

HarmoniX Documentation

- *Repository:** sach1nkhatri/HarmoniX

- *Ge

- - -

Table of Contents

1. [Overview](#overview)
2. [Architecture](#architecture)
3. [Core Components](#core-components)
4. [API Reference](#api-reference)
5. [Usage Examples](#usage-examples)

- - -

Overview

- Error generating content: Request timed out after 300 seconds. Please try again or check LM Studio connection.*

Architecture

- Error generating content: Request timed out after 300 seconds. Please try again or check LM Studio connection.*

Core Components

<think>

Okay, I need to generate comprehensive documentation for the "Core Components" chapter based on the provided code and other context. Let me start by understanding what's required.

First, the user wants a detailed technical documentation chapter that covers main components and modules, includes code examples, explanations, proper structure with subsections, code blocks with syntax highlighting, and practical examples where applicable. The output should be in markdown starting with "## Core Components".

Looking at the source code provided, there are several files mentioned from both frontend (React) and backend (Node.js/Express/MongoDB). The main components seem to be Dashboard, Fretboard, TheorySidebar, TunerWidget, and related modules. Also, backend models like User and LessonProgress are part of the core components.

I need to structure this into sections as per the required documentation outline: Title, Table of Contents, Overview, Architecture & Design, Components & Modules (each with classes, functions, modules), Usage Examples, Technical Details, Summary & Notes.

Starting with the **Overview** section. The purpose is to explain what Core Components are in HarmoniX Academy. High-level description would mention that these are the main building blocks of the application, including both frontend UI components and backend data models. Key technologies used include React, Node.js, Express, MongoDB, Mongoose, Tone.js, Tonal.js, etc.

Next, **Architecture & Design**. The overall structure is a client-server architecture with React on the frontend and Node.js/Express/MongoDB on the backend. Design patterns might include component-based architecture for React, MVC for backend, and maybe some state management using Context or Hooks. Component relationships would involve DashboardPage using Fretboard and TunerWidget, which in turn use other components like TheorySidebar.

For **Components & Modules**, I need to document each major component. Let's list them:

Frontend Components:

- Dashboard.jsx: Renders the dashboard layout with sidebar and main content.

- Fre

Backend Models:

- User.js: Mongoose model for user data (email, passwordHash, etc.).

- Les

Also, backend routes like userRoutes.js and progressRoutes.js handle API endpoints.

Each of these components needs a description, parameters, return types, dependencies, code examples where applicable.

In **Usage & Examples**, I should explain how to use DashboardPage or Fretboard component. Maybe show an example of importing and using them in another part of the app.

- *Technical Details* would cover error handling (like in backend routes if there's a validation error), edge cases such as when getUserMedia is not supported, performance considerations like

rendering optimization for Fretboard.

In **Summary & Notes**, highlight key takeaways about the components' roles and any potential improvements, like adding more error handling or optimizing component re-renders.

I need to make sure that all code examples are taken from the provided files. For instance, Dashboard.jsx has a React function component that returns JSX with Fretboard imported. Similarly, backend models have Mongoose schemas.

Also, check for any other relevant files mentioned in the context, like README.md which lists features and tech stack, but since the task is about Core Components, focus on the code files provided.

Make sure not to invent information not present in the source. For example, if a component doesn't have specific parameters or methods, don't assume them. Only document what's actually there.

Now, structuring all this into markdown with proper headings and sections as per the user's required format.

</think>

Core Components

- - -

Table of Contents

1. [Overview](#overview)
2. [Architecture & Design](#architecture--design)
3. [Components & Modules](#components--modules)
4. [Usage & Examples](#usage--examples)
5. [Technical Details](#technical-details)
6. [Summary & Notes](#summary--notes)

- - -

Overview

Purpose and Scope

The **Core Components** chapter documents the fundamental building blocks of the HarmoniX Academy application, including both frontend UI components and backend data models. These components enable core functionalities such as interactive fretboard visualization, user

authentication, progress tracking, and audio processing.

High-Level Description

- **Frontend**: React-based components for visualizing guitar scales, theory overlays, tuners, and dashboards.

- **E**
prog

- Backend: Express.js, Mongoose, MongoDB
- Audio Processing: Web Audio API, Tone.js

- - -

Architecture & Design

Overall Structure

The application follows a **client-server architecture** with the following layers:

- **Frontend**: React components for UI rendering and user interaction.

- **B**
via M

Design Patterns Used

- **Component-Based Architecture (React)**: Reusable UI elements like `Fretboard`, `TheorySidebar`, and `TunerWidget`.

- **M**
contr

Component Relationships

- `DashboardPage.jsx` imports `Fretboard` and `TunerWidget` to render interactive tools.

- `Fre

Data Flow

1. **Frontend** components trigger actions (e.g., user clicks on a scale).
2. **Backend** APIs process requests, update MongoDB records, and return responses.
3. Updated state is reflected in frontend UI via re-renders or context updates.

- - -

Components & Modules

Frontend Components

1. `Dashboard.jsx`

- **Purpose**: Renders the main dashboard layout with sidebar navigation and content area.

- **D

- Uses CSS from `../css/dashboard.css`.

- **Code Example**:

```
```jsx
export default function Dashboard() {
 return (
 <div className="dashboard-shell">
 <aside className="dash-sidebar">
 <h2>GuitarLab</h2>
 <p>Tools</p>
 </aside>
 <main className="dash-main">
 <h1>Dashboard</h1>
 <Fretboard />
 </main>
 </div>
);
}
```
```

```

#### 2. `Fretboard.jsx`

- \*\*Purpose\*\*: Interactive visual representation of a guitar fretboard.

- \*\*D

- Likely integrates with Tone.js for audio playback and Tonal.js for scale logic (not shown in code snippet).

#### 3. `TheorySidebar.jsx`

- \*\*Purpose\*\*: Displays music theory information, intervals, and chord relationships.

#### 4. `TunerWidget.jsx`

- \*\*Purpose\*\*: Live tuner using browser microphone input.

- \*\*C

```
export default function TunerWidget() {
 // Uses getUserMedia for live audio input
 // Displays note name, frequency, and pitch deviation
}
```

...

#### 5. `DashboardPage.jsx`

- \*\*Purpose\*\*: Main dashboard page with hero section, widgets, and interactive tools.

- \*\*D

- Uses CSS from `../css/dashboard.css`.

---

### Backend Modules

#### 1. `User.js` (Mongoose Model)

- \*\*Purpose\*\*: Represents user data stored in MongoDB.

- \*\*S

```
{
 email: { type: String, required: true, unique: true },
 passwordHash: { type: String, required: true },
 displayName: { type: String },
 xp: { type: Number, default: 0 },
 streak: { type: Number, default: 0 }
}
```

...

#### 2. `LessonProgress.js` (Mongoose Model)

- \*\*Purpose\*\*: Tracks user progress in lessons.

- \*\*S

```
{
 userId: {
 type: mongoose.Schema.Types.ObjectId,
 ref: 'User',
 required: true
 },
 lessonId: { type: String, required: true },
 status: {
 type: String,
 enum: ['not_started','in_progress','completed'],
 default: 'not_started'
 },
 xpAwarded: { type: Number, default: 0 }
}
...
}
```

### 3. `userRoutes.js` (Backend Route)

- **Purpose**: Handles user authentication and profile management.

```
router.post('/register', ctrl.register);
router.post('/login', ctrl.login);
router.get('/me', ctrl.me);
...
-
```

### 4. `progressRoutes.js` (Backend Route)

- **Purpose**: Manages lesson progress tracking.

```
router.get('/', ctrl.getProgress);
router.post('/', ctrl.updateProgress);
...
- --
```

## Usage & Examples

## Frontend Example: Rendering DashboardPage

```
```jsx
import DashboardPage from 'src/features/dashboard/page/DashboardPage';

function App() {
  return <DashboardPage />;
}

```

```

## Backend Example: User Registration Flow

1. \*\*Frontend\*\*: User submits registration form.
  2. \*\*Backend\*\*:
    - `userRoutes.js` triggers `ctrl.register`.
    - Controller validates input, hashes password, and saves to MongoDB via `User` model.
- 

## Technical Details

### Error Handling Approach

- \*\*Frontend\*\*: Uses React error boundaries for component-level errors (not explicitly shown in code).

- \*\*E  
hand

### Edge Cases and Considerations

- \*\*TunerWidget\*\*: Requires browser support for `getUserMedia`. Fallback behavior not implemented.

- \*\*L  
`com

### Performance Characteristics

- \*\*Fretboard Rendering\*\*: Likely optimized with memoization or virtual rendering (not shown in code).

- \*\*D  
`less

---

## Summary & Notes

### Key Takeaways

- Core components are modular, reusable, and decoupled between frontend and backend.
- Frontend is built with React and Redux, while the backend is Node.js with Express.js.

### Potential Improvements

- Add error handling in `TunerWidget` for unsupported browsers.
- Implement better authentication and authorization.

### Important Considerations

- Ensure MongoDB connection stability via `connectDb.js`.
- Frontend needs to handle state management effectively using Redux.

## API Reference

<think>

Okay, I need to generate a comprehensive documentation chapter for the "API Reference" section of the HarmoniX project. The user provided several code files from both the frontend and backend. Let me start by understanding what each part does.

First, looking at the backend server.js file. It sets up an Express server with CORS and JSON parsing. There are routes for /api/users and /api/progress, which probably handle user authentication and progress tracking. The health check endpoint is also there. I should document these endpoints, their methods, parameters, and responses.

Next, the routes files: progressRoutes.js and userRoutes.js. These define the specific routes and link them to controller functions. For example, /api/progress has GET and POST methods handled by getProgress and updateProgress in lessonController. Similarly, /api/users has register, login, and me endpoints using userController.

The frontend App.js uses React Router for navigation between different pages like Dashboard, Fretboard, Tuner, etc. But since the task is about API references, maybe focus more on backend APIs unless there are frontend interfaces involved. However, looking at the files, most of the frontend code seems to be UI components and hooks, not APIs.

The README.md mentions a tech stack with Node.js, Express.js, MongoDB, etc., so the backend is using these technologies. The API endpoints would be part of this backend setup.

Now, I need to structure the documentation as per the user's required format: Title, Table of Contents, Overview, Architecture & Design, Components & Modules, Usage & Examples, Technical Details, Summary & Notes.

In the Overview section, explain the purpose of the APIs?user management and progress tracking. Mention that they're built with Express.js and MongoDB.

Architecture & Design would describe the server setup, use of middleware (CORS, JSON parsing), route organization, and database connection via Mongoose.

Components & Modules should detail each route file and their controllers. For example, userRoutes handle authentication endpoints, while progressRoutes manage user progress data. Also mention the health check endpoint.

Usage examples could include sample requests for registering a user, logging in, fetching progress, etc., using curl or Postman commands.

Technical Details would cover error handling (like DB connection failures), edge cases such as invalid inputs, and performance considerations like using async/await with MongoDB.

I need to make sure all the code examples from the provided files are included. For instance, the server.js code block showing the Express setup, the route definitions in progressRoutes.js and userRoutes.js.

Also, check if there's any other relevant API information. The README mentions JWT for auth, so maybe note that authentication is handled via JWT tokens, which would be part of the login endpoint's response.

Make sure not to include frontend-specific code unless it's related to APIs. For example, the SignupPage.jsx is a frontend component but doesn't relate directly to an API endpoint, so focus on backend files.

Finally, summarize key points and note any potential improvements like adding more detailed error messages or rate limiting for endpoints.

</think>

## API Reference

- - -

### Table of Contents

1. [Overview](#overview)

2. [Architecture & Design](#architecture--design)
  3. [Components & Modules](#components--modules)
  4. [Usage & Examples](#usage--examples)
  5. [Technical Details](#technical-details)
  6. [Summary & Notes](#summary--notes)
- - -

## Overview

### Purpose and Scope

The API endpoints provide core functionality for user authentication, progress tracking, and application health monitoring in the HarmoniX Academy platform. These interfaces enable frontend components to interact with backend services securely and efficiently.

### High-Level Description

- \*\*User Management\*\*: Registration, login, and profile access via JWT-based authentication.
- \*\*P

### Key Technologies/Frameworks Used

- \*\*Backend\*\*: Express.js (routing), MongoDB (data storage), Mongoose (ODM).
- \*\*A

- - -

## Architecture & Design

### Overall Structure

The backend is a RESTful API built using Express.js, organized into modular routes and controllers. Key components include:

- \*\*Server Initialization\*\* (`server.js`): Sets up middleware, connects to MongoDB, and initializes routes.
- \*\*R

### Design Patterns Used

- \*\*Separation of Concerns\*\*: Routes handle HTTP requests; controllers manage data operations.

## Component Relationships

- \*\*`server.js`\*\* initializes the Express app and connects to MongoDB.

## Data Flow

1. Client sends HTTP request (e.g., POST `/api/users/register`).
  2. Express routes dispatch the request to the appropriate controller function.
  3. Controller interacts with MongoDB to perform operations (create user, update progress).
  4. Response is sent back to the client in JSON format.
- - -

## Components & Modules

### Backend Server (`server.js`)

```
```javascript
const express = require('express');
const cors = require('cors');
const dotenv = require('dotenv');
const connectDb = require('./config/db');

dotenv.config();
const app = express();

app.use(cors());
app.use(express.json());

// Routes
app.use('/api/users', require('./routes/userRoutes'));
app.use('/api/progress', require('./routes/progressRoutes'));

// Health check
app.get('/health', (req, res) => res.json({ ok: true }));
```

```

```
const PORT = process.env.PORT || 4000;

connectDb().then(() => {
 app.listen(PORT, () => console.log(`HarmoniX backend running on :${PORT}`));
}).catch(err => {
 console.error('DB connection failed', err);
 process.exit(1);
});
```

```

- *Purpose*: Initializes the Express server, sets up middleware, and connects to MongoDB.

- - -

User Routes (`userRoutes.js`)

```
```javascript
const router = require('express').Router();
const ctrl = require('../controllers/userController');

router.post('/register', ctrl.register);
router.post('/login', ctrl.login);
router.get('/me', ctrl.me);

module.exports = router;
```

```

- *Endpoints*:

- *Returns*: User object and JWT token on success.

- **POST `/api/users/login`**: Authenticates a user.

- *Returns*: JWT token on success.

- **GET `/api/users/me`**: Retrieves the currently authenticated user's profile.

- - -

Progress Routes (`progressRoutes.js`)

```
```javascript
const router = require('express').Router();
const ctrl = require('../controllers/lessonController');

router.get('/', ctrl.getProgress);
router.post('/', ctrl.updateProgress);

module.exports = router;
```

```

- *Endpoints*:

- **POST `/api/progress`**: Updates user progress (e.g., lesson completion).

- *Returns*: Updated progress object.

- - -

Health Check Endpoint

- *GET `/health`*

- { "ok": true }

```

- - -

## Usage & Examples

### Example: User Registration

- \*Request\*:

- POST /api/users/register

- Content-Type: application/json

- {

- \*\*G

- \*P

- \*Pu

```ba

```
"username": "sachin",
"email": "sachin@example.com",
"password": "SecurePass123!"
}
```

```

- \*Response (Success)\*\*:

```
```json
{
  "user": {
    "_id": "64a1b2c3d4e5f67890abcdef",
    "username": "sachin",
    "email": "sachin@example.com"
  },
  "token": "eyJhbGciOiJIUzI1NilsInR5cCI6IkpXVCJ9..."
}
```

```

--

### Example: Update Progress

- \*Request\*\*:

```
POST /api/progress
Content-Type: application/json
Authorization: Bearer <JWT_TOKEN>
```

```
{
 "userId": "64a1b2c3d4e5f67890abcdef",
 "lessonId": "intro_to_scales",
 "status": "completed"
}
```

```

- *Response (Success)**:

```
```json
{
 "progress": {
 "_id": "64a1b2c3d4e5f67890abcdef",
 "status": "completed"
 }
}
```

```

```
"userId": "64a1b2c3d4e5f67890abcdef",
"lessonId": "intro_to_scales",
"status": "completed"
}
}
```
- --
```

## Technical Details

### Error Handling

- **Database Connection Failures**: Caught in `server.js` and logged to console.

- **Im**

- --

### Edge Cases & Considerations

- **Rate Limiting**: Not implemented in current version; may be added later.

- **Ir**  
(e.g..

- --

## Summary & Notes

### Key Takeaways

- The API provides essential functionality for user authentication and progress tracking.

- Mod

### Potential Improvements

- Add rate limiting to prevent abuse of endpoints.

- Imp

### Important Considerations

- Ensure MongoDB connection strings are securely managed (e.g., via environment variables).

- Req

## Usage Examples

- Error generating content: Request timed out after 300 seconds. Please try again or check LM Studio connection.\*