```
import { toDateTime } from '@/shared/utils/helpers';
import { stripe } from '@/shared/utils/stripe/config';
import { createClient } from '@supabase/supabase-js';
import Stripe from 'stripe';
import type { Database, Tables, TablesInsert } from 'types_db';

type Product = Tables<'products'>;
type Price = Tables<'prices'>;

// Change to control trial period length
const TRIAL_PERIOD_DAYS = 0;

// Note: supabaseAdmin uses the SERVICE_ROLE_KEY which you must only use in a secure server-side context
// as it has admin privileges and overwrites RLS policies!

export const supabaseAdmin = createClient<Database>(
  process.env.NEXT_PUBLIC_SUPABASE_URL || '',
  process.env.SUPABASE_SERVICE_ROLE_KEY || '',
);

const upsertInvoiceRecord = async (invoice: Stripe.Invoice) => {
  const customerId = invoice.customer as string;
  const userId = await retrieveUserIdFromCustomerId(customerId);

  let reason: string | null = null;
```

```
// extract reason from metadata
try {
  if (invoice.metadata && invoice.metadata.reason) {
    reason = invoice.metadata.reason;
  }
} catch (e) {
  console.error(e);
}
const data: TablesInsert<'invoices'> = {
  id: invoice.id,
  created: toDateTime(invoice.created).toISOString(),
  stripe_customer_id: customerId,
  user_id: userId,
  is_paid: invoice.paid,
  metadata: invoice?.metadata ?? {},
  status: invoice.status,
  status_transitions: invoice.status_transitions as any,
  total: invoice.total,
  period_at: toDateTime(invoice.period_start).toISOString(),
  period_end: toDateTime(invoice.period_end).toISOString(),
  reason,
};

const { error: upsertError } = await supabaseAdmin
  .from('invoices')
  .upsert([data]);
```

```
  if (upsertError)
    throw new Error(`Invoice insert/update failed: ${upsertError.message}`);

  return true;
};

const upsertProductRecord = async (product: Stripe.Product) => {
  const productData: Product = {
    id: product.id,
    active: product.active,
    name: product.name,
    description: product.description ?? null,
    image: product.images?.[0] ?? null,
    metadata: product.metadata,
  };

  const { error: upsertError } = await supabaseAdmin
    .from('products')
    .upsert([productData]);
  if (upsertError)
    throw new Error(`Product insert/update failed: ${upsertError.message}`);
  console.log(`Product inserted/updated: ${product.id}`);
};

const upsertPriceRecord = async (
  price: Stripe.Price,
```

```
  retryCount = 0,
  maxRetries = 3,
) => {
  const priceData: Partial<Price> = {
    id: price.id,
    product_id: typeof price.product === 'string' ? price.product : '',
    active: price.active,
    currency: price.currency,
    type: price.type,
    unit_amount: price.unit_amount ?? null,
    interval: price.recurring?.interval ?? null,
    interval_count: price.recurring?.interval_count ?? null,
    trial_period_days: price.recurring?.trial_period_days ?? TRIAL_PERIOD_DAYS,
  };

  const { error: upsertError } = await supabaseAdmin
    .from('prices')
    .upsert([priceData as Price]);

  if (upsertError?.message.includes('foreign key constraint')) {
    if (retryCount < maxRetries) {
      console.log(`Retry attempt ${retryCount + 1} for price ID: ${price.id}`);
      await new Promise((resolve) => setTimeout(resolve, 2000));
      await upsertPriceRecord(price, retryCount + 1, maxRetries);
    } else {
      throw new Error(
```

```
        `Price insert/update failed after ${maxRetries} retries: ${upsertError.message}`,
      );
    }
  } else if (upsertError) {
    throw new Error(`Price insert/update failed: ${upsertError.message}`);
  } else {
    console.log(`Price inserted/updated: ${price.id}`);
  }
};

const deleteProductRecord = async (product: Stripe.Product) => {
  const { error: deletionError } = await supabaseAdmin
    .from('products')
    .delete()
    .eq('id', product.id);
  if (deletionError)
    throw new Error(`Product deletion failed: ${deletionError.message}`);
  console.log(`Product deleted: ${product.id}`);
};

const deletePriceRecord = async (price: Stripe.Price) => {
  const { error: deletionError } = await supabaseAdmin
    .from('prices')
    .delete()
    .eq('id', price.id);
  if (deletionError)
```

```
    throw new Error(`Price deletion failed: ${deletionError.message}`);
  console.log(`Price deleted: ${price.id}`);
};

const upsertCustomerToSupabase = async (uuid: string, customerId: string) => {
  const { error: upsertError } = await supabaseAdmin
    .from('customers')
    .upsert([{ id: uuid, stripe_customer_id: customerId }]);

  if (upsertError)
    throw new Error(
      `Supabase customer record creation failed: ${upsertError.message}`,
    );

  return customerId;
};

const createCustomerInStripe = async (uuid: string, email: string) => {
  // first check if the customer already exists in stripe
  const { data: existingStripeCustomer } = await stripe.customers.list({
    email,
  });
  if (existingStripeCustomer.length > 0 && !existingStripeCustomer[0].deleted) {
    return existingStripeCustomer[0].id;
  }
  const customerData = { metadata: { supabaseUUID: uuid }, email: email };
```

```
  const newCustomer = await stripe.customers.create(customerData);
  if (!newCustomer) throw new Error('Stripe customer creation failed.');

  return newCustomer.id;
};

const retrieveUserIdFromCustomerId = async (customerId: string) => {
  const { data: customerData, error: noCustomerError } = await supabaseAdmin
    .from('customers')
    .select('id')
    .eq('stripe_customer_id', customerId)
    .single();

  if (noCustomerError)
    throw new Error(`Customer lookup failed: ${noCustomerError.message}`);

  return customerData.id;
};
const retrieveUserStripeCustomerId = async (uuid: string) => {
  // Check if the customer already exists in Supabase
  const { data: existingSupabaseCustomer, error: queryError } =
    await supabaseAdmin
      .from('customers')
      .select('*')
      .eq('id', uuid)
      .maybeSingle();
```

```
  if (queryError) {
    throw new Error(`Supabase customer lookup failed: ${queryError.message}`);
  }

  if (existingSupabaseCustomer?.stripe_customer_id) {
    return existingSupabaseCustomer.stripe_customer_id;
  } else {
    throw new Error('No stripe customer id found');
  }
};
const createOrRetrieveStripeCustomer = async ({
  email,
  uuid,
}: {
  email: string;
  uuid: string;
}) => {
  // Check if the customer already exists in Supabase
  const { data: existingSupabaseCustomer, error: queryError } =
    await supabaseAdmin
      .from('customers')
      .select('*')
      .eq('id', uuid)
      .maybeSingle();
```

```
if (queryError) {
  throw new Error(`Supabase customer lookup failed: ${queryError.message}`);
}

// Retrieve the Stripe customer ID using the Supabase customer ID, with email fallback
let stripeCustomerId: string | undefined;
if (existingSupabaseCustomer?.stripe_customer_id) {
  try {
    const existingStripeCustomer = await stripe.customers.retrieve(
      existingSupabaseCustomer.stripe_customer_id,
    );
    if (!existingStripeCustomer.deleted) {
      stripeCustomerId = existingStripeCustomer?.id;
    }
  } catch (e) {
    console.error(
      `Failed to retrieve Stripe customer with ID: ${existingSupabaseCustomer.stripe_customer_id}`,
    );
  }
} else {
  // If Stripe ID is missing from Supabase, try to retrieve Stripe customer ID by email
  const stripeCustomers = await stripe.customers.list({ email: email });

  stripeCustomerId =
    stripeCustomers.data.length > 0 ? stripeCustomers.data[0].id : undefined;
}
```

```
// If still no stripeCustomerId, create a new customer in Stripe
const stripeIdToInsert = stripeCustomerId
  ? stripeCustomerId
  : await createCustomerInStripe(uuid, email);
if (!stripeIdToInsert) throw new Error('Stripe customer creation failed.');

if (existingSupabaseCustomer && stripeCustomerId) {
  // If Supabase has a record but doesn't match Stripe, update Supabase record
  if (existingSupabaseCustomer.stripe_customer_id !== stripeCustomerId) {
    const { error: updateError } = await supabaseAdmin
      .from('customers')
      .update({ stripe_customer_id: stripeCustomerId })
      .eq('id', uuid);

    if (updateError)
      throw new Error(
        `Supabase customer record update failed: ${updateError.message}`,
      );
    console.warn(
      `Supabase customer record mismatched Stripe ID. Supabase record updated.`,
    );
  }
  // If Supabase has a record and matches Stripe, return Stripe customer ID
  return stripeCustomerId;
} else {
```

```
    console.warn(
      `Supabase customer record was missing. A new record was created.`,
    );

    // If Supabase has no record, create a new record and return Stripe customer ID
    const upsertedStripeCustomer = await upsertCustomerToSupabase(
      uuid,
      stripeIdToInsert,
    );
    if (!upsertedStripeCustomer)
      throw new Error('Supabase customer record creation failed.');

    return upsertedStripeCustomer;
  }
};

/**
 * Copies the billing details from the payment method to the customer object.
 */
const copyBillingDetailsToCustomer = async (
  uuid: string,
  payment_method: Stripe.PaymentMethod,
) => {
  //Todo: check this assertion
  const customer = payment_method.customer as string;
  const { name, phone, address } = payment_method.billing_details;
```

```
  if (!name || !phone || !address) return;
  //@ts-ignore
  await stripe.customers.update(customer, { name, phone, address });
  const { error: updateError } = await supabaseAdmin
    .from('users')
    .update({
      billing_address: { ...address },
      payment_method: { ...payment_method[payment_method.type] },
    })
    .eq('id', uuid);
  if (updateError)
    throw new Error(`Customer update failed: ${updateError.message}`);
};

const manageSubscriptionStatusChange = async (
  subscriptionId: string,
  customerId: string,
  createAction = false,
) => {
  // Get customer's UUID from mapping table.
  const { data: customerData, error: noCustomerError } = await supabaseAdmin
    .from('customers')
    .select('id')
    .eq('stripe_customer_id', customerId)
    .single();
```

```
if (noCustomerError)
  throw new Error(`Customer lookup failed: ${noCustomerError.message}`);

const { id: uuid } = customerData!;

const subscription = await stripe.subscriptions.retrieve(subscriptionId, {
  expand: ['default_payment_method'],
});
// Upsert the latest status of the subscription object.
const subscriptionData: TablesInsert<'subscriptions'> = {
  id: subscription.id,
  user_id: uuid,
  metadata: subscription.metadata,
  status: subscription.status,
  price_id: subscription.items.data[0].price.id,
  //TODO check quantity on subscription
  // @ts-ignore
  quantity: subscription.quantity,
  cancel_at_period_end: subscription.cancel_at_period_end,
  cancel_at: subscription.cancel_at
    ? toDateTime(subscription.cancel_at).toISOString()
    : null,
  canceled_at: subscription.canceled_at
    ? toDateTime(subscription.canceled_at).toISOString()
    : null,
  current_period_start: toDateTime(
```

```
    subscription.current_period_start,
  ).toISOString(),
  current_period_end: toDateTime(
    subscription.current_period_end,
  ).toISOString(),
  created: toDateTime(subscription.created).toISOString(),
  ended_at: subscription.ended_at
    ? toDateTime(subscription.ended_at).toISOString()
    : null,
  trial_start: subscription.trial_start
    ? toDateTime(subscription.trial_start).toISOString()
    : null,
  trial_end: subscription.trial_end
    ? toDateTime(subscription.trial_end).toISOString()
    : null,
};

const { error: upsertError } = await supabaseAdmin
  .from('subscriptions')
  .upsert([subscriptionData]);
if (upsertError)
  throw new Error(
    `Subscription insert/update failed: ${upsertError.message}`,
  );
console.log(
  `Inserted/updated subscription [${subscription.id}] for user [${uuid}]`,
```

```
  );

  // For a new subscription copy the billing details to the customer object.
  // NOTE: This is a costly operation and should happen at the very end.
  if (createAction && subscription.default_payment_method && uuid)
    //@ts-ignore
    await copyBillingDetailsToCustomer(
      uuid,
      subscription.default_payment_method as Stripe.PaymentMethod,
    );
};

const increaseUserCredit = async (uuid: string, amount: number) => {
  const { credit: currentCredit, credit_count } = await getUserCredit(uuid);
  // Increase credit amount
  const newCredit = currentCredit + amount;

  // Increase credit count
  const newCreditCount = credit_count + 1;

  // Perform upsert operation
  const response = await supabaseAdmin
    .from('swarms_cloud_users_credits')
    .upsert(
      {
        user_id: uuid,
```

```
        credit: newCredit,
        credit_count: newCreditCount,
      },
      {
        onConflict: 'user_id',
      },
    );

  if (response.error) {
    throw new Error(response.error.message);
  } else {
    console.log('Upsert operation successful');
    return true;
  }
};

const getUserCreditPlan = async (uuid: string) => {
  const { data, error } = await supabaseAdmin
    .from('users')
    .select('credit_plan')
    .eq('id', uuid)
    .single();

  if (error) {
    console.error(`Failed to fetch user credit plan: ${error.message}`);
    throw new Error(`Failed to fetch user credit plan: ${error.message}`);
```

```
  }

  return data.credit_plan ?? 'default';
};

const getUserCredit = async (uuid: string) => {
  const { data, error } = await supabaseAdmin
    .from('swarms_cloud_users_credits')
    .select('credit, free_credit, credit_count')
    .eq('user_id', uuid)
    .single();
  if (error) {
    console.error(error.message);
    throw new Error(`Failed to fetch user credit: ${error.message}`);
  }
  return {
    credit: data?.credit ?? 0,
    free_credit: data?.free_credit ?? 0,
    credit_count: data?.credit_count ?? 0,
  };
};

const getStripeCustomerId = async (userId: string): Promise<string | null> => {
  const { data, error } = await supabaseAdmin
    .from('customers')
    .select('stripe_customer_id')
```

```
    .eq('id', userId)
    .single(); // Using .single() as we expect only one record for each user

  if (error) {
    console.error('Error fetching Stripe customer ID:', error);
    throw new Error('Failed to fetch Stripe customer ID');
  }

  return data ? data.stripe_customer_id : null;
};

export {
  getUserCredit,
  getUserCreditPlan,
  increaseUserCredit,
  upsertProductRecord,
  upsertPriceRecord,
  deleteProductRecord,
  deletePriceRecord,
  createOrRetrieveStripeCustomer,
  manageSubscriptionStatusChange,
  retrieveUserStripeCustomerId,
  upsertInvoiceRecord,
  getStripeCustomerId,
  retrieveUserIdFromCustomerId,
};
```