Design 1: Dual-Deck Linear Timeline

Architecture Overview

Component Structure:

- `App.js` - Main container with Firebase auth state management
- `AuthProvider.js` - Firebase authentication wrapper
- `DeckController.js` - Parent component managing two deck instances
- `Deck.js` - Individual track player (reusable component)
- `Timeline.js` - Horizontal waveform/scrubber visualization
- `AudioControls.js` - EQ, pitch, tempo, volume sliders
- `AIRecommendations.js` - Sidebar with suggested tracks
- `ProjectManager.js` - Save/load mashup configurations
- `SpotifyService.js` - API integration service

Layout:

```
+-----------------------------------------------+
| Header (User Profile, Save/Load)              |
+-----------------------------------------------+
| Deck A: [Track Info] [BPM] [Key]              |
| [═══════════════ Timeline ═══════════════]    |
| [EQ] [Pitch] [Tempo] [Volume]                 |
+-----------------------------------------------+
| Deck B: [Track Info] [BPM] [Key]              |
| [═══════════════ Timeline ═══════════════]    |
| [EQ] [Pitch] [Tempo] [Volume]                 |
+-----------------------------------------------+
| AI Recommendations Sidebar                    |
| - Track 1 (BPM: 128, Key: Am)                 |
| - Track 2 (BPM: 130, Key: Cm)                 |
+-----------------------------------------------+
```

Data Flow:

1. Firebase Auth → User login → Spotify OAuth flow
2. Spotify API → Track selection → Extract metadata (BPM, key, waveform)
3. Web Audio API → Real-time audio manipulation
4. Firebase Firestore → Store mashup configs (track IDs, timestamps, settings)
5. AI Recommendation Engine → Query Spotify API based on current tracks

Pros:

- Familiar DJ interface (traktor/serato-like)
- Clear visual separation between tracks
- Easy to implement independently testable components

Cons:

- Limited to two tracks simultaneously
- Takes more vertical space

Design 2: Stacked Layer Mixer

Architecture Overview

Component Structure:

- `App.js` - Main container with routing
- `AuthProvider.js` - Firebase authentication
- `MixerCanvas.js` - Parent component for all layers
- `TrackLayer.js` - Reusable layer component (unlimited instances)
- `LayerControls.js` - Collapsed/expandable controls per layer
- `MasterOutput.js` - Final mix controls
- `AIAssistant.js` - Floating panel with recommendations
- `LibraryBrowser.js` - Spotify track search/selection
- `SessionManager.js` - Firebase CRUD for projects

Layout:

```
+-----------------------------------------------+
| Library Browser | AI Assistant (Floating Panel)  |
+-----------------------------------------------+
| Layer 1: Track A [▶] [BPM:128] [Key:Am] [⚙]    |
|   └─ [Waveform═══════════════════════════]       |
|   └─ [EQ Controls] (Collapsed by default)       |
+-----------------------------------------------+
| Layer 2: Track B [▶] [BPM:130] [Key:Cm] [⚙]    |
|   └─ [Waveform═══════════════════════════]       |
+-----------------------------------------------+
| [+ Add Layer]                                 |
+-----------------------------------------------+
```

| Master Output: [Volume] [Save] [Export] |

+---+

Data Flow:

1. User logs in → Firebase Auth → Spotify OAuth
2. LibraryBrowser queries Spotify API → Returns tracks with metadata
3. TrackLayer instances manage individual Web Audio nodes
4. AIAssistant analyzes active layers → Queries Spotify for compatible tracks
5. SessionManager saves layer stack to Firestore (array of track configs)

Pros:

- Scalable to multiple tracks (not just two)
- DAW-like interface familiar to producers
- Collapsed controls save screen space

Cons:

- More complex state management (track order, layer visibility)
- Harder to visualize alignment between multiple tracks

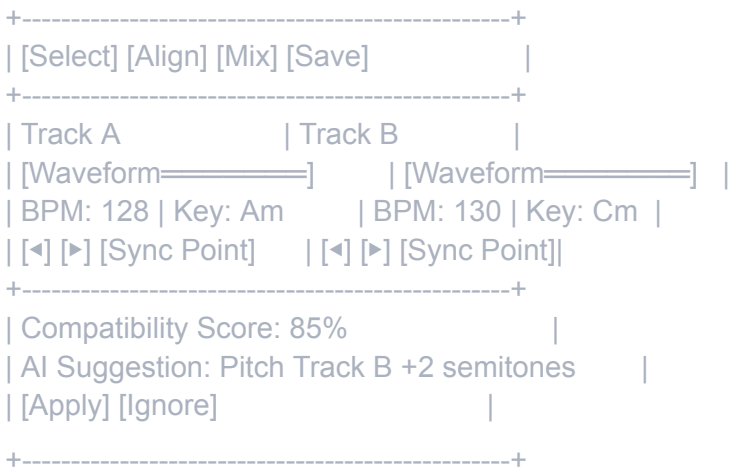
Design 3: Split-Screen Comparison + Workflow Tabs

Architecture Overview

Component Structure:

- `App.js` - Main app with tab navigation
- `AuthProvider.js` - Firebase auth wrapper
- `WorkflowTabs.js` - Tab switcher (Select → Align → Mix → Save)
- `SelectTab.js` - Track browsing + AI recommendations
- `AlignTab.js` - Side-by-side waveform comparison
- `MixTab.js` - Audio controls and real-time preview
- `SaveTab.js` - Project management and export
- `TrackComparison.js` - Dual waveform viewer
- `CompatibilityAnalyzer.js` - BPM/key matching visualization
- `SpotifyIntegration.js` - Centralized API service
- `FirebaseService.js` - Database operations

Layout (Align Tab Example):



Data Flow:

1. Firebase Auth → Spotify OAuth → Access token stored in context
2. **Select Tab**: Spotify API search → AIRecommendations analyzes → Display
3. **Align Tab**: Fetch audio features → Web Audio API for waveforms → User adjusts sync points
4. **Mix Tab**: Apply EQ/pitch/tempo → Real-time Web Audio processing
5. **Save Tab**: Serialize all settings → Firestore document with user ID

Pros:

- Guided workflow reduces cognitive load
- Each tab focuses on one task (separation of concerns)
- Easy to add/remove features per tab without affecting others

Cons:

- Context switching between tabs may disrupt flow
- Can't see all controls simultaneously

Design Proposals for AI-Enhanced Music Mashup Studio

Design 1: Playlist Timeline with Transition Zones

Concept

A horizontally scrolling timeline where tracks appear as blocks placed side-by-side. Between each pair of tracks is a visible "transition zone" that shows how many bars will be used for the crossfade/blend. Think of it like a video editing timeline, but for music.

Visual Layout

The top of the screen shows your playlist name, save/load buttons, and AI suggestions for the next track to add. The main area is a horizontal timeline that you can scroll through, with each track appearing as a rectangular block. Between tracks, you see a wavy/gradient transition zone labeled with the bar count (e.g., "8 bars"). Below the timeline is a detailed inspector panel that shows the waveform and all controls (EQ, pitch, tempo, volume) for whichever track you've selected.

How Users Interact

- Drag tracks from your library onto the timeline
- Click between two tracks to open a transition editor where you specify how many bars to use, which bar of the first song to start from, and which bar of the second song to blend into
- The transition zone visually shows the overlap between songs
- AI suggests the next track based on what's already in your playlist
- Click any track to see its detailed controls in the inspector below

Key Features

- Easy to see the entire playlist flow at once (with scrolling)
- Transition zones make it obvious where blends happen
- Can precisely control which bars of each song are used in transitions
- AI analyzes the last few tracks to suggest compatible next additions

Strengths

- Intuitive visual representation of song order and transitions

- Familiar to anyone who's used video editing software
- Clear separation between tracks and transitions
- Scales well to playlists of 10-20+ songs

Limitations

- Requires horizontal scrolling for longer playlists
- Editing a track in the middle might require adjusting surrounding transitions
- Timeline can get crowded with many tracks

Design 2: Vertical Playlist Builder with Expandable Transitions

Concept

A card-based vertical list where each track is a card, and between each pair of cards is a collapsible transition card. The screen is split into three panels: the playlist on the left, detailed track editing in the middle, and AI recommendations on the right.

Visual Layout

The left panel shows your playlist as a vertical stack of cards. Each track card displays the song name, artist, BPM, and key, with Edit/Remove buttons. Between each track card is a smaller transition card showing "8 bars | Crossfade" with an "Edit Transition" button that expands the card to show detailed controls. The middle panel shows detailed controls for whichever track you've selected (waveform, EQ, pitch, tempo). The right panel is the AI Queue showing 3-5 suggested tracks with compatibility scores.

How Users Interact

- Add tracks to the bottom of the playlist (they stack vertically)
- Click a track card to see its detailed controls in the middle panel
- Click "Edit Transition" on any transition card to expand it and adjust bar count, fade type, and bar alignment
- AI continuously suggests next tracks based on the last song in your playlist
- Drag to reorder tracks in the list
- All three panels are visible simultaneously

Key Features

- Natural top-to-bottom reading flow

- Transition settings are tucked away but easily accessible
- Three-panel layout means you never lose context
- Clear visual separation between tracks (cards) and transitions (smaller cards)
- AI queue is always visible on the side

Strengths

- Natural vertical scrolling for playlists of any length
- All tools accessible without switching views
- Card-based design feels modern and organized
- Easy to understand the sequence of your playlist
- Transitions don't clutter the view until you need to edit them

Limitations

- Takes up more horizontal screen space with three panels
- Long playlists require vertical scrolling
- Might feel cramped on smaller screens

Design 3: Grid-Based Playlist Workshop

Concept

A node-based visual interface where tracks appear as draggable boxes (nodes) connected by lines representing transitions. It's like a flowchart for your playlist. The left sidebar contains your track library, and the main canvas is where you build your playlist by dragging and connecting tracks.

Visual Layout

The left sidebar shows your Spotify library at the top (searchable) and AI-suggested tracks below. The main canvas area shows your playlist as connected nodes - each track is a box showing the song name, BPM, and key. Lines connect the boxes, with labels showing the bar count for each transition (e.g., "8 bars" on the connecting line). At the bottom is an "AI Coach" panel that analyzes your entire playlist and gives real-time suggestions like "Track 2 → Track 3 has a large BPM jump. Consider a 16-bar transition instead of 8."

How Users Interact

- Drag tracks from the library sidebar onto the canvas
- Tracks automatically connect in the order you place them, with default 8-bar transitions
- Click any track node to open an editor popup showing all controls

- Click any connection line to edit the transition (bar count, fade type, alignment points)
- AI Coach watches your entire playlist and suggests improvements
- Move nodes around the canvas to organize visually (though playback order stays linear)

Key Features

- Visual, spatial representation of your playlist
- See the big picture of your entire mix at once
- Nodes and connections make relationships between tracks explicit
- AI Coach provides contextual feedback on the entire playlist structure
- Drag-and-drop feels intuitive and playful

Strengths

- Highly visual and engaging interface
- Easy to see your entire playlist structure without scrolling
- Flexible canvas lets you organize however makes sense to you
- AI feedback is contextual to your whole playlist, not just one track
- Feels creative and experimental

Limitations

- Node-based interfaces might be unfamiliar to casual users
- Could be harder to maintain strict linear order if the canvas gets messy
- Requires more complex rendering (might need a library like React Flow)
- Less conventional - users need to learn the interaction model

Design Notes Before Finalizing - EH

1) Design 1
    - Pros
        - Easiest for users who have no experience (not complex)
        - Simple state model to track decks (easy testing)
    - Cons
        - Limited to two tracks, although typical lacks user freedom (plus, adding tracks if needed requires structure/layout redesign)
        - If users want to explore alternative tracks in existing projects, they have to manually delete existing tracks
2) Design 2
    - Pros
        - Familiar design, typically used in audio manipulation software like Audacity or video editing software like CapCut
        - Model is expandable, can start with 2 tracks initially
        - Track features and editing options are collapsable, not overwhelming UI for new users (allows for exploration)
        - Can add additional features not relevant to document such as the ability to zoom in/out on track layer, add navigation bar or collapsable panels for actions, etc.
    - Cons
        - Each new track added can affect audio complexity and app rendering
        - Dragging/reordering tracks has to be carefully managed
3) Design 3
    - Pros
        - Most creative autonomy out of all designs, users can form any sort of structure along with parameters (picturing Unreal Engine 5's Blueprint programming)
        - AI notes optional suggestions based on track features and user ideas in real-time (feels collaborative/helpful)
        - Users organize tracks however desired while system follows clear playback order behind the scenes (can add visual cues of this)
    - Cons
        - Spatial positioning doesn't reflect playback behavior; can be hard to visualize length out output without timemarkers, which are difficult to know where to place with this design
        - Some users might confuse themselves with overcomplexity or need to feel a need of mapping out design before starting
        - Track state is more complex to manage, higher bug chance due to edge cases, and more rendering capabilities needed to run

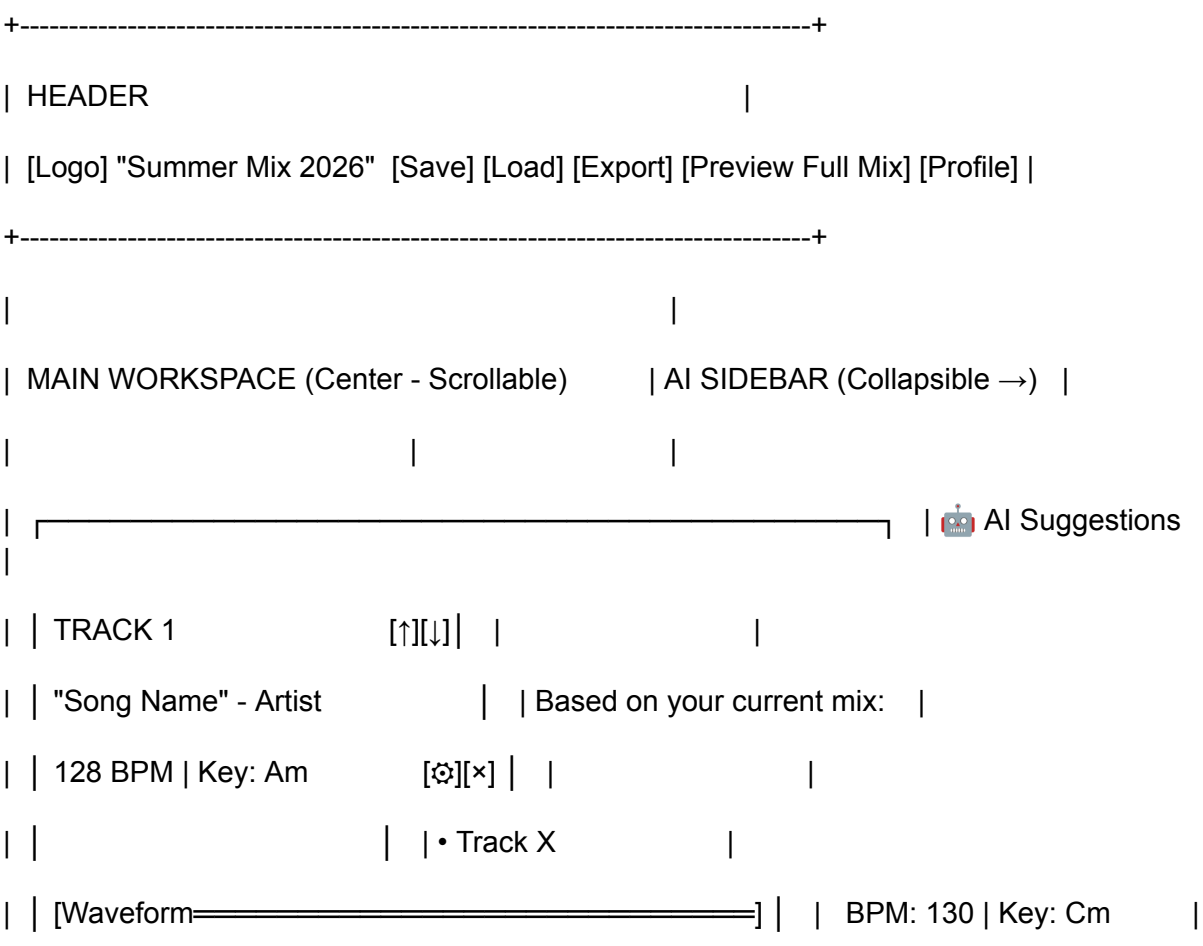
AI-Enhanced Music Mashup Studio - Final Design

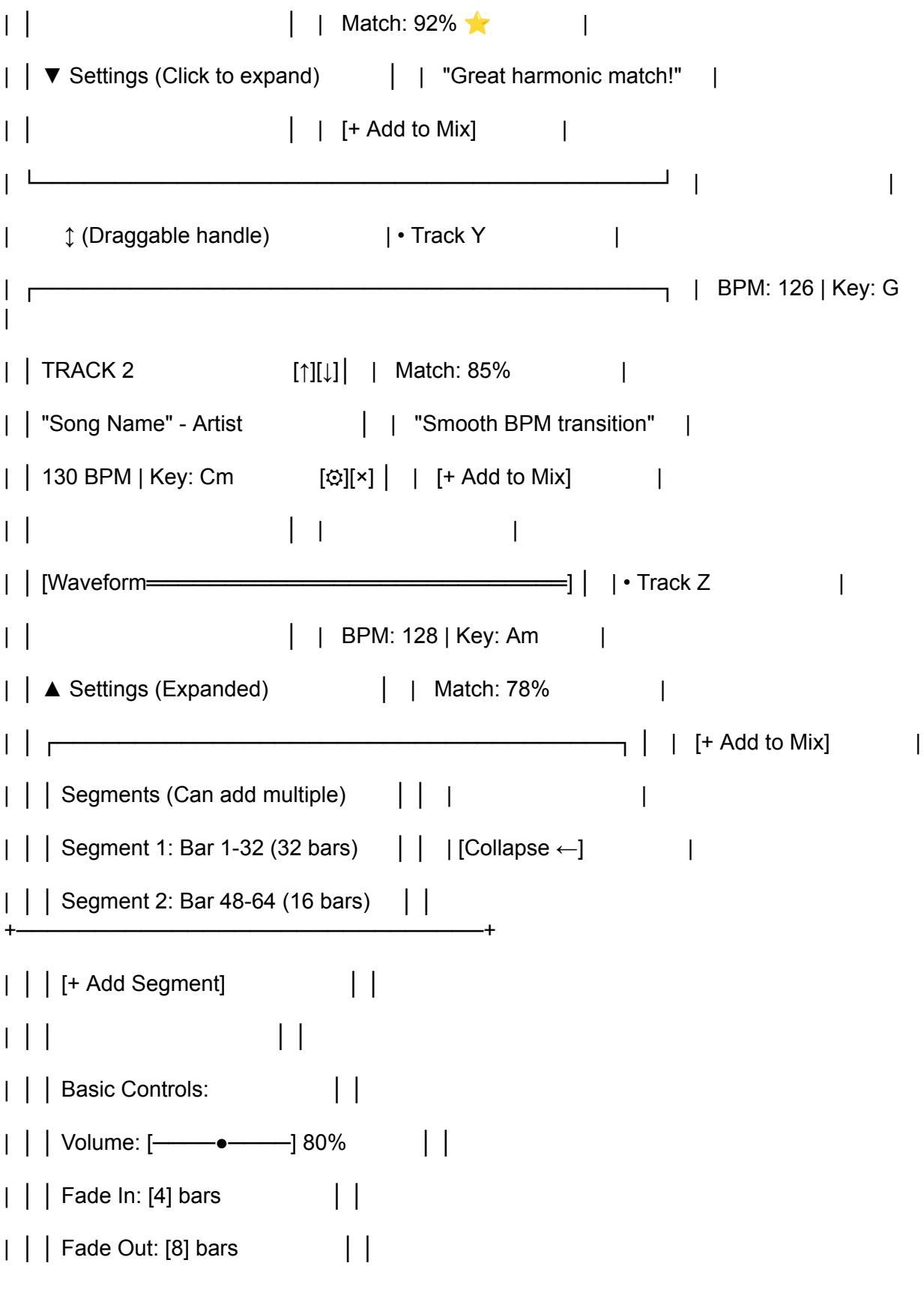
Vertical Track Stack with Collapsible Controls

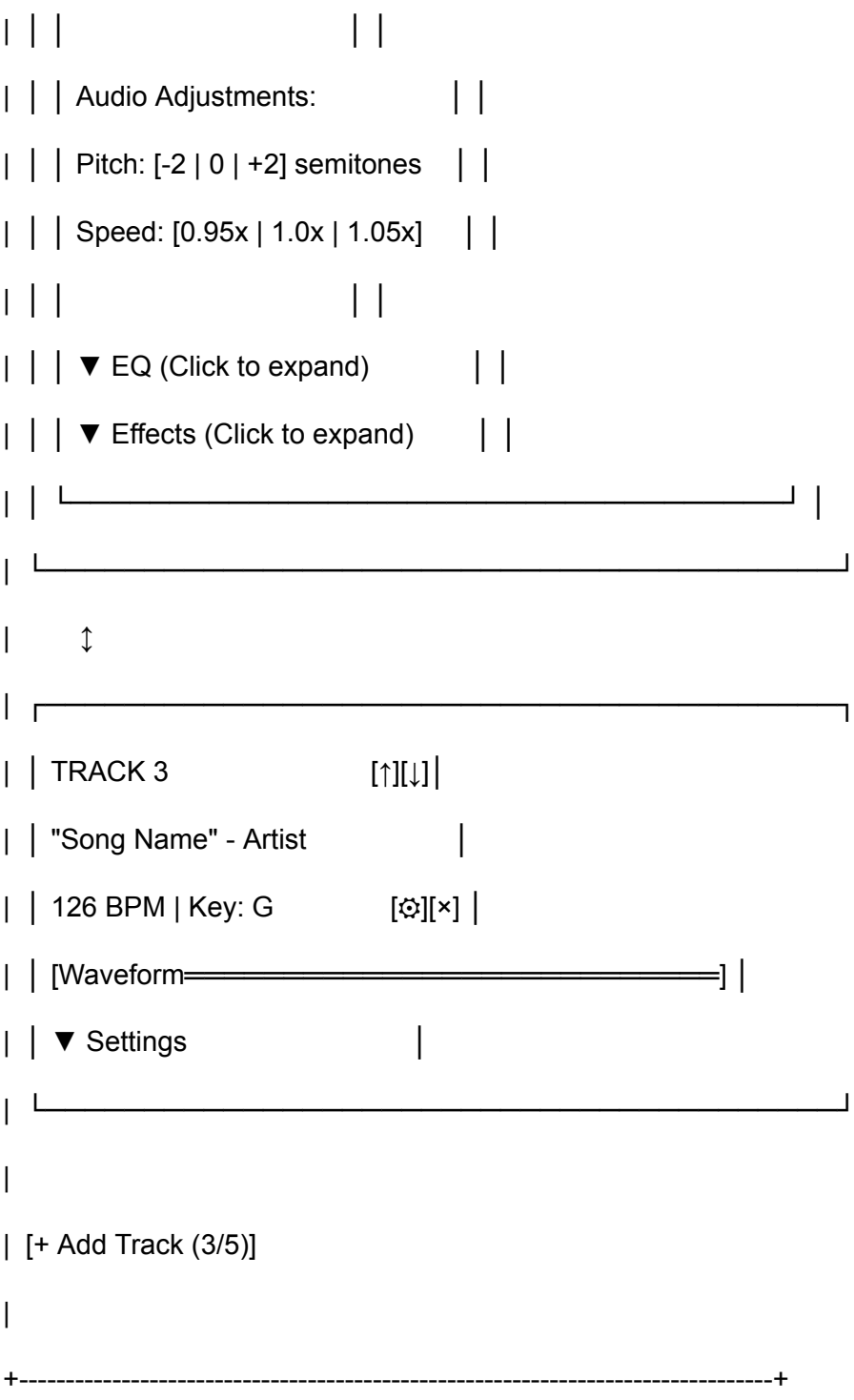
Overall Layout Philosophy

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




Match: 92% ⭐
▼ Settings (Click to expand)
"Great harmonic match!"
[+ Add to Mix]
↕ (Draggable handle)
• Track Y
BPM: 126 | Key: G
TRACK 2
[↑][↓]
Match: 85%
"Song Name" - Artist
"Smooth BPM transition"
130 BPM | Key: Cm
[⚙][×]
[+ Add to Mix]
[Waveform
• Track Z
BPM: 128 | Key: Am
▲ Settings (Expanded)
Match: 78%
[+ Add to Mix]
Segments (Can add multiple)
Segment 1: Bar 1-32 (32 bars)
[Collapse ←]
Segment 2: Bar 48-64 (16 bars)
[+ Add Segment]
Basic Controls:
Volume: [———●———] 80%
Fade In: [4] bars
Fade Out: [8] bars



Detailed Component Breakdown

Header Bar

- **Left side**: Logo/app name and current playlist name (editable)
- **Center**: Primary actions - Save, Load, Export
- **Right side**: Preview Full Mix button (plays entire playlist with all transitions), User profile

Main Workspace (Central Column)

Track Card (Collapsed State)

- **Track header**:
    - Song title and artist
    - BPM and Key displayed prominently
    - Up/Down arrows for reordering (or drag handle between cards)
    - Settings gear icon (expands settings panel)
    - X button to remove track
- **Waveform display**:
    - Full waveform visualization showing the entire track
    - Visual indicators for selected segments (highlighted regions)
    - Playhead indicator when previewing
- **Settings toggle**: "▼ Settings" button to expand/collapse detailed controls

Track Card (Expanded State)

When settings are expanded, the card grows vertically to show:

1. Segments Section (Top of settings)

- List of all segments/sections of the track being used
- Each segment shows: "Segment 1: Bar 1-32 (32 bars)"
- Can add multiple segments from the same track (trim/split functionality)
- Each segment can be independently adjusted
- "+ Add Segment" button to create new segments from the same track

2. Basic Controls

- **Volume slider**: Visual slider (0-100%) with current value
- **Fade In**: Number input for bar count (e.g., "4 bars")
- **Fade Out**: Number input for bar count (e.g., "8 bars")

3. Audio Adjustments

- **Pitch**: Slider or buttons for -12 to +12 semitones
- **Speed/Tempo**: Slider for 0.5x to 2.0x with detents at common values

4. Collapsible Advanced Controls

▼ **EQ (Collapsed by default)** When expanded shows:

- Low: Slider with dB values
- Mid: Slider with dB values
- High: Slider with dB values
- Visual frequency curve (optional)
- Reset to defaults button

▼ **Effects (Collapsed by default)** When expanded shows:

- Effect type dropdown (Reverb, Delay, Filter, Distortion, etc.)
- Effect-specific parameters (dry/wet mix, intensity, etc.)
- Can add multiple effects
- Effect chain order (draggable)
- "+ Add Effect" button

Track Reordering

- Drag handle (≡) between track cards allows dragging tracks up/down
- Alternatively, arrow buttons [↑][↓] in track header move tracks
- Visual feedback during drag (card follows cursor, other cards shift)

Add Track Button

- Located at bottom of track stack
- Shows current count: "[+ Add Track (3/5)]"
- Disabled when 5 tracks are added
- Opens track search/library modal

AI Suggestions Sidebar (Right Panel)

Collapsed State:

- Thin vertical bar with "🤖 AI" and expand arrow
- Saves screen space when not needed

Expanded State:

- **Header**: "🤖 AI Suggestions" with collapse button
- **Context**: "Based on your current mix:" or "Great next track for Track 3:"
- **Suggestion cards** (3-5 tracks):
    - Track name and artist
    - BPM and Key

    - Compatibility match percentage with visual indicator (stars/color)
    - Brief explanation ("Great harmonic match!", "Smooth BPM transition")
    - "+ Add to Mix" button
- **Refresh suggestions** button at bottom
- AI explains *why* each track was suggested (showing BPM difference, key compatibility, genre match)

Interaction Flows

Adding a Track

1. Click "[+ Add Track]" button
2. Search/browse Spotify library modal appears
3. Select track → Track card added to bottom of stack
4. AI sidebar immediately updates with new suggestions based on added track

Editing a Track

1. Click "▼ Settings" or gear icon on track card
2. Settings panel expands inline (card grows vertically)
3. Adjust basic controls (volume, fades, pitch, speed)
4. Click "▼ EQ" to expand equalizer controls
5. Click "▼ Effects" to add/configure effects
6. Changes apply in real-time (or with "Preview" button)

Creating Multiple Segments from One Track

1. In expanded settings, click "[+ Add Segment]"
2. Waveform becomes interactive - click and drag to select region
3. New segment appears in list: "Segment 2: Bar 16-24 (8 bars)"
4. Each segment can have independent settings (volume, fades, etc.)
5. Segments play in order they're listed (can reorder)

Reordering Tracks

1. **Method A**: Click and hold drag handle (≡) between cards, drag up/down
2. **Method B**: Click [↑] or [↓] arrows in track header
3. Other tracks shift to make space
4. AI suggestions update based on new order

Using AI Suggestions

1. Review suggested tracks in sidebar
2. Read compatibility score and explanation
3. Click "[+ Add to Mix]" on desired suggestion
4. Track automatically added to bottom of playlist
5. New suggestions generated based on updated playlist

Previewing

- **Individual track**: Play button on track card (plays with all current settings)
- **Transition preview**: Hover between tracks, click "Preview Transition" to hear 8 bars before/after
- **Full mix**: "Preview Full Mix" in header plays entire playlist start to finish

---

Data Structure Considerations

Firebase Schema (Simplified for this design)

Playlist Document:

- userId

- playlistId

- name: "Summer Mix 2026"

- tracks: [

```
  {
    order: 0,
    spotifyTrackId: "...",
    segments: [
     {
       startBar: 1,
       endBar: 32,
       settings: {
         volume: 0.8,
```

```
        fadeIn: 4,
        fadeOut: 8,
        pitch: 0,
        speed: 1.0,
        eq: { low: 0, mid: 0, high: 0 },
        effects: []
      }
    }
  ]
 },
 // ... up to 4 more tracks
]
```

Key Design Decisions

Why vertical stacking?

- Natural reading flow (top to bottom = start to end of playlist)
- Familiar pattern (like every music app playlist view)
- Easy to scan entire mix at a glance

Why collapsible controls?

- Prevents overwhelming users with too many options at once
- Keeps commonly-used controls (volume, fades) always visible
- Advanced users can dive deep into EQ/Effects when needed

Why segments instead of single trim?

- Allows creative mixing (use intro + outro of same song, skip middle)
- DJs often use multiple parts of one track in a set
- More flexibility without UI complexity

Why limit to 5 tracks?

- Semester project scope management
- Prevents performance issues with Web Audio API
- Still enough to create meaningful 15-20 minute mixes
- Keeps UI manageable on single screen

Why collapsible AI sidebar?

- Not everyone wants AI suggestions all the time
- Maximizes workspace for track editing
- AI remains easily accessible but not intrusive
- Users can focus on creativity or get help as needed

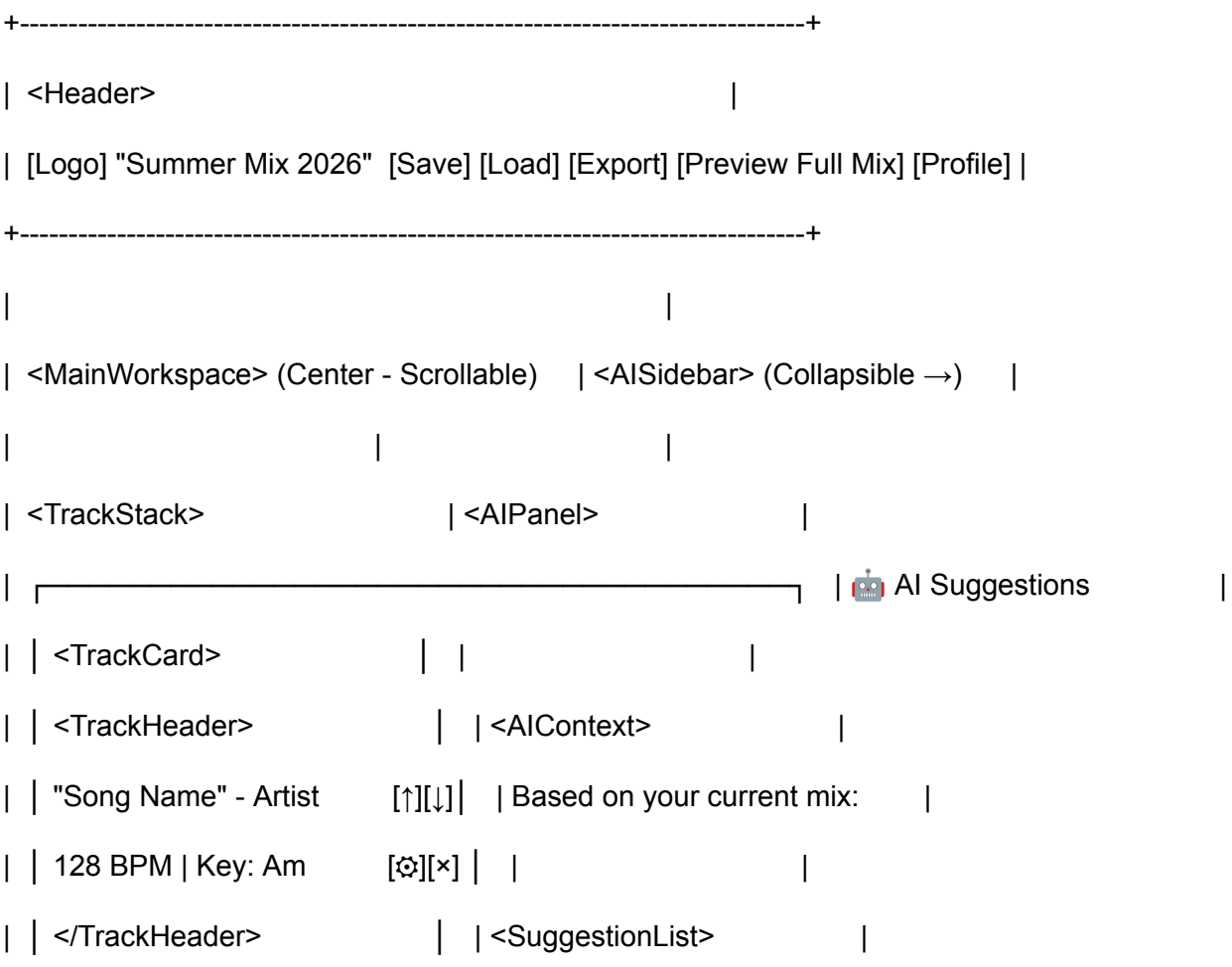
AI-Enhanced Music Mashup Studio - Complete Design Document

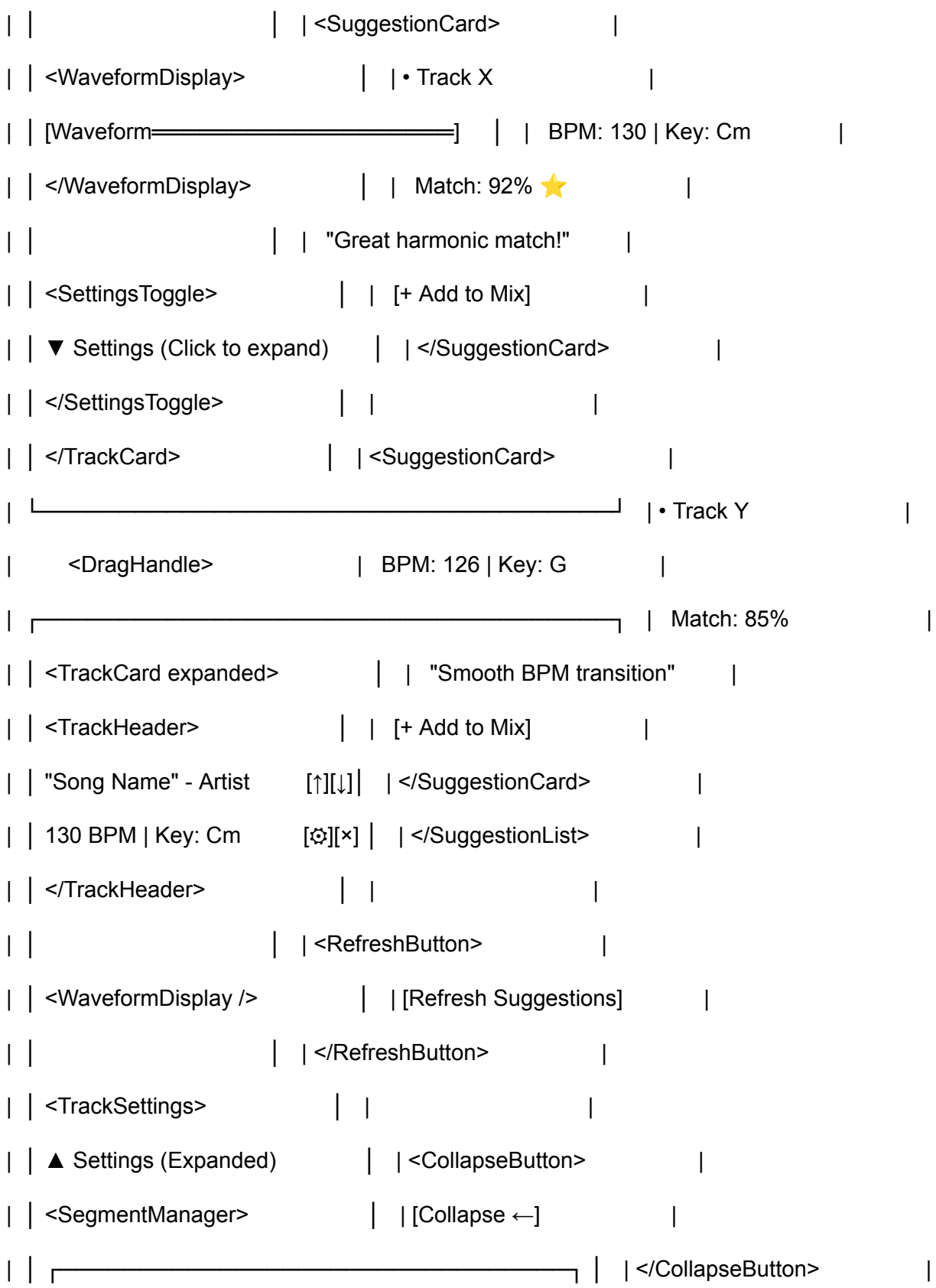
Visual Design & Component Mapping

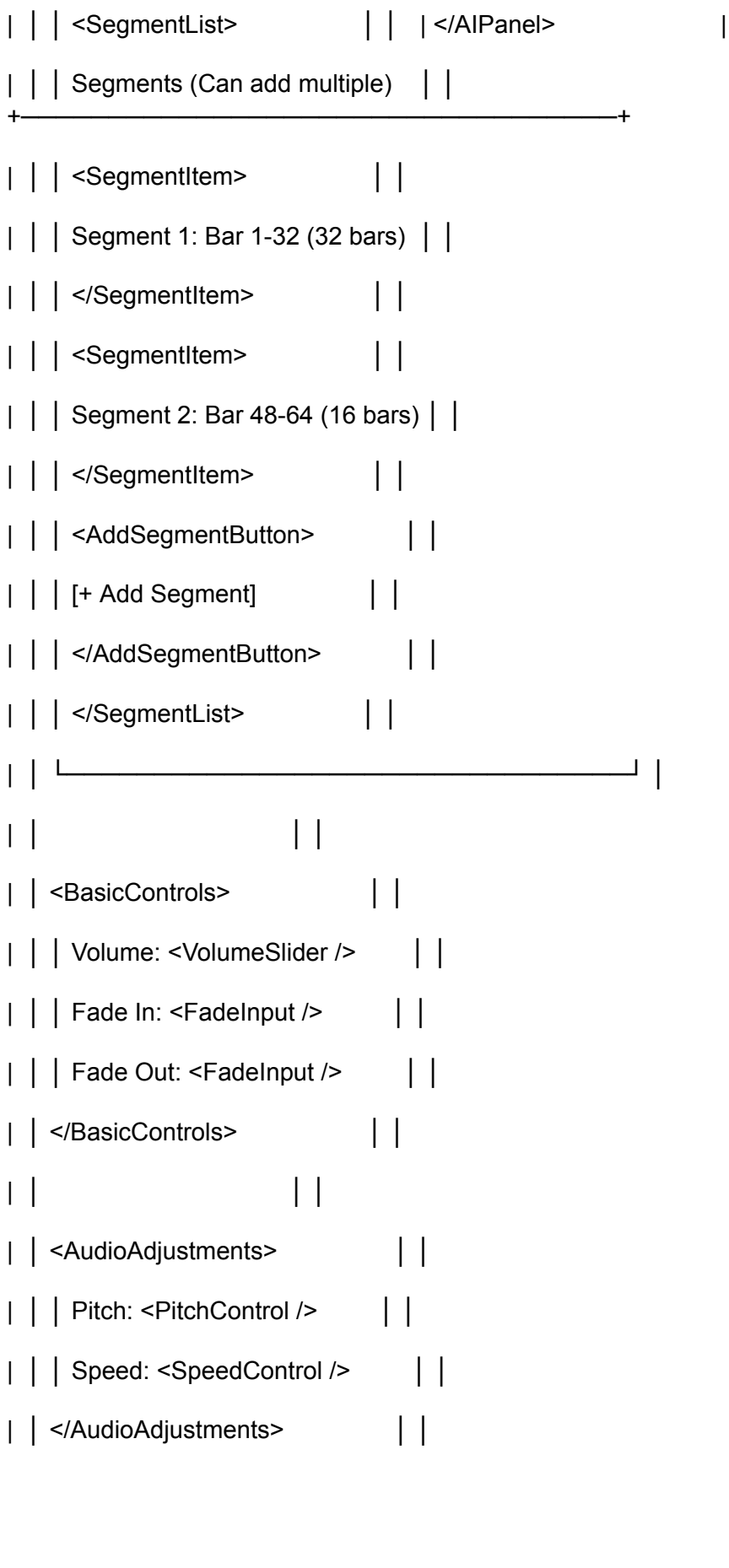
Overall Layout Philosophy

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




		<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

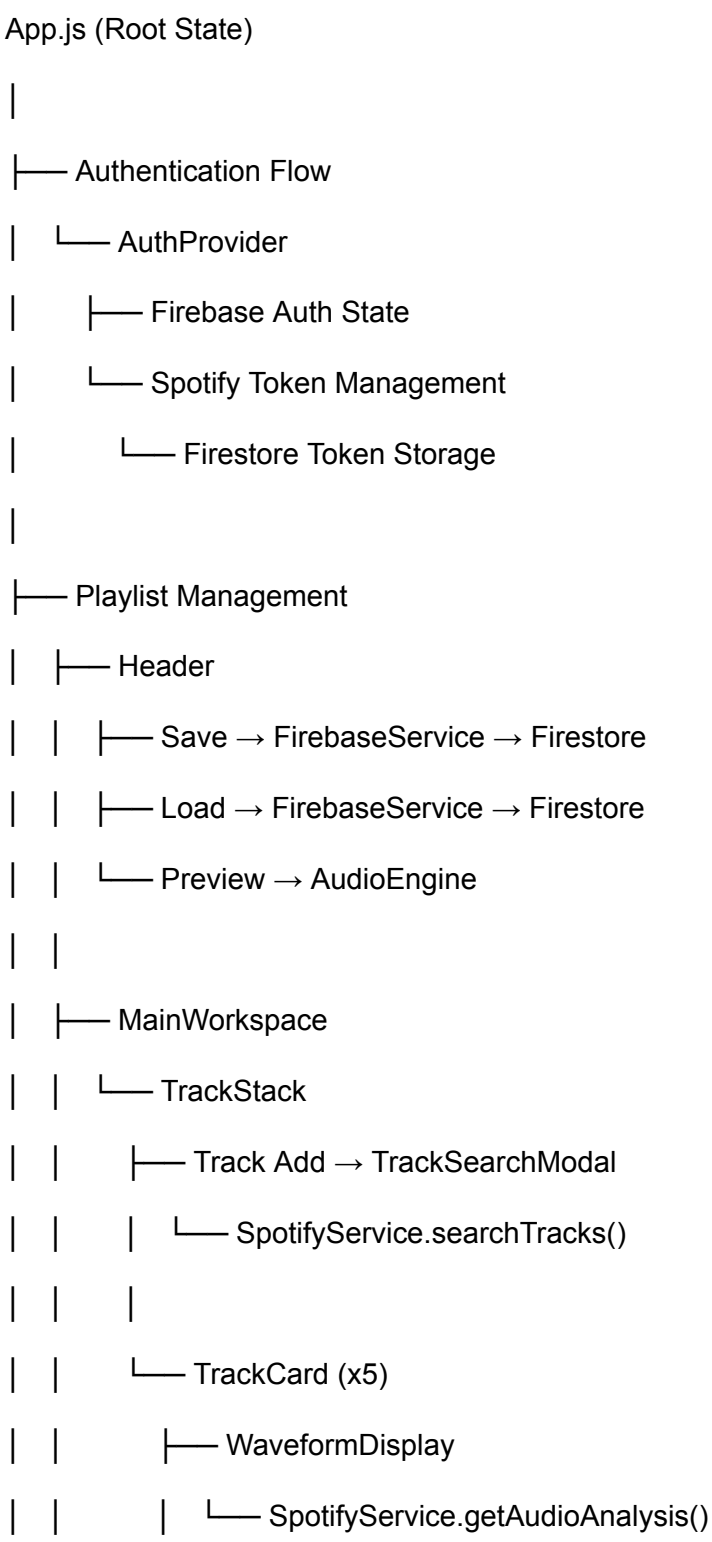
AI Recommendation Flow

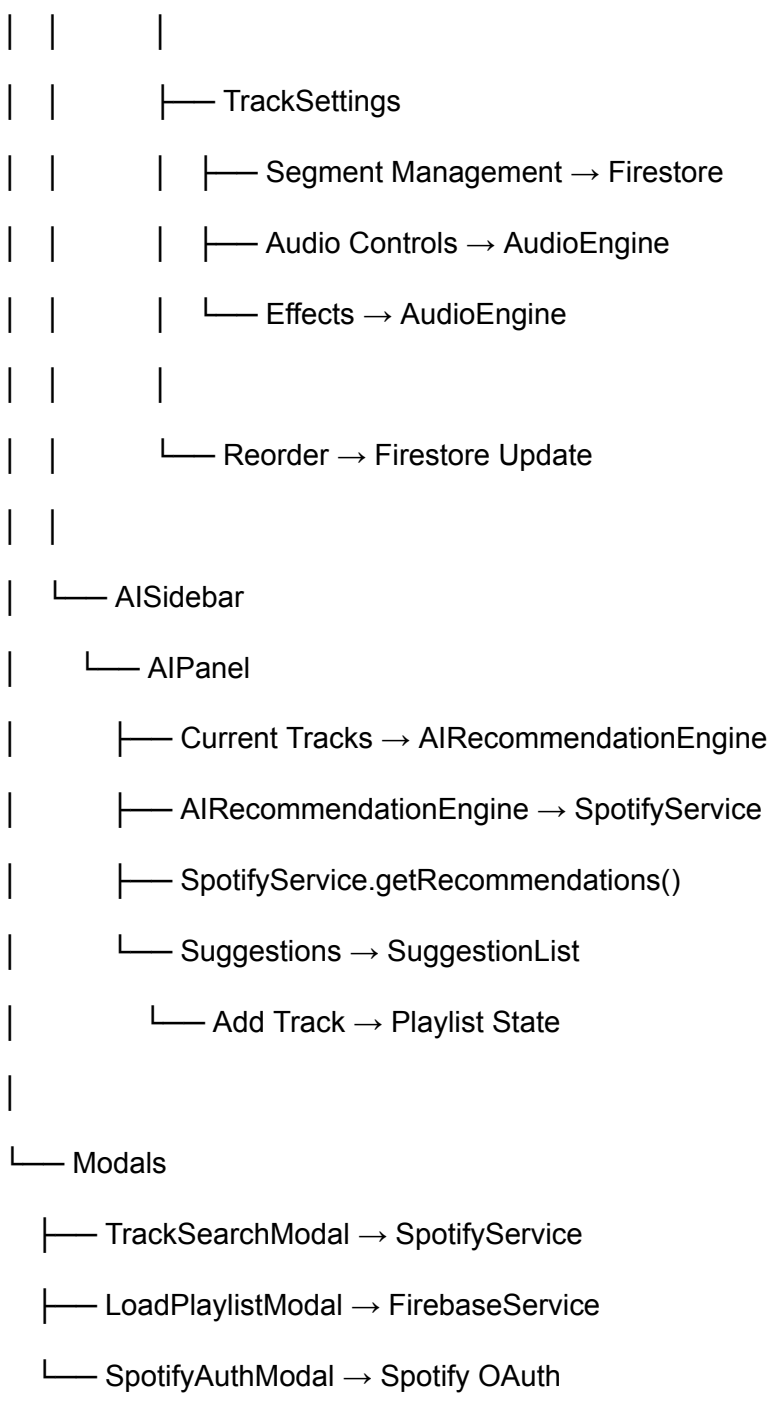
1. User has tracks in playlist
2. `AIPanel` analyzes current tracks
3. `AIRecommendationEngine.generateRecommendations()` called
4. Engine calls `SpotifyService.getRecommendations()` with:
    - Seed tracks (last 1-2 tracks)
    - Target BPM (from last track)
    - Target key (from last track)
5. Spotify 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





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