songIQ Web App - Cursor Agent Prompts

Prompt 1: Project Setup & Architecture

> Create a modern web application called "songIQ" - a music intelligence platform for artists. Set up the project with:
>
> TECH STACK:
> - Frontend: React with TypeScript, Tailwind CSS for styling
> - Backend: Node.js with Express
> - Database: MongoDB with Mongoose
> - File handling: Multer for file uploads
> - Audio processing: Web Audio API + tone.js for client-side analysis
> - Machine Learning: TensorFlow.js for predictions
>
> PROJECT STRUCTURE:

```
songiq/
├── client/
│   ├── src/
│   │   ├── components/
│   │   ├── pages/
│   │   ├── hooks/
│   │   ├── utils/
│   │   └── types/
├── server/
│   ├── src/
│   │   ├── routes/
│   │   ├── models/
│   │   ├── services/
│   │   └── utils/
└── shared/
└── types/
```

> Initialize the project with proper package.json files, TypeScript configs, and basic folder structure. Include scripts for development and build processes.

Prompt 2: Database Models & Types

Create MongoDB models and TypeScript interfaces for the songIQ application:

MODELS NEEDED:
1. **Song Model**
   - id, title, artist, uploadDate
   - audioFile (file path/URL)
   - isReleased (boolean)
   - analysisResults (embedded document)
   - performanceMetrics (for released songs)

2. **Analysis Results Model**
   - songId (reference)
   - audioFeatures: { genre, subGenre, tempo, key, mood, energy, duration }
   - vocalCharacteristics: { style, range, intensity }
   - instrumentation: string[]
   - lyricalThemes: string[] (if provided)
   - successPrediction: { score, ranking, insights }

3. **Performance Metrics Model**
   - songId (reference)
   - streamingData: { spotify, apple, youtube }
   - chartPositions: { billboard, genre-specific }
   - socialMetrics: { mentions, engagement }
   - demographics: { ageGroups, locations }

Create proper Mongoose schemas with validation, indexes, and TypeScript interfaces that match these models.

Prompt 3: Audio Upload & Processing Component

Create a React component for audio file upload and initial processing:

REQUIREMENTS:
1. **File Upload Interface**
   - Drag & drop zone for audio files
   - Support formats: MP3, WAV, FLAC, M4A
   - File size limit: 50MB
   - Progress indicator during upload
   - File validation and error handling

2. **Song Status Declaration**
   - Radio buttons: "Already Released" vs "Not Yet Released"
   - Conditional form fields based on selection
   - For released songs: optional fields for release date, platforms

3. **Audio Preview**
   - Built-in audio player for uploaded file
   - Waveform visualization using Web Audio API
   - Basic playback controls

4. **Form Integration**
   - Form validation with proper error messages
   - TypeScript interfaces for form data
   - Integration with React Hook Form or similar

Use modern React patterns (hooks, context if needed) and Tailwind for responsive, accessible styling.

Prompt 4: Audio Analysis Service

Create an audio analysis service that extracts musical features from uploaded audio files:

ANALYSIS FEATURES TO EXTRACT:
1. **Basic Properties**
   - Duration, sample rate, bit rate
   - File format and quality metrics

2. **Musical Features** (using Web Audio API + tone.js)
   - Tempo (BPM) detection
   - Key detection and chord analysis
   - Spectral analysis (frequency distribution)
   - Dynamic range and loudness (LUFS)

3. **Advanced Analysis**
   - Genre classification (create basic ML model)
   - Mood/energy detection based on tempo, key, dynamics
   - Instrumentation detection (drums, bass, guitar, etc.)
   - Vocal presence and characteristics

4. **Mock ML Integration**
   - Create placeholder functions for advanced ML features
   - Structure code to easily integrate real ML models later
   - Return confidence scores for all predictions

IMPLEMENTATION:
- Create service class with async methods for each analysis type
- Use Web Workers for heavy processing to avoid UI blocking
- Return structured data matching the database models
- Include error handling and fallback values

Prompt 5: Success Prediction Algorithm

Build a success prediction system that analyzes songs and compares them to market data:

PREDICTION COMPONENTS:
1. **Feature Comparison Engine**
   - Compare extracted audio features to successful songs database
   - Weight different features based on genre and current trends
   - Calculate similarity scores using cosine similarity or similar metrics

2. **Mock Market Database**
   - Create sample dataset of "successful" songs with features
   - Include various genres, time periods, and success metrics
   - Structure: { songFeatures, chartPerformance, streamingNumbers }

3. **Success Score Calculation**
   - Combine multiple factors: feature similarity, trend alignment, uniqueness
   - Generate score from 0-100%
   - Provide breakdown of contributing factors

4. **Insights Generation**
   - Identify strongest/weakest aspects of the song
   - Suggest improvements based on successful song patterns
   - Compare to similar artists and songs in the same genre

5. **Trend Analysis**
   - Mock current music trends (tempo ranges, popular keys, etc.)
   - Factor trending elements into success predictions
   - Provide market positioning insights

Create modular functions that can be easily updated with real ML models and market data later.

Prompt 6: Results Dashboard & Visualization

Create a comprehensive results dashboard that displays analysis results and predictions:

DASHBOARD SECTIONS:
1. **Success Score Overview**
   - Large, prominent success percentage (0-100%)
   - Color-coded indicator (red/yellow/green)
   - Brief summary of prediction confidence

2. **Audio Features Breakdown**
   - Visual charts for tempo, key, energy, mood
   - Comparison bars showing how song compares to genre averages
   - Interactive elements to explore each feature

3. **Market Comparison**
   - List of similar successful songs with similarity percentages
   - Performance metrics of comparable tracks
   - Genre ranking and positioning

4. **Actionable Insights**
   - Bullet points of strengths and improvement areas
   - Specific recommendations for increasing hit potential
   - Market timing suggestions

5. **Performance Metrics** (for released songs)
   - Streaming numbers, chart positions
   - Social media engagement metrics
   - Demographic breakdown of listeners

TECHNICAL REQUIREMENTS:
- Use Chart.js or similar for data visualizations
- Responsive design with mobile-first approach
- Loading states and skeleton screens
- Export functionality (PDF reports)
- Interactive tooltips and explanations

Prompt 7: API Routes & Backend Services

Create Express.js API routes and backend services for the songIQ application:

API ENDPOINTS:
1. **Song Management**
   - POST /api/songs/upload - Handle file upload and initial song creation
   - GET /api/songs/:id - Retrieve song details and analysis
   - PUT /api/songs/:id - Update song information
   - DELETE /api/songs/:id - Remove song and associated data

2. **Analysis Pipeline**
   - POST /api/analysis/start/:songId - Trigger analysis process
   - GET /api/analysis/status/:songId - Check analysis progress
   - GET /api/analysis/results/:songId - Retrieve completed analysis

3. **Market Data**
   - GET /api/market/compare/:songId - Get similar songs and market data
   - GET /api/market/trends/:genre - Current genre trends
   - GET /api/market/predictions/:songId - Success predictions

4. **Performance Tracking** (for released songs)
   - POST /api/performance/update/:songId - Update streaming/chart data
   - GET /api/performance/:songId - Retrieve performance metrics

BACKEND SERVICES:
- File storage service (local storage or cloud integration ready)
- Analysis queue system for processing uploaded songs
- Caching layer for market data and predictions
- Error handling and logging middleware
- Rate limiting and security measures

Include proper validation, error handling, and API documentation comments.

Prompt 8: Integration & Polish

```
Complete the songIQ application by integrating all components and adding polish:

INTEGRATION TASKS:
1. **Frontend-Backend Connection**
   - Set up API client with proper error handling
   - Implement loading states throughout the application
   - Add real-time updates for analysis progress

2. **User Experience Enhancements**
   - Add animations and transitions for smooth interactions
   - Implement toast notifications for user feedback
   - Create guided tour for first-time users

3. **Error Handling & Edge Cases**
   - Comprehensive error boundaries in React
   - Graceful degradation for failed API calls
   - User-friendly error messages and recovery options

4. **Performance Optimizations**
   - Lazy loading for heavy components
   - Image optimization for waveforms and charts
   - Caching strategies for repeated requests

5. **Testing & Deployment Prep**
   - Add basic unit tests for critical functions
   - Environment configuration for development/production
   - Build optimization and bundle analysis

6. **Documentation**
   - README with setup instructions
   - API documentation
   - Component documentation for future development

FINAL POLISH:
- Responsive design testing across devices
- Accessibility improvements (ARIA labels, keyboard navigation)
- SEO optimization for landing pages
- Analytics integration preparation
```

Usage Instructions

Use these prompts in sequence with Cursor, allowing each step to complete before moving to the next. Each prompt builds upon the previous work and can be customized based on your specific technical preferences or constraints.

For best results:

1. Start with Prompt 1 and let Cursor set up the basic project structure
2. Review the generated code and make any necessary adjustments
3. Proceed to the next prompt, referencing the existing codebase
4. Iterate on each section as needed before moving forward