File: server/routes.ts

Type: ts | **Size:** 13500 bytes

```
import type { Express, Request, Response, NextFunction } from "express";
import { createServer, type Server } from "http";
import { storage } from "./storage";
import express from "express";
import { z } from "zod";
import OpenAI from "openai";
import Stripe from "stripe";
import {
  insertReadingSchema,
  insertDailyCardSchema,
  type SubscriptionPlan
} from "@shared/schema";
// Initialize Stripe
const stripe = new Stripe(process.env.STRIPE_SECRET_KEY!, {
  apiVersion: "2025-03-31.basil", // Using the latest available Stripe API version
});
// Initialize OpenAI
const openai = new OpenAI({
  apiKey: process.env.OPENAI_API_KEY || "sk-dummy-key-for-development"
});
export async function registerRoutes(app: Express): Promise {
  // API routes
  const apiRouter = express.Router();
  // Get oracle reading
  apiRouter.post("/readings", async (req, res) => {
    try {
      const { question, intention } = req.body;

      if (!question) {
        return res.status(400).json({ message: "Question is required" });
      }
      // Generate reading using OpenAI
      const response = await openai.chat.completions.create({
        // the newest OpenAI model is "gpt-4o" which was released May 13, 2024. do not change this unless ex
pl...
        model: "gpt-4o",
        messages: [
          {
            role: "system",
            content: `You are a mystical love oracle providing relationship advice. Create a 3-card reading
ba...
            Each card should have: title, position (Past/Present/Future), a short message, detailed explanat
io...
            Also provide a summary of the entire reading with advice. Format response as JSON.`
          },
          {
            role: "user",
            content: `My question is: "${question}". My intention is: "${intention || 'Seeking clarity'}"`
          }
        ],
        response_format: { type: "json_object" }
      });
      const readingData = JSON.parse(response.choices[0].message.content || '{}');

      // Process cards and ensure all required properties exist
      const cardsWithIds = (readingData.cards || []).map((card: any, index: number) => {
        // Ensure all required fields have values
        return {
          id: card.id || `card-${index}-${Date.now()}`,
          title: card.title || `Card ${index + 1}`,
          position: card.position || ["Past", "Present", "Future"][index % 3],
          icon: card.icon || "fa-star", // Default icon if not provided
          message: typeof card.message === 'string' ? card.message : 'Reflect on your feelings',
          details: typeof card.details === 'string' ? card.details : 'Take time to reflect on your relations
hi...
          color: card.color || "#8A2BE2" // Default color if not provided
        };
      });

      // Create a standardized reading result
      const reading = {
        question,
        intention: intention || "Seeking clarity",
```

```
        cards: cardsWithIds,
        summary: typeof readingData.summary === 'string' ? readingData.summary : 'The cards suggest focusing
 o...
        tags: readingData.tags || ["Relationship"],
        date: new Date()
      };
      // Save reading if user is logged in and userId is provided
      if (req.body.userId) {
        const insertData = {
          userId: req.body.userId,
          question,
          intention,
          cards: cardsWithIds,
          summary: typeof readingData.summary === 'string' ? readingData.summary : 'The cards suggest focusi
ng...
          tags: readingData.tags || ["Relationship"]
        };

        await storage.createReading(insertData);
      }
      res.json(reading);
    } catch (error) {
      console.error("Error generating reading:", error);
      res.status(500).json({ message: "Failed to generate reading" });
    }
  });
  // Get daily card
  apiRouter.post("/dailycard", async (req, res) => {
    try {
      const userId = req.body.userId;

      // Check if user already has a daily card for today
      if (userId) {
        const existingCard = await storage.getDailyCardForToday(userId);
        if (existingCard) {
          return res.json(existingCard);
        }
      }
      // Generate new daily card using OpenAI
      const response = await openai.chat.completions.create({
        // the newest OpenAI model is "gpt-4o" which was released May 13, 2024. do not change this unless ex
pl...
        model: "gpt-4o",
        messages: [
          {
            role: "system",
            content: `Create a daily oracle card with relationship advice. Include:
            1. A title (one or two words)
            2. A general message about today's energy
            3. A specific relationship focus
            4. An affirmation
            5. A relevant icon suggestion (Font Awesome icon name)
            Format response as JSON.`
          }
        ],
        response_format: { type: "json_object" }
      });
      const dailyCardData = JSON.parse(response.choices[0].message.content || '{}');

      // Create a standardized daily card result
      const dailyCard = {
        title: dailyCardData.title,
        date: new Date(),
        message: dailyCardData.message,
        relationshipFocus: dailyCardData.relationshipFocus,
        affirmation: dailyCardData.affirmation,
        icon: dailyCardData.icon || "fa-moon"
      };
      // Save daily card if user is logged in
      if (userId) {
        const insertData = {
          userId,
          title: dailyCardData.title,
          message: dailyCardData.message,
          relationshipFocus: dailyCardData.relationshipFocus,
          affirmation: dailyCardData.affirmation
```

```
      };

      await storage.createDailyCard(insertData);
    }
    res.json(dailyCard);
  } catch (error) {
    console.error("Error generating daily card:", error);
    res.status(500).json({ message: "Failed to generate daily card" });
  }
});
// Get user's reading history
apiRouter.get("/readings/:userId", async (req, res) => {
  try {
    const userId = parseInt(req.params.userId);
    const readings = await storage.getReadingsByUserId(userId);
    res.json(readings);
  } catch (error) {
    res.status(500).json({ message: "Failed to fetch readings" });
  }
});
// Get user's daily card history
apiRouter.get("/dailycards/:userId", async (req, res) => {
  try {
    const userId = parseInt(req.params.userId);
    const dailyCards = await storage.getDailyCardsByUserId(userId);
    res.json(dailyCards);
  } catch (error) {
    res.status(500).json({ message: "Failed to fetch daily cards" });
  }
});
// Define subscription plans
const subscriptionPlans: SubscriptionPlan[] = [
  {
    id: 'monthly',
    name: 'Monthly Premium',
    description: 'Unlock all premium features with our monthly subscription.',
    features: [
      'Unlimited personalized readings',
      'Access to all deep analysis tools',
      'Interactive "choose your own adventure" style readings',
      'Daily relationship insights and affirmations',
      'Ad-free experience'
    ],
    price: 9.99,
    interval: 'month',
    stripePriceId: 'price_monthly' // This would be replaced with a real Stripe price ID
  },
  {
    id: 'yearly',
    name: 'Yearly Premium',
    description: 'Save 16% with an annual subscription.',
    features: [
      'All monthly premium features',
      'Advanced relationship compatibility tests',
      'Premium card designs',
      'Priority access to new features',
      'Save 16% compared to monthly plan'
    ],
    price: 99.99,
    interval: 'year',
    stripePriceId: 'price_yearly' // This would be replaced with a real Stripe price ID
  }
];
// Get subscription plans
apiRouter.get("/subscription-plans", (req, res) => {
  res.json(subscriptionPlans);
});
// Check if user has premium access
apiRouter.get("/user-premium/:userId", async (req, res) => {
  try {
    const userId = parseInt(req.params.userId);
    const user = await storage.getUser(userId);

    if (!user) {
      return res.status(404).json({ message: "User not found" });
    }
```

```
    res.json({
      isPremium: user.isPremium || false,
      subscriptionStatus: user.subscriptionStatus || 'free',
      subscriptionExpiry: user.subscriptionExpiry || null
    });
  } catch (error) {
    res.status(500).json({ message: "Failed to fetch premium status" });
  }
});
// Create a Stripe payment intent for one-time payments
apiRouter.post("/create-payment-intent", async (req, res) => {
  try {
    const { amount } = req.body;

    const paymentIntent = await stripe.paymentIntents.create({
      amount: Math.round(amount * 100), // Convert to cents
      currency: "usd",
    });

    res.json({ clientSecret: paymentIntent.client_secret });
  } catch (error: any) {
    res.status(500).json({ message: "Error creating payment intent: " + error.message });
  }
});
// Create a Stripe subscription
apiRouter.post("/get-or-create-subscription", async (req, res) => {
  try {
    const { userId, planId, email } = req.body;

    if (!userId || !planId || !email) {
      return res.status(400).json({ message: "Missing required fields" });
    }

    const user = await storage.getUser(parseInt(userId));

    if (!user) {
      return res.status(404).json({ message: "User not found" });
    }

    // If user already has a subscription, retrieve it
    if (user.stripeSubscriptionId) {
      const subscription = await stripe.subscriptions.retrieve(user.stripeSubscriptionId);

      res.json({
        subscriptionId: subscription.id,
        clientSecret: (subscription.latest_invoice as any)?.payment_intent?.client_secret || null,
        status: subscription.status
      });

      return;
    }

    // Create a new customer if needed
    let customerId = user.stripeCustomerId;

    if (!customerId) {
      const customer = await stripe.customers.create({
        email: email,
        name: user.displayName || user.username,
      });

      customerId = customer.id;
      await storage.updateStripeCustomerId(user.id, customerId);
    }

    // Find the selected plan
    const plan = subscriptionPlans.find(p => p.id === planId);

    if (!plan || !plan.stripePriceId) {
      return res.status(400).json({ message: "Invalid plan selected" });
    }

    // Create the subscription
    const subscription = await stripe.subscriptions.create({
      customer: customerId,
```

```
        items: [{
          price: plan.stripePriceId,
        }],
        payment_behavior: 'default_incomplete',
        expand: ['latest_invoice.payment_intent'],
      });

      // Update user with subscription info
      await storage.updateUserStripeInfo(user.id, {
        customerId: customerId,
        subscriptionId: subscription.id
      });

      // Return the client secret for the payment
      res.json({
        subscriptionId: subscription.id,
        clientSecret: (subscription.latest_invoice as any)?.payment_intent?.client_secret || null,
        status: subscription.status
      });
    } catch (error: any) {
      res.status(500).json({ message: "Error creating subscription: " + error.message });
    }
  });
  // Webhook for Stripe events
  apiRouter.post("/webhook", express.raw({ type: 'application/json' }), async (req, res) => {
    let event;

    try {
      const signature = req.headers['stripe-signature']!;
      // In production, you would have a Stripe webhook secret
      // event = stripe.webhooks.constructEvent(req.body, signature, process.env.STRIPE_WEBHOOK_SECRET);

      // For development, we'll just parse the event
      event = JSON.parse(req.body.toString());
    } catch (err: any) {
      console.log(`■■ Webhook signature verification failed: ${err.message}`);
      return res.status(400).send(`Webhook Error: ${err.message}`);
    }

    // Handle the event
    switch (event.type) {
      case 'customer.subscription.created':
      case 'customer.subscription.updated':
        const subscription = event.data.object;

        // Find user by Stripe customer ID
        const users = Array.from(storage.getUsers().values());
        const user = users.find(u => u.stripeCustomerId === subscription.customer);

        if (user) {
          // Calculate expiry date
          const currentPeriodEnd = new Date(subscription.current_period_end * 1000);

          // Update user subscription status
          await storage.updateSubscriptionStatus(
            user.id,
            subscription.status,
            currentPeriodEnd
          );
        }

        break;
      default:
        console.log(`Unhandled event type ${event.type}`);
    }

    res.status(200).json({ received: true });
  });
  // Register API routes
  app.use("/api", apiRouter);
  const httpServer = createServer(app);
  return httpServer;
}
```