Backend Administration, Training & Integration Module

Developed by: SB Barath Kumaran

---

1. Module Overview

This module represents my primary backend contribution to the Internship Burgeonpath Solutions
It focuses on **administrative management, training systems, certificate handling, support ticketing, and backend integration**.

In addition to implementing core backend features, I was responsible for **integrating and unifying the backend work of all team members** into **a single, production-ready backend application** with consistent authentication, authorization, and database design.

The module covers:

- Training & learning management
- Certificate handling with cloud storage
- Ticket & support workflows
- Backend service integration across teams

---

2. Tech Stack Used

Backend & Runtime

- Node.js (ES Modules)
- Express.js
- Morgan (request logging)

Databases

- MongoDB
    - Training modules
    - Training progress
    - Certificates
    - Tickets
- Redis
    - Caching
    - Token lifecycle management

Cloud & External Services

- AWS S3 (certificate & training file storage)

Security & Utilities

- dotenv for environment management
- Centralized error handling

---

3. Architecture Overview

The backend follows a modular layered architecture:

Client → Routes → Middleware → Controllers → Services → Databases

- Routes define REST API contracts
- Middleware handles authentication, authorization, and errors
- Controllers manage request–response logic
- Services encapsulate business logic
- Models define MongoDB schemas
- Prisma manages relational data consistency

This architecture ensures scalability, maintainability, and clean separation of concerns.

---

4. Functional Scope of the Module

4.1 Admin Management

- Admin dashboard APIs
- Listener approval workflow
- Role-restricted endpoints
- System-level audit logging

4.2 Training & Learning Management

- Training module creation
- File-based learning resources
- Progress tracking per listener
- Preview-based progress updates
- MongoDB-backed persistence

4.3 Certificate Management

- Secure certificate upload
- AWS S3 integration
- Listener-specific access control
- Certificate retrieval APIs

4.4 Ticket & Support System

- Ticket creation and retrieval
- Admin-only resolution workflow
- Status-based ticket lifecycle
- MongoDB persistence

4.5 Authentication & Authorization

- Role-based access enforcement
- Centralized middleware usage

---

5. Database Design

MongoDB Collections

- training_modules
- training_progress
- certificates
- tickets

---

6. APIs Delivered

Admin APIs

- GET /api/admin/tickets

Training APIs

- POST /api/training/modules
- GET /api/training/modules
- POST /api/training/preview
- GET /api/training/progress/:moduleId

Certificate APIs

- POST /api/listener/certificates/upload
- GET /api/listener/certificates/my

Ticket APIs

- GET /api/admin/tickets
- PUT /api/admin/tickets/:id/resolve

---

7. Final File Structure – My Work

src/

├── config/
│ ├── mongo.js
│ ├── redis.js
│ └── s3.js
├── controllers/
│ ├── admin.controller.js
│ ├── training.controller.js
│ ├── certificate.controller.js
│ └── ticket.controller.js
├── middleware/
│ ├── auth.js
│ ├── role.js
│ └── errorHandler.js
├── models/
│ ├── TrainingModule.js
│ ├── TrainingProgress.js
│ ├── Certificate.js
│ ├── Ticket.js
│ └── AuditLog.js
├── routes/

```
│   ├── admin.routes.js
│   ├── training.routes.js
│   ├── certificate.routes.js
│   └── ticket.routes.js
├── utils/
│   ├── logger.js
│   └── s3Upload.js
├── app.js
└── server.js
├── .env
├── package.json
└── package-lock.json
```

---

8. Final File Structure – After Full Team Integration

```
backend/
├── prisma/
│   ├── migrations/
│   └── schema.prisma
│
├── src/
│   ├── config/
│   │   ├── jwt.js
│   │   ├── mongo.js
│   │   ├── postgres.js
│   │   ├── redis.js
│   │   └── s3.js
│   │
│   ├── controllers/
│   │   ├── admin.controller.js
```

```
│   │   ├── auth.controller.js
│   │   ├── availability.controller.js
│   │   ├── certificate.controller.js
│   │   ├── earnings.controller.js
│   │   ├── listener.controller.js
│   │   ├── report.controller.js
│   │   ├── session.controller.js
│   │   ├── support.controller.js
│   │   ├── ticket.controller.js
│   │   ├── training.controller.js
│   │   ├── user.controller.js
│   │   └── wallet.controller.js
│   │
│   ├── middleware/
│   │   ├── auth.js
│   │   ├── role.js
│   │   ├── errorHandler.js
│   │   └── validation.middleware.js
│   │
│   ├── models/
│   │   ├── AuditLog.js
│   │   ├── Certificate.js
│   │   ├── chatLog.model.js
│   │   ├── moderationReport.model.js
│   │   ├── supportTicket.model.js
│   │   ├── Ticket.js
│   │   ├── TrainingModule.js
│   │   └── TrainingProgress.js
│   │
```

```
│   ├── routes/
│   │   ├── admin.routes.js
│   │   ├── auth.routes.js
│   │   ├── availability.routes.js
│   │   ├── certificate.routes.js
│   │   ├── listener.routes.js
│   │   ├── payout.routes.js
│   │   ├── report.routes.js
│   │   ├── session.routes.js
│   │   ├── support.routes.js
│   │   ├── ticket.routes.js
│   │   ├── training.routes.js
│   │   ├── user.routes.js
│   │   └── wallet.routes.js
│   │
│   ├── services/
│   │   ├── audit.service.js
│   │   ├── chat.service.js
│   │   ├── payment.service.js
│   │   ├── report.service.js
│   │   └── support.service.js
│   │
│   ├── sockets/
│   │   ├── index.js
│   │   ├── sessionChat.socket.js
│   │   └── supportChat.socket.js
│   │
│   ├── utils/
│   │   ├── auditLogger.js
```

```
│   │   ├── logger.js
│   │   ├── s3Upload.js
│   │   └── sessionTimer.js
│   │
│   ├── app.js
│   ├── prismaClient.js
│   └── server.js
│
├── .env
├── package.json
└── package-lock.json
```

---

9. Integration Responsibility

I was responsible for:

- Merging all backend modules into a single codebase
- Resolving dependency conflicts
- Aligning authentication across all services
- Ensuring Prisma schema consistency
- Securing all routes using role-based middleware

---

10. Module Outcome

This backend module delivers:

- Secure admin workflows
- Scalable training and learning management
- Cloud-based file handling
- Reliable ticket & support system
- Seamless integration of multi-developer backend work

The final backend system is production-ready, fully integrated, and aligned with enterprise-level backend standards.

Wallet & Payouts Backend Module Work Report

Developed by: Jeffrin Stewart P

---

1. Module Overview

This module is responsible for handling all **financial transactions, listener earnings, wallet management, and withdrawal workflows** for the platform. It is designed as a secure, transactional backend subsystem that ensures accurate financial tracking and strict compliance with the platform's monetary policies.

The module focuses on:

- Real-time wallet balance management (Credits/Debits).
- Listener earning tracking and dashboard statistics.
- Secure withdrawal processing with strict validation rules (Min/Max limits).
- Admin payout approval and integration with payment gateways (Razorpay).
- Transaction history and auditability.

---

2. Tech Stack Used

Backend & Runtime

- **Node.js (ES Modules):** Modern JavaScript standard using import/export.
- **Express.js:** REST API Framework.
- **Prisma ORM:** Type-safe database access.

Databases

- **PostgreSQL:** Relational database for structured financial data (Wallets, Transactions, Payouts).

Integrations & Utilities

- **Razorpay:** Payment gateway for processing payouts.
- **dotenv:** Environment configuration management.
- **Cors:** Cross-Origin Resource Sharing handling.

---

3. Architecture Overview

The module follows a **layered architecture** to ensure separation of concerns and security: Client → Routes → Controllers → Services → Database (Prisma)

- **Routes:** Expose secure REST APIs.
- **Controllers:** Handle request validation and business logic flow.
- **Services:** Encapsulate external integrations (e.g., PaymentService for Razorpay).
- **Prisma Client:** Manages direct database interactions and transactions.

---

4. Functional Scope of the Module

4.1 Wallet Management

- Maintains user and listener wallet balances.
- Supports atomic credit and debit operations.
- **Strict Formula:** Available Balance = Lifetime Earnings - (Total Withdrawn + Pending Requests).

4.2 Withdrawal Requests

- Listeners can request payouts from their available balance.
- Validation Rules Implemented:
    - **Minimum Withdrawal:** ₹500.
    - **Maximum Withdrawal:** ₹50,000 per transaction.
    - **Anti-Spam:** Only **one active request** allowed at a time.

4.3 Admin Payout Processing

- Admins can view and approve pending withdrawal requests.
- Integration with **Razorpay Payouts** for fund transfers.
- Mock mode support for development testing.

4.4 Dashboard Statistics

- Provides real-time financial metrics to listeners.
- Aggregates Total Earnings, Total Withdrawn, and Pending amounts.

---

5. Database Design (Prisma Schema)

Key Models:

- **Wallet:** Stores current balance and currency for each user.
- **Transaction:** Logs every credit/debit action for audit trails.
- **Earning:** Records individual session earnings for listeners.
- **Payout:** Tracks withdrawal requests and their status (REQUESTED, APPROVED, PAID).

---

6. APIs Deliverables

Wallet Routes (/api/wallet)

- GET /balance/:userId - Fetch current wallet balance.
- POST /add - Credit funds to wallet (Mock/Admin use).
- POST /deduct - Debit funds (System use).

Payout & Earnings Routes (/api/payments)

- GET /listener/stats/:listenerId - Get comprehensive earning stats.
- POST /withdraw/request - Initiate a withdrawal (with validation).
- POST /admin/payout/approve - Admin approval triggers Razorpay transfer.

---

7. Final File Structure

```
Backend-wallet/
├── controllers/
│   ├── earningsController.js    (Stats & Withdrawal logic)
│   └── walletController.js      (Balance & Transaction logic)
├── routes/
│   ├── payoutRoutes.js          (API definitions for payments)
│   └── walletRoutes.js          (API definitions for wallets)
├── services/
│   └── paymentService.js        (Razorpay integration wrapper)
├── prisma/
│   └── schema.prisma            (Database schema definition)
├── app.js                       (Main application entry point)
├── package.json                 (Project configuration)
└── .env                         (Environment variables)
```

---

8. Key Utilities

paymentService.js

- Centralized service for handling Razorpay interactions.
- Includes a **Mock Mode** toggle to simulate payments without real money during development.

earningsController.js (Logic Core)

- Contains the strict financial math mandated by the Requirement Document.
- Enforces the "Minimum ₹500" and "One Request Rule" logic blocks.

---

9. ES Module Compliance

- **Modern Syntax:** The entire module has been converted to **ES Modules** (import / export).
- **Configuration:** package.json explicitly sets "type": "module".
- **Compatibility:** All local file imports use the required .js extension (e.g., import ... from './file.js').

---

10. Module Outcome

This module delivers a production-ready financial backend that enables:

- **Secure Transactions:** Prevents overdrafts and ensures data integrity.
- **Policy Compliance:** Strictly adheres to the ₹500-₹50,000 withdrawal limits.
- **Scalability:** Built on Prisma and Postgres to handle high transaction volumes.
- **Ready for Integration:** Fully "Merge Ready" with clear API endpoints for the frontend.

AUTH, LISTENER CORE & AVAILABILITY

Developer: Varshaa T

Status: Production-Ready (Verified with Company Credentials)

Stack:

1. Node.js
2. Express.js
3. PostgreSQL
4. Prisma ORM
5. Redis
6. JWT.

Database:

1. User
2. Listener
3. Availability
4. Session
5. Booking
6. Rating

---

1. System Architecture & Infrastructure

1.1 Overview

The system is built on a modular, stateless REST architecture. It manages three distinct user roles: **Admin**, **User**, and **Listener**. The backend is designed for high performance using Redis for session invalidation and PostgreSQL for persistent relational data.

1.2 The Infrastructure Stack

- **Database (PostgreSQL):** Hosted on the company's server. Managed via Prisma ORM for type-safe queries and schema synchronization.
- **Caching & Session (Redis):** Handles session blacklisting and real-time availability caching. All active logins are tracked here.
- **Authentication (JWT):** All tokens are signed with a secret and carry a **24-hour expiration (1 day)**.

1.3 Environment Configuration

The .env file must be configured with the following (kept in .env.local for backup):

- DATABASE_URL: Connection string for the company PostgreSQL.
- REDIS_URL: Connection link for company Redis.
- JWT_SECRET: Private key for token signing.

1.4 Authentication and Security Policy

- **Token Validity:** All JWTs are set to expire in **24 hours (1 day)**.
- **Redis Integration:** On logout, tokens are blacklisted/deleted in Redis. Even if a token hasn't expired, it will be rejected if not present in the active Redis session store.
- **Header Format: * Key:** Authorization
    - **Value:** Bearer <token_string>
- **Logging:** Monitor the console for "✅ Connected to Redis" and "✅ Connected to Database" on startup
- **Prisma:** Run npx prisma generate after any schema changes.

Sample Tokens (For Testing Reference):

Note: These are examples from the test environment.

- **Admin:** Bearer eyJhbGci...makUk
- **User:** Bearer eyJhbGci...Zmw
- **Listener:** Bearer eyJhbGci...Cn7c

2. Detailed Project Structure

```
auth-core/
├── prisma/
│   ├── schema.prisma       # Core database definitions
│   └── migrations/
├── src/
│   ├── config/
│   │   ├── jwt.js
│   │
│   ├── controllers/
│   │   ├── auth.controller.js
```

```
│   │   ├── user.controller.js
│   │   ├── listener.controller.js
│   │   ├── availability.controller.js
│   │   └── session.controller.js sessions
│   ├── middleware/
│   │   ├── auth.middleware.js
│   │   └── role.middleware.js
│   ├── routes/
│   │   ├── auth.routes.js
│   │   ├── user.routes.js
│   │   ├── listener.routes.js
│   │   ├── availability.routes.js
│   │   └── session.routes.js
│   ├── app.js
│   ├── redis.js
│   └── prismaClent.js
│
└── package.json
```

3. Detailed Module Documentation

A. Admin Module

Responsible for system-wide integrity and Listener verification.

- **Logic:** Listeners are registered with isApproved: false. They cannot log in until an Admin toggles this status.
- **Promotion:** Currently, Admins are promoted manually via **Prisma Studio** by changing the role column in the User table.

B. User Module

Handles person-to-person support seeking.

- **Profile Management:** Supports fetching and updating personal metadata (Age, Language, Category).

- **Deactivation:** Implements "Soft Delete." The user record is marked as inactive, and their token is invalidated.

C. Listener Module

The core service provider module.

- **Availability:** Uses a custom logic to post available slots.
- **Dashboard:** Aggregates session data to show the Listener their performance metrics.
- **Approval Gate:** Middleware specifically checks the isApproved flag during the login process.

D. Session & Booking Module

Manages the lifecycle of an interaction.

- **Status Workflow:** PENDING → ACCEPTED / REJECTED.
- **Conflict Prevention:** The system checks the Listener's availability table before allowing a user to POST a request for a specific time.

SPECIFICATIONS

1 Authentication & Redis Session Management

- **Logic:** Upon login, a JWT is generated. Simultaneously, the session ID is stored in Redis.
- **Token Expiry:** 24 Hours.
- **Logout:** When a user logs out, the key is removed from Redis. The authMiddleware checks Redis on every request; if the key is missing, the token is rejected even if it hasn't expired.

2 The Listener Approval Workflow

- Listeners are registered with isApproved: false.
- They are barred from logging in by a specific check in authController.js.
- **Admin Promotion:** Handled manually via Prisma Studio (npx prisma studio) by updating the role field to ADMIN.

3. Availability & Session Logic

- **Conflict Prevention:** The system checks for existing slots before allowing a Listener to post availability.
- **Session Lifecycle:** A Session begins as PENDING when a User requests it. The Listener can then toggle the status to ACCEPTED or REJECTED.

---

4. API Inventory & Documentation

Admin:

PUT http://localhost:5000/api/listener/approve/1

User:

POST http://localhost:5000/api/auth/user/register

POST http://localhost:5000/api/auth/user/login

GET http://localhost:5000/api/user/profile

PUT http://localhost:5000/api/user/profile (Updating User)

POST http://localhost:5000/api/sessions/request

PUT http://localhost:5000/api/user/deactivate

POST http://localhost:5000/api/auth/logout

Listener:

POST http://localhost:5000/api/auth/listener/register

POST http://localhost:5000/api/auth/listener/login

GET http://localhost:5000/api/listener/profile

PUT http://localhost:5000/api/listener/update

POST http://localhost:5000/api/availability

GET http://localhost:5000/api/availability

GET http://localhost:5000/api/sessions/requests

PUT http://localhost:5000/api/sessions/1 (Accepting/Rejecting Session id 1)

GET http://localhost:5000/api/sessions

GET http://localhost:5000/api/listener/dashboard

Real-Time Session, Support & Moderation Backend Module

Developed by: Madhumithaa P (Team Lead)

---

1. Module Overview

This module is responsible for handling all **real-time communication, moderation, support, and audit-related backend infrastructure** for the HMM Talk platform. It is designed as an independent backend subsystem that integrates seamlessly with the core authentication, user, listener, and session management services built by other team members.

The module focuses on: - Real-time chat and signaling - User reporting and moderation workflows - Support ticketing with live chat - Session lifecycle and timer tracking - Centralized audit logging

---

2. Tech Stack Used

Backend & Runtime

- Node.js (ES Modules)
- Express.js
- Socket.IO

Databases

- MongoDB
    - Chat logs
    - Moderation reports
    - Support tickets
    - Audit logs
- Redis
    - Session timers
    - In-memory session state

Security & Utilities

- JWT-based authentication (role-aware)
- dotenv for environment configuration
- Nodemon for development

---

3. Architecture Overview

The module follows a layered architecture to ensure scalability and maintainability:

Client → Routes / Sockets → Controllers → Services → Databases

- **Routes** expose REST APIs
- **Sockets** handle real-time communication

- **Controllers** manage request flow
- **Services** contain business logic
- **Models** define MongoDB schemas
- **Utils** handle shared cross-cutting concerns

---

4. Functional Scope of the Module

4.1 Real-Time Session Chat

- Session-based chat rooms
- Socket namespace: /session-chat
- Messages persisted in MongoDB
- Supports WebRTC signaling (offer, answer, ICE candidates)

4.2 Support Ticket System

- Users can raise support tickets
- Admins and listeners can respond
- Real-time chat via /support-chat namespace
- All messages stored per ticket

4.3 Reporting & Moderation

- Users can report sessions or users
- Reports stored in MongoDB moderation queue
- Admins can update report status
- All actions are audit-logged

4.4 Session Lifecycle & Timer Tracking

- Session start timestamp stored in Redis
- Session duration calculated on end
- No dependency on local or relational DB for timers

4.5 Audit Logging System

- Centralized audit logging for sensitive actions
- Logs include:
    - Actor ID
    - Actor role (USER / LISTENER / ADMIN)
    - Action performed
    - Entity type & ID
    - Metadata

---

5. Database Design

MongoDB Collections

- chat_logs
- moderation_reports

- support_tickets
- audit_logs

These collections are automatically created by Mongoose when data is first inserted.

Redis Keys

- session:{sessionId}:start → stores session start timestamp

---

6. APIs & Socket Deliverables

REST APIs

- POST /api/session/:id/report
- POST /api/support/ticket
- GET /api/support/tickets
- PATCH /api/support/ticket/:id/status

Socket Namespaces

- /session-chat
- /support-chat

---

7. Final File Structure

```
src/
├── config/
│   ├── db.js
│   └── redis.js
├── controllers/
│   ├── report.controller.js
│   └── support.controller.js
├── middlewares/
│   ├── auth.middleware.js
│   ├── error.middleware.js
│   └── validation.middleware.js
├── models/
│   ├── auditLog.model.js
│   ├── chatLog.model.js
│   ├── moderationReport.model.js
│   └── supportTicket.model.js
├── routes/
│   ├── report.routes.js
│   └── support.routes.js
├── services/
│   ├── audit.service.js
│   ├── chat.service.js
│   ├── report.service.js
│   └── support.service.js
├── sockets/
```

```
│   ├── index.js
│   ├── sessionChat.socket.js
│   └── supportChat.socket.js
├── utils/
│   ├── auditLogger.js
│   └── sessionTimer.js
├── app.js
├── server.js
└── package.json
```

---

8. Key Utilities

auditLogger.js

- Centralized audit logging utility
- Used across services and sockets
- Fault-tolerant (does not break flows)

sessionTimer.js

- Stores session start time in Redis
- Computes session duration on end
- Keeps session logic stateless

---

9. ES Module Compliance

- Entire module uses ES module syntax
- Consistent import / export
- Configured via "type": "module" in package.json

---

10. Module Outcome

This module delivers a production-ready backend subsystem that enables: - Real-time communication - Secure moderation workflows - Reliable support handling - Accurate session tracking - Complete auditability

The implementation is scalable, maintainable, and fully aligned with enterprise backend standards.

End of Document