Handover Document for Clync App

Overview

This document provides an overview of the Project Structure, Codebase Walkthrough, Installation, and Setup to ensure smooth knowledge transfer and ongoing project maintainability.

Technologies and Infrastructure

1. **Backend Framework:** Node.js (Express)
2. **ORM:** Prisma ORM
3. **Database:** PostgreSQL
    a. **Database 1:** For Social and Admin services
    b. **Database 2:** For Financial service
4. **Storage:** Azure Blob Storage for media files
5. **API Gateway:** Node.js API Gateway for routing requests
6. **Container Orchestration:** Kubernetes for managing microservices
7. **Architecture:** Microservices

Microservices Overview

The application consists of four independent microservices:

1. **Social Service:** Handles social module functionalities.
2. **Financial Service:** Manages financial-related features (Planning Pockets).
3. **Flagship Service:** Integrates third-party (Flagship) financial APIs.
4. **Admin Service:** Manages the APIs for the admin portal.

Cross-Database Communication

Since separate databases are used for different modules, cross-database communication is facilitated via API calls. Data from different databases is retrieved through API endpoints, and results are joined and processed at the code level.

Project Directory Structure and Descriptions

api-gateway

- **Description:** Serves as the main entry point for the backend, acting as a reverse proxy and routing incoming requests to the appropriate backend services.
- **Functionality:**
    - Handles shared functionality like authentication, request validation, and rate limiting.

    - Improves scalability and security.

apps

- **Description:** Contains both frontend and backend services, each encapsulated in separate directories for modular development and deployment.

  – **frontend**

  - **Description:** The client-side application that interacts with backend APIs to provide a user interface.
  - **Structure:**
    - Organized with reusable components, utilities, and service files for organized development.

  – **backend**

  - **Description:** Houses individual backend services, organized as microservices for separation of concerns.
    - **social:**
      - **Functionality:** Manages social interactions (e.g., user connections, notifications, activity feeds).
    - **financial:**
      - **Functionality:** Handles financial features (e.g., planning pockets).
    - **admin:**
      - **Functionality:** Provides APIs for platform management, user oversight, and analytics.
    - **flagship:**
      - **Functionality:** Contains core backend functionalities for financial transactions by utilizing flagship APIs.

charts

- **Description:** Helm charts for Kubernetes deployment configurations, enabling the containerized deployment and scaling of services.
- **Structure:**
  - Each chart is specific to a module or service, with shared charts for common configurations.

Directory Structure and Description of a Backend Service

Root Directory

- **index.js:** The main entry point for the application. It initializes the server using configurations and imports core modules, starting the API services.

Environments

- **.env.development / .env.production:** Environment configuration files for different stages (development, production). These store environment variables like database URLs, API keys, and secrets.

Locales

- **ar/translation.json, en/translation.json:** Translation files for localization, allowing the app to support multiple languages (Arabic and English). These JSON files contain key-value pairs of strings in respective languages.

Logs

- **logs:** Directory where log files are stored, typically for monitoring application events, errors, and activity history.

Node Modules

- **node_modules:** Contains all installed npm packages and dependencies.

Prisma

- **migrations:** Stores SQL migration files that track schema changes for the Prisma ORM, allowing the database to be kept in sync with model definitions.
- **schema.prisma:** The main Prisma schema file where database models and relationships are defined. Used by Prisma to generate types and manage the database.

Source Directory (src)

The main source directory contains the core business logic, configurations, and modules.

Config

- **azureBlobStorage.js:** Configuration for Azure Blob Storage, handling media file storage and retrieval.
- **env_config.js:** Environment loader that validates and loads environment variables using dotenv and Joi.
- **firebase.js:** Initializes Firebase Admin SDK for managing authentication and other Firebase services.
- **logger.js:** Configures logging, usually with winston or morgan, to provide error and info logs.
- **morgan.js:** Sets up HTTP request logging with morgan middleware.
- **server.js:** Initializes the server and integrates Socket.IO, handling WebSocket connections.

Constants

- **index.js:** Contains constant values like enums, status codes, and fixed strings used throughout the application for consistency.

Controllers

Subdirectories for each API resource (e.g., auth, user, admin, chat, circle): Each folder contains controllers that handle HTTP requests and responses for their respective API resources.

Global

- **index.js:** Houses globally accessible variables, methods, or initializations used across the application.

Helper

Contains reusable modules to assist with common tasks:

- **azure.service.js:** Manages Azure Blob Storage operations (upload/download).
- **bcrypt.js:** Provides password hashing utilities.
- **cronJob.js:** Contains scheduled tasks or background jobs.
- **flagShip.js:** Handles interactions with Flagship, a third-party API.
- **generateInviteLink.js:** Creates dynamic invitation links.
- **jwtToken.js:** Manages JSON Web Token operations (creation/verification).
- **paginate.js:** Helper for pagination in responses.
- **prismaClient.js:** Prisma client initialization for database access.
- **push-notification.service.js:** Sends push notifications (e.g., FCM).
- **sms.service.js:** Manages SMS notifications.

Middlewares

- **auth.middleware.js:** Handles authentication and authorization.
- **error.middleware.js:** Handles global error processing.
- **i18n.js:** Manages internationalization (language support).
- **multer.middleware.js, multerMultifile.middleware.js:** Configure file upload using multer.
- **socket.middleware.js:** Middleware for processing data in Socket.IO connections.
- **validate.middleware.js:** Validates incoming request data, typically with Joi or a similar library.

Routes

- **v1/index.js:** Entry point for API routes (v1/index.js): Entry point for API routes (versioned). Calls each API module's route definition. Subdirectories for each API resource (e.g., auth, user, admin, chat, circle): Each contains route definitions for endpoints of their respective API. Routes map HTTP requests to controller methods.

Services

- **index.js:** General index for accessing all services in a centralized way. Subdirectories for each API resource (e.g., auth, user, admin, chat, circle): Services contain the business logic for each feature, implementing the core processing and database interactions required by controllers.

Utils

- **ApiError.js:** Custom error class for API-specific error handling.
- **catchAsync.js:** Utility function to handle async errors in routes.
- **generateOtp.js:** Helper for creating one-time passwords.
- **generateUniqueCode.js:** Generates unique codes (e.g., invite codes).
- **isPrismaError.js:** Utility to handle Prisma ORM-specific errors.
- **joi-common.js:** Common Joi validation schemas or helpers.
- **loadLanguageFile.js:** Loads i18n language files dynamically.
- **pick.js:** Helper to pick specific properties from an object.
- **prismaErrorMapper.js:** Maps Prisma errors to more user-friendly messages.

Validation

- **isPrismaError.js:** Repeated utility to check Prisma errors.
- **joi-common.js:** Defines reusable Joi validation schemas for data validation.

Detailed Flow of Execution

Here's an overview of the full flow, from server creation to handling a request through middleware and routing, and finally sending a response back to the client:

1. **Server Initialization (index.js):**
    a. The index.js file is the main entry point of the application, where the Express server is initialized.
    b. An Express instance is created, and basic configurations, such as environment settings and logger configuration, may be applied.
    c. The server listens on a specific port defined in the environment variables, and an initial success message is logged to confirm the server is running.
    d. The app instance from app.js is imported, which contains the main application configuration, middleware setup, and routing.
2. **Application Configuration and Middleware Setup (app.js):**
    a. In app.js, the application instance (app) is configured with a variety of middlewares that handle different aspects of request processing.
    b. Key middleware includes:
        i. Error Handling Middleware: Provides consistent error handling across the app.
        ii. Body Parser Middleware: Parses incoming request bodies for JSON and URL-encoded data, making it accessible as req.body.
        iii. Authentication Middleware: Checks if a valid token is present and attaches the authenticated user to req.user.

        iv. Validation Middleware: Validates incoming request data against predefined schemas, ensuring each request has the correct structure and content.

3. **Routing Setup (app.js and routes/index.js):**
    a. After applying middleware, the app is configured with routes, which handle specific API endpoints.
    b. The routes/index.js file is a centralized place to define route groups. For example, each route, like /user, has a dedicated file (e.g., user.routes.js) where all endpoints related to transactions are grouped.
    c. In index.js, each route group is mapped to a base path (e.g., /api/v1/user), so requests to specific paths will be directed to the relevant router file.
4. **Request Handling in Routes (-.routes.js):**
    a. When a request matches a defined route, it is forwarded to the specific routes.js file.
    b. Inside routes.js, each endpoint specifies:
        i. The HTTP method (e.g., POST).
        ii. Middleware for authentication and validation. For instance, verifyToken ensures the request is from an authenticated user.
        iii. Validation Middleware: This checks that the request body matches the defined schema before passing it on.
    c. After middleware processing, the request is directed to the corresponding controller function.
5. **Controller Processing (-.controller.js):**
    a. The controller layer handles the core logic of each request.
    b. For instance, a controller receives the request data, extracts relevant parameters, and calls the appropriate service function.
    c. The controller may perform additional data formatting or translation if required before passing the request to the service layer.
6. **Service Logic (-.service.js:**
    a. The service layer is where the business logic resides. Here, the service method performs complex operations. It may use external helper functions (e.g., fetchFinancialData) to fetch cross database (financial or social) data or perform database queries and performs the business logic.
    b. The service then returns the updated data or result to the controller.
7. **Sending the Response:**
    a. Once the service completes the request, it returns the result to the controller.
    b. The controller formats this result and sends it back to the client with a status code (e.g., 200 OK).

c. If an error occurs at any step, it's caught by error-handling middleware that sends a standardized error response to the client.

This process ensures a structured flow, with clear separation of concerns across server setup, middleware, routing, controllers, and services, resulting in a scalable, maintainable application.

Docker Image Creation and Deployment Using Helm

1. **Build the Docker Image**
   a. First, you need to create a Docker image from the Dockerfile that resides in your project directory.
   b. **Step 1.1:** Navigate to the directory where the Dockerfile is located.
   c. **Step 1.2:** Build the Docker image using the docker build command. docker build -t your-image-name:your-tag .
      i. **your-image-name**: A custom name for your Docker image.
      ii. **your-tag**: Version or tag for the image (e.g., v1.0, latest).
      iii. **.**: Refers to the current directory where the Dockerfile is located.
   d. **Step 1.3:** Verify the image creation by listing all Docker images.
      docker images
2. **Push the Image to a Docker Registry**
   a. To deploy your image using Helm, you need to push it to a Docker registry (such as Docker Hub, Azure Container Registry, AWS ECR, etc.).
   b. **Step 2.1:** Log in to your Docker registry (if required).
      docker login

      This command will prompt you for your registry credentials.
   c. **Step 2.2:** Tag the image to include your registry's URL.
      docker tag your-image-name:your-tag registry-url/your-image-name:tag
      i. **registry-url**: URL of the registry (e.g., docker.io for Docker Hub, or a custom registry URL).
      ii. **your-image-name:tag**: Image name and tag that you want to push.
   d. **Step 2.3:** Push the image to the registry.
      docker push registry-url/your-image-name:your-tag

   This uploads your Docker image to the specified registry.

3. **Create a Helm Chart for Deployment**
   a. Next, you need to set up a Helm chart for deploying your Docker image to Kubernetes.
   b. **Step 3.1:** If you don't already have a Helm chart, create a new chart.

```
helm create your-helm-chart
```

This creates a new directory called **your-helm-chart** with basic files and templates for a Kubernetes deployment.

c. **Step 3.2:** Modify the **values.yaml** file in the chart to specify your Docker image. In **your-helm-chart/values.yaml**, update the image section to reflect your image repository and tag.
   i. **repository**: The URL of your Docker registry (e.g., docker.io/your-image-name or a private registry URL).
   ii. **tag**: The version/tag of your image (e.g., latest or v1.0).

d. **Step 3.3:** You may also modify other configurations such as resources, environment variables, and ports as needed.

4. **Install/Upgrade the Helm Chart**
   a. Once your Helm chart is configured to use the correct image, you can deploy it to your Kubernetes cluster.
   b. **Step 4.1:** Install or upgrade the Helm release.
   ```
   helm upgrade --install your-release-name your-helm-chart/
   ```
      i. **your-release-name**: A name for your Helm release (e.g., clyncApp-release).
      ii. **your-helm-chart/**: The path to your Helm chart directory. This will deploy the Docker image to the Kubernetes cluster defined in your kubeconfig.
   c. **Step 4.2:** Verify the deployment.
   ```
   kubectl get pods
   ```

This will show the list of pods running in your Kubernetes cluster. You should see your application pod running.