# Knowledge Transfer Documentation - DHTE NestJS Monorepo

## Project Overview

This document provides an overview of the ZMA NestJS Monorepo project architecture, key services, and technical stack to help new team members onboard effectively.

The project is a comprehensive microservices-based platform built with NestJS, organized as a monorepo using Nx. It follows modern best practices for scalable, maintainable Node.js applications with a GraphQL-first approach for most services, complemented by gRPC and Kafka for service-to-service communication.

## Technical Stack

* **Framework**: NestJS v11
* **Language**: TypeScript
* **Package Manager**: pnpm (v10+)
* **Node Version**: Node.js 24+ (managed via Volta)
* **Monorepo Management**: Nx
* **API Paradigm**: GraphQL (Yoga), gRPC, WebSockets, REST
* **Database**: MongoDB (via Mongoose)
* **Messaging**: Kafka
* **Real-time Communication**: Socket.IO, WebRTC (via mediasoup)
* **Caching**: Redis
* **Containerization**: Docker
* **CI/CD**: GitHub Actions, Semaphore
* **Development Tools**: ESLint, Prettier, Jest, commitlint

## Core Architecture Concepts

1. **Monorepo Structure**
   * apps/: Contains all microservices
   * libs/: Shared libraries and utilities
   * graphqls/: Generated GraphQL schemas
2. **Service Communication**
   * **GraphQL**: Primary API interface for frontend clients
   * **gRPC**: Efficient service-to-service communication
   * **Kafka**: Event-driven communication, async processing
   * **WebSockets**: Real-time communication for signaling and live updates
3. **Domain-Driven Design** Each service follows similar architecture patterns:
   * controllers/: GraphQL resolvers, gRPC controllers, REST endpoints
   * use-cases/: Business logic implementation
   * services/: Infrastructure services
   * core/: Domain models, DTOs, interfaces
   * common/: Shared utilities specific to the service
   * frameworks/: Third-party integrations

## Key Services Overview

### 1. zma-auth

**Purpose**: Authentication and authorization service.

**Key Features**:

* JWT-based authentication
* Multiple auth strategies (email/password, social logins like Google, Zalo)
* Token generation, validation and refresh
* Password reset and email verification

**Communication Methods**:

* GraphQL API for client authentication
* Works closely with zma-user service via gRPC

**Notable Endpoints**:

* authServiceEmailLogin: Email-password authentication
* authServiceRegister: User registration
* authServiceRefreshToken: JWT refresh
* authServiceImpersonateUser: User impersonation (admin feature)
* Multiple social login integrations

### 2. zma-user

**Purpose**: User management and profile service.

**Key Features**:

* User CRUD operations
* Profile management
* User search and filtering
* Tenant-based user segmentation
* Social provider integration

**Communication Methods**:

* GraphQL API for client applications
* gRPC server for internal service-to-service communication
* Kafka consumer for event-driven updates

**Notable Components**:

* gRPC server exposing user management functions to other services
* Extensive user search capabilities
* Tenant-scoped user data access

**Database Schema**:

* Users collection with tenant-based data segregation
* Social provider integration (Google, Facebook, Zalo)
* Authentication tracking (failed login attempts, verification)

### 3. zma-meeting

**Purpose**: Video conferencing and online meeting management.

**Key Features**:

* Meeting room creation
* Integration with signaling service for WebRTC

**Communication Methods**:

* GraphQL API for meeting management
* Integrates with zma-signaling for real-time communication

**Notable Endpoints**:

* meetingServiceCreateRoom: Creates a new meeting room

### 4. zma-dictionary

**Purpose**: Dictionary and language services.

**Key Features**:

* Dictionary management
* Word and definition storage
* Web crawling for dictionary content

**Communication Methods**:

* GraphQL API

**Notable Endpoints**:

* Dictionary CRUD operations
* Word crawling functionality

### 5. zma-signaling

**Purpose**: WebRTC signaling server for real-time communication.

**Key Features**:

* WebSocket-based signaling for WebRTC
* Room management and user presence
* Media capabilities negotiation
* Audio/video streaming with mediasoup
* Recording functionality

**Communication Methods**:

* WebSockets (Socket.IO)
* Works with mediasoup for WebRTC SFU functionality

**Key Components**:

* WebSocket gateway handling all signaling events
* Integration with mediasoup for WebRTC media handling
* Comprehensive room management
* Participant authorization and access control

### 6. zma-upload

**Purpose**: File upload management service.

**Key Features**:

* Secure file upload handling
* S3 integration for storage
* Upload request and completion workflows

**Communication Methods**:

* GraphQL API

**Notable Endpoints**:

* uploadServiceRequestUpload: Initiates the upload process
* uploadServiceCompleteUpload: Finalizes an upload

### 7. larva-course

**Purpose**: Course and learning management.

**Key Features**:

* Lesson management
* Topic organization
* User progress tracking
* Learning analytics

**Communication Methods**:

* GraphQL API

**Notable Components**:

* Lesson creation and management
* Topic organization
* User collections
* Sentence management for language learning

### 8. zma-fns

**Purpose**: Form and survey management service.

**Key Features**:

* Form creation and management
* Survey distribution
* Response collection and analysis

**Communication Methods**:

* GraphQL API

**Notable Endpoints**:

* Form and survey CRUD operations
* User response management

## Getting Started

1. **Environment Setup**

# Install required tools  
make prerequire  
make post-install  
  
# Environment variables setup  
cp .env.example .env  
# Edit .env with required values

1. **Starting the Development Environment**

# Start a specific service  
nx serve <service-name>  
  
# For example:  
nx serve zma-auth

1. **Running Tests**

# Run tests for a specific service  
nx test <service-name>  
  
# Run e2e tests  
nx e2e <service-name>

1. **Development Workflow**

* Use commitizen for standardized commits
* Run linting before commits
* Create changeset for version management

## Key Libraries and Dependencies

1. **@nestjs** modules:
   * @nestjs/common, @nestjs/core: Core NestJS framework
   * @nestjs/graphql: GraphQL integration
   * @nestjs/microservices: gRPC and Kafka integration
   * @nestjs/mongoose: MongoDB integration
   * @nestjs/websockets: WebSockets support
2. **GraphQL Ecosystem**:
   * @graphql-yoga/nestjs: Yoga GraphQL server integration
   * graphql-subscriptions: GraphQL subscription support
   * @escape.tech/graphql-armor: GraphQL security features
3. **Communication**:
   * @grpc/grpc-js: gRPC client/server
   * kafkajs: Kafka client
   * socket.io: WebSocket implementation
4. **WebRTC**:
   * mediasoup: WebRTC SFU for media handling
5. **Storage**:
   * @aws-sdk/client-s3: S3 storage client
6. **Utilities**:
   * lodash: Utility functions
   * dayjs: Date manipulation
   * winston: Logging

## Deployment Architecture

The services are containerized using Docker and deployed using Kubernetes (manifest files in .k8s directory). The deployment workflow is managed through CI/CD pipelines (GitHub Actions and Semaphore).

## Recommended Learning Path

1. **NestJS Fundamentals**
   * Official NestJS documentation
   * Module system and dependency injection
2. **GraphQL with NestJS**
   * Resolver pattern
   * Schema generation
   * Directives and middleware
3. **Microservices Communication**
   * gRPC concepts and protocol buffers
   * Kafka message patterns
   * WebSockets with Socket.IO
4. **WebRTC (for signaling service)**
   * WebRTC fundamentals
   * SDP negotiation
   * mediasoup architecture
5. **Nx Monorepo Management**
   * Workspace concepts
   * Code generation
   * Build and dependency management

## Common Development Tasks

### Adding a New Service

nx g @nx/nest:app new-service-name

### Creating a New Library

nx g @nx/js:lib new-lib-name

### Generating NestJS Components

nx g @nx/nest:controller new-controller --project=service-name  
nx g @nx/nest:service new-service --project=service-name

## Troubleshooting

1. **Service Communication Issues**
   * Check network configuration in application.yaml
   * Verify service discovery settings
   * Check Kafka broker and topic configuration
2. **GraphQL Schema Issues**
   * Review generated schemas in graphqls/ directory
   * Check resolver implementations
   * Validate types in TypeScript interfaces match GraphQL types
3. **Authentication Problems**
   * Verify JWT settings in AppConfigService
   * Check tenant resolution middleware
   * Validate guard implementations

## Additional Resources

* Project Wiki (internal)
* Architecture Decision Records (ADRs) in docs/
* NestJS official documentation
* Nx documentation

## Database Design

The project uses MongoDB as its primary database, with a multi-tenant design pattern where most collections include a tenantId field for data isolation between tenants.

### Core Data Models

#### Entity Definitions

The entity models are defined in the following locations:

1. **User Service**
   * libs/zma-types/src/services/models/user/user.model.ts: Contains user models and DTO definitions
   * apps/zma-user/src/core/types: Contains GraphQL input/output types
2. **Meeting Service**
   * libs/zma-types/src/services/models/meeting/room.model.ts: Contains room models
   * apps/zma-meeting/src/core/types: Contains meeting-specific types
3. **Dictionary Service**
   * apps/zma-dictionary/src/core/types: Contains dictionary and word models
4. **Upload Service**
   * libs/zma-types/src/services/models/upload/upload.model.ts: Contains upload models
   * apps/zma-upload/src/core/types: Contains upload request/response types
5. **Course Service (Larva Course)**
   * apps/larva-course/src/core/types: Contains topic, lesson, and user collection models
6. **Forms Service (ZMA-FNS)**
   * apps/zma-fns/src/core/types/graphql: Contains form and survey models

#### Repository Implementation

MongoDB repositories are implemented using generic patterns:

* libs/zma-repositories/src/lib/mongo-generic.repository.ts: Base repository for non-tenant resources
* libs/zma-repositories/src/lib/tenant-mongo-generic.repository.ts: Repository for tenant-aware resources

### Database Design Patterns

1. **Multi-tenancy**: Most collections include a tenantId field to isolate data between different tenants.
2. **Repository Pattern**: The application uses a repository pattern implemented through MongoGenericRepository and TenantMongoGenericRepository classes, providing standardized CRUD operations.
3. **Mongoose Integration**: MongoDB integration is facilitated through Mongoose ODM, with schemas defined in the service modules.
4. **Indexes**: Common indexes include:
   * tenantId (for tenant isolation)
   * createdAt (for time-based queries)
   * Compound indexes on frequently queried fields
5. **Schema Validation**: MongoDB schema validation rules are applied at the database level to ensure data integrity.

## 3rd Services:

### 1. OpenAI API

**Purpose**: Integration with OpenAI services for AI capabilities across the platform.

**Key Features**:

* Text generation and completion
* Translation services

**Communication Methods**:

* REST API integration via HTTP client
* Caching layer for optimizing token usage

### 2. R2 Storage

**Purpose**: Cloudflare R2 object storage integration for file storage needs.

**Key Features**:

* S3-compatible API integration
* Secure file storage and retrieval
* Configurable access policies
* CDN integration for fast content delivery
* Cost-effective storage solution

**Communication Methods**:

* S3 SDK integration
* Presigned URL generation for direct uploads