```
import { router, userProcedure } from '@/app/api/trpc/trpc-router';
import { PLATFORM } from '@/shared/constants/links';
import { getURL } from '@/shared/utils/helpers';
import { createPaymentSession } from '@/shared/utils/stripe/client';
import { stripe } from '@/shared/utils/stripe/config';
import {
  addPaymentMethodIfNotExists,
  checkoutWithStripe,
  createStripePortal,
  getSubscriptionStatus,
  getUserStripeCustomerId,
} from '@/shared/utils/stripe/server';
import { createOrRetrieveStripeCustomer } from '@/shared/utils/supabase/admin';
import { User } from '@supabase/supabase-js';
import { TRPCError } from '@trpc/server';
import Stripe from 'stripe';
import { z } from 'zod';

const paymentRouter = router({
  // payment
  createStripePaymentSession: userProcedure.mutation(async ({ ctx }) => {
    const user = ctx.session.data.session?.user as User;
    const customer = await getUserStripeCustomerId(user);
    if (!customer) {
      throw new TRPCError({
        code: 'INTERNAL_SERVER_ERROR',
        message: 'Error while creating stripe customer',
      });
    }

    const stripeSession = await createPaymentSession(user.id);
    return stripeSession.url;
  }),
  createSubscriptionCheckoutSession: userProcedure.mutation(async ({ ctx }) => {
    const user = ctx.session.data.session?.user as User;
    const stripe_product_id = process.env
      .NEXT_PUBLIC_STRIPE_SUBSCRIPTION_PRODUCT_ID as string;

    if (!stripe_product_id) {
      throw new TRPCError({
        code: 'INTERNAL_SERVER_ERROR',
        message: 'Stripe product id not found',
      });
    }

    const res = await ctx.supabase
      .from('prices')
      .select('*')
      .eq('id', stripe_product_id)
      .eq('type', 'recurring')
      .single();
```

```
if (res.error) {
  throw new TRPCError({
    code: 'INTERNAL_SERVER_ERROR',
    message: 'Error while getting stripe price',
  });
}
const productRow = res.data;
if (!productRow) {
  throw new TRPCError({
    code: 'INTERNAL_SERVER_ERROR',
    message: 'Stripe price not found',
  });
}

const { errorRedirect, sessionId } = await checkoutWithStripe(
  productRow,
  PLATFORM.ACCOUNT,
);
if (errorRedirect) {
  throw new TRPCError({
    code: 'INTERNAL_SERVER_ERROR',
    message: errorRedirect,
  });
} else {
  return sessionId as string;
}
```

```
}),
createStripePortalLink: userProcedure.mutation(async ({ ctx }) => {
  const user = ctx.session.data.session?.user as User;
  const url = await createStripePortal(
    user,
    `${getURL()}${PLATFORM.ACCOUNT}`,
  );
  return url;
}),
getSubscriptionStatus: userProcedure.query(async ({ ctx }) => {
  const user = ctx.session.data.session?.user as User;
  return await getSubscriptionStatus(user);
}),
//
getUserPaymentMethods: userProcedure.query(async ({ ctx }) => {
  const user = ctx.session.data.session?.user as User;
  const stripeCustomerId = await getUserStripeCustomerId(user);
  if (!stripeCustomerId) {
    throw new TRPCError({
      code: 'INTERNAL_SERVER_ERROR',
      message: 'Error while creating stripe customer',
    });
  }
  // get the cards
  const cards = await stripe.paymentMethods.list({
    customer: stripeCustomerId,
    type: 'card',
  });
  return cards.data;
}),
attachPaymentMethod: userProcedure
  .input(
    z.object({
      payment_method_id: z.string(),
    }),
  )
  .mutation(async ({ ctx, input }) => {
    const user = ctx.session.data.session?.user as User;
    const stripeCustomerId = await createOrRetrieveStripeCustomer({
      email: user.email ?? '',
      uuid: user.id,
    });
    if (!stripeCustomerId) {
      throw new TRPCError({
        code: 'INTERNAL_SERVER_ERROR',
        message: 'Error while creating stripe customer',
      });
    }
    try {
      const paymentMethod = await addPaymentMethodIfNotExists(
        stripeCustomerId,
        input.payment_method_id,
      );
      if (!paymentMethod) {
        throw new TRPCError({
          code: 'INTERNAL_SERVER_ERROR',
          message: 'Error while attaching payment method',
        });
      }
      return paymentMethod;
    } catch (error) {
      throw new TRPCError({
        code: 'INTERNAL_SERVER_ERROR',
        message: error as string,
      });
    }
  }),
detachPaymentMethod: userProcedure
  .input(
    z.object({
      payment_method_id: z.string(),
    }),
  )
  .mutation(async ({ ctx, input }) => {
    const user = ctx.session.data.session?.user as User;
    const stripeCustomerId = await getUserStripeCustomerId(user);
    if (!stripeCustomerId) {
      throw new TRPCError({
```

```
        code: 'INTERNAL_SERVER_ERROR',
        message: 'Error while creating stripe customer',
      });
    }
    const paymentMethod = await stripe.paymentMethods.detach(
      input.payment_method_id,
    );
    if (!paymentMethod) {
      throw new TRPCError({
        code: 'INTERNAL_SERVER_ERROR',
        message: 'Error while detaching payment method',
      });
    }
    return paymentMethod;
  }),
setDefaultPaymentMethod: userProcedure
  .input(
    z.object({
      payment_method_id: z.string(),
    }),
  )
  .mutation(async ({ ctx, input }) => {
    const user = ctx.session.data.session?.user as User;
    const stripeCustomerId = await getUserStripeCustomerId(user);
    if (!stripeCustomerId) {
      throw new TRPCError({
        code: 'INTERNAL_SERVER_ERROR',
        message: 'Error while creating stripe customer',
      });
    }
    const customer = await stripe.customers.update(stripeCustomerId, {
      invoice_settings: {
        default_payment_method: input.payment_method_id,
      },
    });
    if (!customer) {
      throw new TRPCError({
        code: 'INTERNAL_SERVER_ERROR',
        message: 'Error while setting default payment method',
      });
    }
    return customer;
  }),
getDefaultPaymentMethod: userProcedure.query(async ({ ctx }) => {
  const user = ctx.session.data.session?.user as User;
  const stripeCustomerId = await getUserStripeCustomerId(user);
  if (!stripeCustomerId) {
    throw new TRPCError({
      code: 'INTERNAL_SERVER_ERROR',
      message: 'Error while creating stripe customer',
    });
  }
```

```
    const customer = (await stripe.customers.retrieve(
      stripeCustomerId,
    )) as Stripe.Customer;
    if (!customer) {
      throw new TRPCError({
        code: 'INTERNAL_SERVER_ERROR',
        message: 'Error while getting default payment method',
      });
    }
    return customer.invoice_settings.default_payment_method;
  }),
});

export default paymentRouter;
```