# Workout Tracker Backend Overview

## Overview

The Workout Tracker backend is a robust architecture designed to support a comprehensive application for users looking to manage and track their fitness routines effectively. Built using **Node.js**, **Express**, and **MongoDB** (utilizing Mongoose), this backend serves as the backbone of the Workout Tracker, facilitating key functionalities that enhance user experience and engagement.

### Core Features

* **User Management:** At its core, the backend provides a user management system that handles authentication through **Clerk**. This ensures secure user registration and login processes while maintaining user privacy through a unique identifier (clerkUserId).
* **Workout & Exercise Tracking:** Users can easily log their workouts, categorizing entries by day. This feature allows seamless tracking of progress and facilitates the synchronization of calendar events to integrate workouts into daily routines.
* **Feedback Mechanism:** The solution incorporates a feedback system that empowers users to share their experiences and suggestions for improvement, which is vital for refining the application's offerings.
* **Notifications:** The backend implements a notification system that alerts users of important events, such as workout reminders, thereby enabling better adherence to fitness plans.
* **Payment Integration:** Payments are managed securely via **Stripe**, allowing users to process transactions smoothly while maintaining a record of their payment history within the application.
* **Google Ads Integration:** Although managed on the frontend, the backend supports **Google Ads** integration, providing monetization opportunities without complicating backend complexity.

This modular structure enhances maintainability and scalability, making it easy for developers to implement additional features and improvements over time.

## Technology Stack

The Workout Tracker backend is developed using a modern technology stack that consists of several key components:

* Node.js
* Express.js
* Mongo db

### Clerk for Authentication

Clerk is an advanced authentication service that simplifies user management by offering seamless registration and login processes. It handles user identity verification and uses secure tokens to ensure user data is protected.

### Stripe for Payment Processing

Stripe is a leading payment processing platform that allows businesses to manage online transactions securely. The integration of Stripe in the Workout Tracker backend enables the handling of payment transactions smoothly and reliably, providing users with an efficient way to pay for premium services.

## Folder Structure

*project-root/*

*│*

*├── package.json*

*├── server.js*

*├── config.js*

*├── app.js*

*│*

*├── config/*

*│ ├── aws-s3.js*

*│ ├── db.js*

*├── controllers/*

*│ ├── calendarController.js*

*│ ├── dayController.js*

*│ ├── entryController.js*

*│ ├── exerciseCatalogController.js*

*│ ├── feedbackController.js*

*│ ├── notificationController.js*

*│ ├── paymentController.js*

*│ ├── userController.js*

*│ ├── variationController.js*

*│ └── workoutController.js*

*│*

*├── models/*

*│ ├── CalendarEvent.js*

*│ ├── Day.js*

*│ ├── Entry.js*

*│ ├── ExerciseCatalog.js*

*│ ├── Feedback.js*

*│ ├── Notification.js*

*│ ├── Payment.js*

*│ ├── User.js*

*│ ├── Variation.js*

*│ └── Workout.js*

*│*

*├── routes/*

*│ ├── calendarRoutes.js*

*│ ├── dayRoutes.js*

*│ ├── entryRoutes.js*

*│ ├── exerciseCatalogRoutes.js*

*│ ├── feedbackRoutes.js*

*│ ├── notificationRoutes.js*

*│ ├── paymentRoutes.js*

*│ ├── userRoutes.js*

*│ ├── variationRoutes.js*

*│ └── workoutRoutes.js*

*│*

*├── middlewares/*

*│ └── authenticateUserByClerkId.js*

## Database Models

The database for the Workout Tracker backend comprises several key models defined using Mongoose, which establishes schema structures to manage various data types and relationships effectively.

### User Model

*const* mongoose = require("mongoose");

*const* UserSchema = new mongoose.Schema({

    clerkUserId: {

        type: String,

        required: true,

        unique: true

    },

    name: {

        type: String,

        required: true

    },

    username: {

        type: String,

        required: true

    },

    email: {

        type: String,

        required: true

    },

    image: {

        type: String

    },

    isVerified: {

        type: Boolean,

        default: false

    },

    membership: {

        type: String,

        enum: ["premium", "freemium"],

        default: "freemium"

    },

    createdAt: {

        type: Date,

        default: Date.now

    },

    updatedAt: {

        type: Date,

        default: Date.now

    }

});

*// Update `updatedAt` on save*

UserSchema.pre("save", *function* (*next*) {

    this.updatedAt = Date.now();

    next();

});

*module*.*exports* = mongoose.model("User", UserSchema);

### Calendar Event Model

*const* mongoose = require("mongoose");

*const* CalendarEventSchema = new mongoose.Schema({

    user: {

        type: mongoose.Schema.Types.ObjectId,

        ref: "User",

        required: true

    },

    googleEventId: {

        type: String,

        required: true

    },

    title: {

        type: String,

        required: true

    },

    start: {

        type: Date,

        required: true

    },

    end: {

        type: Date,

        required: true

    },

    description: {

        type: String

    }

});

*module*.*exports* = mongoose.model("CalendarEvent", CalendarEventSchema);

### Day Model

*const* mongoose = require("mongoose");

*const* DaySchema = new mongoose.Schema({

    user: {

        type: mongoose.Schema.Types.ObjectId,

        ref: "User",

        required: true

    },

    name: {

        type: String,

        required: true

    },

    entries: [{

        type: mongoose.Schema.Types.ObjectId,

        ref: "Entry"

    }],

    date: {

        type: Date,

        default: Date.now

    }

});

*module*.*exports* = mongoose.model("Day", DaySchema);

### Entry Model

*const* mongoose = require("mongoose");

*const* EntrySchema = new mongoose.Schema({

    user: {

        type: mongoose.Schema.Types.ObjectId,

        ref: "User",

        required: true

    },

    type: {

        type: String,

        enum: ["lift", "cardio"],

        required: true

    },

    bodyGroup: {

        type: String,

        required: true,

        enum: ["Arms", "Back", "Chest", "Legs", "Abs"]

    },

    muscleGroup: {

        type: String,

        required: true

    },

    workoutType: {

        type: String

    },

    variation: {

        type: mongoose.Schema.Types.ObjectId,

        ref: "Variation",

        required: true

    },

    machine: {

        type: mongoose.Schema.Types.ObjectId,

        ref: "Machine",

        required: true

    },

    limbOption: {

        type: String,

        enum: ["Single", "Both", "N/A"],

        default: "N/A"

    },

    notes: {

        type: String

    },

    sets: [{

        reps: {

            type: Number,

            required: true

        },

        weight: {

            type: Number,

            required: true

        }

    }],

    date: {

        type: Date,

        default: Date.now

    }

});

*module*.*exports* = mongoose.model("Entry", EntrySchema);

### Feedback Model

*const* mongoose = require("mongoose");

*const* FeedbackSchema = new mongoose.Schema({

    user: {

        type: mongoose.Schema.Types.ObjectId,

        ref: "User",

        required: true

    },

    userLikes: {

        type: String

    },

    wantImprovements: {

        type: String

    },

    rating: {

        type: Number,

        min: 1,

        max: 5

    },

    others: {

        type: String

    },

    createdAt: {

        type: Date,

        default: Date.now

    }

});

*module*.*exports* = mongoose.model("Feedback", FeedbackSchema);

### Variation Model

*const* mongoose = require("mongoose");

*const* VariationSchema = new mongoose.Schema({

    user: {

        type: mongoose.Schema.Types.ObjectId,

        ref: "User",

        required: true

    },

    name: {

        type: String,

        required: true

    },

    createdAt: {

        type: Date,

        default: Date.now

    }

});

*module*.*exports* = mongoose.model("Variation", VariationSchema);

### Workout Model

*const* mongoose = require("mongoose");

*const* WorkoutSchema = new mongoose.Schema({

    user: {

        type: mongoose.Schema.Types.ObjectId,

        ref: "User",

        required: true

    },

    name: {

        type: String,

        required: true

    },

    days: [{

        type: mongoose.Schema.Types.ObjectId,

        ref: "Day"

    }],

    createdAt: {

        type: Date,

        default: Date.now

    }

});

*module*.*exports* = mongoose.model("Workout", WorkoutSchema);

### Notification Model

*const* mongoose = require("mongoose");

*const* NotificationSchema = new mongoose.Schema({

    user: {

        type: mongoose.Schema.Types.ObjectId,

        ref: "User",

        required: true

    },

    title: {

        type: String,

        required: true

    },

    message: {

        type: String,

        required: true

    },

    read: {

        type: Boolean,

        default: false

    },

    createdAt: {

        type: Date,

        default: Date.now

    }

});

*module*.*exports* = mongoose.model("Notification", NotificationSchema);

### Payment Model

*const* mongoose = require("mongoose");

*const* PaymentSchema = new mongoose.Schema({

    user: {

        type: mongoose.Schema.Types.ObjectId,

        ref: "User",

        required: true

    },

    stripePaymentIntentId: {

        type: String,

        required: true,

        unique: true

    },

    amount: {

        type: Number,

        required: true

    },

    currency: {

        type: String,

        default: "usd"

    },

    status: {

        type: String,

        enum: ["succeeded", "pending", "failed", "canceled"],

        default: "pending"

    },

    createdAt: {

        type: Date,

        default: Date.now

    },

    updatedAt: {

        type: Date,

        default: Date.now

    }

});

*// Automatically update `updatedAt` on save*

PaymentSchema.pre("save", *function* (*next*) {

    this.updatedAt = Date.now();

    next();

});

*module*.*exports* = mongoose.model("Payment", PaymentSchema);

## API & Module Structure

The API for the Workout Tracker backend is organized into several modules, each corresponding to a core feature of the application. This modular organization not only simplifies the development process but also enhances maintainability and clarity.

✅ **User API** (Authentication via Clerk)

* **POST /register** — Create a new user account.
* **POST /login** — Authenticate users.
* **GET /profile** — Retrieve user profile.
* **PUT /profile** — Update user profile.
* **DELETE /** — Delete user account.

✅ **Calendar API**

* **GET /events** — Get all calendar events.
* **POST /events** — Create a new calendar event.
* **PUT /events/:id** — Update a calendar event.
* **DELETE /events/:id** — Delete a calendar event.

✅ **Day API**

* **GET /days** — Get all days.
* **POST /days** — Create a new day.
* **PUT /days/:id** — Update a day.
* **DELETE /days/:id** — Delete a day.

✅ **Entry API**

* **GET /entries** — Get all entries.
* **POST /entries** — Create a new workout entry.
* **PUT /entries/:id** — Update a workout entry.
* **DELETE /entries/:id** — Delete a workout entry.

✅ **Exercise Catalog API**

* **GET /exercises** — Get available exercises.
* **POST /exercises** — Create a new exercise.
* **PUT /exercises/:id** — Update an exercise.
* **DELETE /exercises/:id** — Delete an exercise.

✅ **Feedback API**

* **GET /feedback** — Get all feedback.
* **POST /feedback** — Submit user feedback.
* **PUT /feedback/:id** — Update feedback.
* **DELETE /feedback/:id** — Delete feedback.

✅ **Variation API**

* **GET /variations** — Get all exercise variations.
* **POST /variations** — Create a new variation.
* **PUT /variations/:id** — Update a variation.
* **DELETE /variations/:id** — Delete a variation.

✅ **Workout API**

* **GET /workouts** — Get all workouts.
* **POST /workouts** — Create a new workout routine.
* **PUT /workouts/:id** — Update a workout routine.
* **DELETE /workouts/:id** — Delete a workout routine.

✅ **Notification API**

* **GET /notifications** — Get all notifications.
* **POST /notifications** — Send a user notification.
* **DELETE /notifications/:id** — Delete a notification.

✅ **Payment API** (Stripe Integration)

* **POST /payments/create** — Create a payment session.
* **GET /payments/status/:id** — Check payment status.

### Middleware for Authentication

All endpoints, with exceptions for public routes (like calendar event retrieval), utilize the clerkAuth.js middleware to ensure that only authenticated users can access personal data. This secure implementation is crucial to protect user privacy and data integrity.

## Google Ads Integration

The integration of Google Ads within the Workout Tracker application is strategically implemented on the frontend. This approach maximizes efficiency by eliminating the need for any backend architecture to facilitate ad management. By opting for a client-side solution using Google AdSense, developers can seamlessly embed ads directly into the application UI.

### Performance Implications

This architecture choice minimizes backend load, allowing the server to focus on core functionalities such as user management, workout tracking, and payments. Since Google Ads are rendered on the client side, it enhances application responsiveness, ensuring a smoother user experience.

### Strategies for Success

To leverage Google Ads effectively, consider the following strategies:

* **Ad Placement:** Strategically position ads within the application's interface to ensure visibility without detracting from user experience.
* **Monitoring Performance:** Regularly analyze ad performance metrics through Google AdSense to optimize placements and improve revenue generation.
* **Responsive Design:** Ensure that ads are responsive, adapting to various screen sizes to engage a broader audience.

By prioritizing frontend integration, the Workout Tracker maintains a clean and efficient backend, essential for ongoing development and scalability.

## Additional Considerations

In the development of the Workout Tracker backend, several vital considerations significantly influence security, error handling, and project organization.

### Security Measures

Security is paramount, especially in applications dealing with sensitive user data. The backend implements **Clerk's authentication middleware** to secure endpoints, ensuring that all requests are verified and authorized. This setup effectively protects against unauthorized access to user-specific data and enforces strict user authentication prerequisites.

### Modularity Principle

The principle of modularity is central to the project's structure. By organizing the codebase into distinct folders for **controllers**, **models**, **routes**, and **middlewares**, the architecture promotes maintainability and scalability. Each module encapsulates specific functionalities, making it easy for developers to navigate, update, or expand the codebase. This modular design not only simplifies the initial development process but also paves the way for future enhancements and integrations, ensuring the application can evolve in response to user needs and technological advancements.

## Conclusion

This documentation provides a comprehensive overview of the Workout Tracker backend architecture, detailing its core features including user management, workout tracking, feedback collection, notifications, payments, and Google Ads integration. The architecture leverages a modern tech stack with Node.js, Express, and MongoDB, structured in a modular way to enhance maintainability and scalability. Each aspect, from database models to API organization, has been designed for clarity and efficiency. For any questions or further clarifications regarding the implementation, please feel free to reach out.