```
import { stripe } from '@/shared/utils/stripe/config';
import { supabaseAdmin } from '../supabase/admin';
import { User } from '@supabase/supabase-js';
import {
  attemptAutomaticCharge,
  findUnpaidInvoice,
  getLatestBillingTransaction,
} from './charge-customer';
import Stripe from 'stripe';
import { getUserStripeCustomerId } from '../stripe/server';
import { getOrganizationOwner } from './organization';
import { getMonthStartEndDates } from '../helpers';

export class BillingService {
  private userId: string;

  constructor(userId: string) {
    this.userId = userId;
  }

  async calculateTotalMonthlyUsage(month: Date): Promise<{
    status: number;
    message: string;
    user: { totalCost: number; id: string };
    organizations: {
      name?: string;
```

```
    organizationId: string;
    totalCost: number;
    ownerId: string;
  }[];
}> {
  try {
    const { start, end } = getMonthStartEndDates(month);

    // Get user activities (excluding organization activities)
    const { data: userActivities, error: userError } = await supabaseAdmin
      .from('swarms_cloud_api_activities')
      .select('invoice_total_cost')
      .eq('user_id', this.userId)
      .is('organization_id', null)
      .gte('created_at', start)
      .lte('created_at', end);

    if (userError) {
      console.error('Error fetching user activities:', userError);
      return {
        status: 500,
        message: 'Internal server error',
        user: { totalCost: 0, id: this.userId },
        organizations: [],
      };
    }
```

```
const userTotal = userActivities.reduce(
  (acc, item) => acc + (item.invoice_total_cost ?? 0),
  0,
);

// Get organization activities
const { data: organizationActivities, error: orgError } =
  await supabaseAdmin
    .from('swarms_cloud_api_activities')
    .select('invoice_total_cost, organization_id')
    .not('organization_id', 'is', null)
    .gte('created_at', start)
    .lte('created_at', end);

if (orgError) {
  console.error('Error fetching organization activities:', orgError);
  return {
    status: 500,
    message: 'Internal server error',
    user: { totalCost: Number(userTotal), id: this.userId },
    organizations: [],
  };
}

const organizations: {
```

```
  organizationId: string;
  totalCost: number;
  ownerId: string;
  name?: string;
}[] = [];

// Aggregate organization data
for (const activity of organizationActivities) {
  const organizationId = activity.organization_id;
  let existingOrg = organizations.find(
    (org) => org.organizationId === organizationId,
  );

  if (existingOrg) {
    existingOrg.totalCost += activity.invoice_total_cost || 0;
  } else {
    // Fetch owner ID for the organization
    const { data: ownerData, error: ownerError } = await supabaseAdmin
      .from('swarms_cloud_organizations')
      .select('owner_user_id, name')
      .eq('id', organizationId ?? '')
      .single();

    if (ownerError) {
      console.error('Error fetching owner ID:', ownerError);
      continue;
```

```
      }

      if (ownerData) {
        organizations.push({
          organizationId: organizationId ?? '',
          name: ownerData.name ?? '',
          totalCost: activity.invoice_total_cost || 0,
          ownerId: ownerData.owner_user_id ?? '',
        });
      } else {
        console.error('Organization not found for ID:', organizationId);
      }
    }
  }

  return {
    status: 200,
    message: 'Success',
    user: { totalCost: Number(userTotal), id: this.userId },
    organizations,
  };
} catch (error) {
  console.error('Error calculating total monthly usage:', error);
  return {
    status: 500,
    message: 'Internal server error',
```

```
      user: { totalCost: 0, id: this.userId },
      organizations: [],
    };
  }
}

async sendInvoiceToUser(
  totalAmount: number,
  user: User,
  message = 'Monthly API Usage billing',
): Promise<void> {
  if (totalAmount <= 0) return;

  if (!user) {
    throw new Error('User session not found');
  }

  try {
    const customerId = await getUserStripeCustomerId(user);

    if (!customerId) {
      throw new Error('Customer ID not found');
    }

    // Check for default payment method
    const customer = (await stripe.customers.retrieve(
```

```
    customerId,
  )) as Stripe.Customer;

  if (!customer || !customer.invoice_settings.default_payment_method) {
    console.error(
      'No default payment method found for customer:',
      customer.email,
    );
    throw new Error('No default payment method found for user');
  }

  const invoice = await stripe.invoices.create({
    customer: customerId,
    auto_advance: true,
    description: message,
    default_payment_method: customer.invoice_settings
      .default_payment_method as string,
    collection_method: 'charge_automatically', // Charge automatically
    // due_date: Math.floor(Date.now() / 1000) + 72 * 60 * 60, // Due date (72 hours from now)
  });

  let invoiceItem = await stripe.invoiceItems.create({
    customer: customerId,
    amount: Number(totalAmount) * 100,
    currency: 'usd',
    invoice: invoice.id,
```

```
  description: `Monthly API Usage billing for user ${user.email} with invoice ID ${invoice.id}`,
});

const captureResult = await stripe.invoices.pay(invoice.id);

// Check capture result for success
console.log('Invoice payment captured:', captureResult.paid);

const billingTransactions = await supabaseAdmin
  .from('swarm_cloud_billing_transcations')
  .insert([
    {
      user_id: user.id,
      total_montly_cost: parseFloat(totalAmount.toFixed(5)),
      stripe_customer_id: customerId,
      invoice_id: invoice.id,
      payment_successful: captureResult.paid,
    },
  ]);

if (billingTransactions.error) {
  console.error(
    'Error inserting billing transaction:',
    billingTransactions.error.message,
  );
  throw new Error('Could not insert billing transaction');
```

```
    }

    console.log('User successfully charged automatically');
  } catch (error) {
    console.error('Error charging user automatically:', error);
    throw new Error('Could not charge user automatically');
  }
}

async checkInvoicePaymentStatus(organizationPublicId?: string): Promise<{
  status: number;
  message: string;
  is_paid: boolean;
  unpaidInvoiceId?: string | null;
}> {
  try {
    let userId = this.userId;

    if (organizationPublicId) {
      // Fetch the organization owner's user ID
      const orgId = await getOrganizationOwner(organizationPublicId);
      if (!orgId)
        return {
          status: 500,
          message: 'Internal server error - invoice organization not found',
          is_paid: false,
```

```
      unpaidInvoiceId: null,
    };

  userId = orgId ?? '';
}

if (!userId) {
  return {
    status: 400,
    message: 'User session not found',
    is_paid: false,
    unpaidInvoiceId: null,
  };
}

const { status, message, transaction } =
  await getLatestBillingTransaction(userId);

if (status !== 200) {
  return {
    status,
    message,
    is_paid: false,
  };
}
```

```
// user might not be billed yet
if (!transaction) {
  return {
    status: 200,
    message: 'User has no billing transactions yet.',
    is_paid: true, // so consider user as paid if no transactions exist
    unpaidInvoiceId: null,
  };
}

const unpaidInvoice = await findUnpaidInvoice(transaction);

if (!unpaidInvoice) {
  return {
    status: 200,
    message: 'Success.',
    is_paid: true,
    unpaidInvoiceId: null,
  };
}

console.error('Found last unpaid invoice:', unpaidInvoice.id);

// Attempt automatic charge for unpaid invoice
const charged = await attemptAutomaticCharge(unpaidInvoice.id);
if (!charged) {
```

```
        console.error(
          'Failed to automatically charge for invoice:',
          unpaidInvoice.id,
        );
      }

      // Regardless of auto-charge success/failure, return unpaid invoice info
      return {
        status: 402,
        message: 'Unpaid invoice found. Automatic charges attempted.',
        is_paid: false,
        unpaidInvoiceId: unpaidInvoice.id,
      };
    } catch (error) {
      console.error('Error checking invoice payment status:', error);
      return {
        status: 500,
        message: `Internal server error ${error}`,
        is_paid: false,
        unpaidInvoiceId: null,
      };
    }
  }
}
```