

Why Limit to 5 Tracks?

- Manageable scope for semester project
- Prevents Web Audio API performance issues
- Still allows 15-20 minute mixes
- Keeps UI manageable on single screen without excessive scrolling

Why Collapsible AI Sidebar?

- AI assistance is optional, not mandatory
- Maximizes workspace when AI not needed
- Reduces cognitive load for users who prefer manual selection
- Easy to access when inspiration needed

Why Firebase for Storage?

- Real-time sync capabilities for future collaboration features
- Authentication already integrated
- NoSQL structure matches dynamic playlist data
- Scalable without complex backend setup

Why Separate Service Layers?

- Clean separation of concerns
- Easier to test individual components
- Can swap implementations (e.g., different audio engine)
- Centralized error handling
- Reusable across components