```
import { router, userProcedure } from '@/app/api/trpc/trpc-router';
import { z } from 'zod';
import { TRPCError } from '@trpc/server';
import { User } from '@supabase/supabase-js';
import {
  createOrRetrieveStripeCustomer,
  getUserCredit,
} from '@/shared/utils/supabase/admin';
import { stripe } from '@/shared/utils/stripe/config';
import Stripe from 'stripe';
import { Tables } from '@/types_db';
import { getOrganizationUsage, userAPICluster } from '@/shared/utils/api/usage';
import { isEmpty } from '@/shared/utils/helpers';

const panelRouter = router({
  getUserCredit: userProcedure.query(async ({ ctx }) => {
    const user = ctx.session.data.session?.user as User;
    const { credit, free_credit } = await getUserCredit(user.id);
    return credit + free_credit;
  }),
  getUserCreditPlan: userProcedure.query(async ({ ctx }) => {
    const user = ctx.session.data.session?.user as User;
    const { data, error } = await ctx.supabase
      .from('users')
      .select('credit_plan')
      .eq('id', user.id)
```

```
      .single();

    if (error) {
      throw new TRPCError({
        code: 'INTERNAL_SERVER_ERROR',
        message: 'Error while fetching user credit plan',
      });
    }

    return data;
  }),
  updateAgent: userProcedure
  .input(
    z.object({
      agent_id: z.string().uuid(),
      name: z.string(),
      description: z.string(),
      system_prompt: z.string(),
      llm: z.string(),
    }),
  )
  .mutation(async ({ ctx, input }) => {
    const user_id = ctx.session.data.session?.user?.id || '';

    const { data, error } = await ctx.supabase
      .from('swarms_spreadsheet_session_agents')
```

```
      .update({
        name: input.name,
        description: input.description,
        system_prompt: input.system_prompt,
        llm: input.llm,
      })
      .eq('id', input.agent_id)
      .eq('user_id', user_id)
      .select()
      .single();

    if (error) {
      throw new TRPCError({
        code: 'INTERNAL_SERVER_ERROR',
        message: 'Error while updating agent',
      });
    }

    return data;
  }),
  getUserFreeCredits: userProcedure.query(async ({ ctx }) => {
    const user = ctx.session.data.session?.user as User;
    const { data, error } = await ctx.supabase
      .from('users')
      .select('had_free_credits')
      .eq('id', user.id)
```

```
  .single();

if (error) {
  throw new TRPCError({
    code: 'INTERNAL_SERVER_ERROR',
    message: 'Error while fetching users',
  });
}

if (!data?.had_free_credits) {
  console.log('User has no free credits yet');
  return;
}

const { data: credits, error: creditError } = await ctx.supabase
  .from('swarms_cloud_users_credits')
  .select('free_credit,credit_grant')
  .eq('user_id', user.id)
  .single();

if (creditError) {
  throw new TRPCError({
    code: 'INTERNAL_SERVER_ERROR',
    message: 'Error while fetching user free credits',
  });
}
```

```
  return { freeCredit: credits.free_credit, grant: credits.credit_grant };
}),
updateUserCreditPlan: userProcedure
  .input(z.object({ credit_plan: z.string() }))
  .mutation(async ({ ctx, input }) => {
    const user = ctx.session.data.session?.user as User;

    const userCredit = await getUserCredit(user.id);

    if (input.credit_plan === 'invoice' && userCredit.credit_count < 3) {
      throw new TRPCError({
        code: 'BAD_REQUEST',
        message:
          'You need at least three(3) manual credit payments before switching to invoice plan',
      });
    }

    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',
  });
}
const paymentMethods = await stripe.paymentMethods.list({
  customer: stripeCustomerId,
  type: 'card',
});

if (!paymentMethods.data.length) {
  throw new TRPCError({
    code: 'NOT_FOUND',
    message:
      'Please add a payment method in the "Manage Cards" section to switch to invoice plan',
  });
}

const customer = (await stripe.customers.retrieve(
  stripeCustomerId,
)) as Stripe.Customer;
if (!customer || !customer.invoice_settings.default_payment_method) {
  throw new TRPCError({
    code: 'INTERNAL_SERVER_ERROR',
    message:
      'No default payment method found. Click on added card to set as default',
  });
}
```

```
    const credits = await ctx.supabase
      .from('users')
      .update({
        credit_plan: input.credit_plan as Tables<'users'>['credit_plan'],
      })
      .eq('id', user.id);

    if (credits.error) {
      throw new TRPCError({
        code: 'INTERNAL_SERVER_ERROR',
        message: 'Error while updating user credit plan',
      });
    }
    return true;
  }),
// onboarding
getOnboarding: userProcedure.query(async ({ ctx }) => {
  const user = ctx.session.data.session?.user as User;
  const userOnboarding = await ctx.supabase
    .from('users')
    .select('*')
    .eq('id', user.id)
    .single();
  if (userOnboarding.error) {
    throw new TRPCError({
```

```
      code: 'INTERNAL_SERVER_ERROR',
      message: 'Error while fetching user onboarding status',
    });
  }
  return {
    basic_onboarding_completed:
      userOnboarding.data.basic_onboarding_completed,
    full_name: userOnboarding.data.full_name,
  };
}),
updateOnboarding: userProcedure
  .input(
    z.object({
      full_name: z.string().optional(),
      company_name: z.string().optional(),
      job_title: z.string().optional(),
      country_code: z.string().optional(),
      basic_onboarding_completed: z.boolean(),
      referral: z.string().optional(),
      signup_reason: z.string().optional(),
      about_company: z.string().optional(),
    }),
  )
  .mutation(async ({ ctx, input }) => {
    const user = ctx.session.data.session?.user as User;
    const updatedOnboarding = await ctx.supabase
```

```
      .from('users')
      .update(input)
      .eq('id', user.id);
    if (updatedOnboarding.error) {
      throw new TRPCError({
        code: 'INTERNAL_SERVER_ERROR',
        message: 'Error while updating user onboarding status',
      });
    }
    return true;
  }),
getUsageAPICluster: userProcedure
  .input(
    z.object({
      month: z.date(),
    }),
  )
  .mutation(async ({ ctx, input: { month } }) => {
    const user = ctx.session.data.session?.user as User;

    const cluster = await userAPICluster(user.id, month);

    if (cluster.status !== 200) {
      throw new Error(cluster.message);
    }
```

```
    if (isEmpty(cluster.user)) {
      return null;
    }

    return cluster.user;
  }),
getOrganizationUsage: userProcedure
  .input(
    z.object({
      month: z.date(),
    }),
  )
  .mutation(async ({ ctx, input: { month } }) => {
    const user = ctx.session.data.session?.user as User;

    const usage = await getOrganizationUsage(user.id, month);

    if (usage.status !== 200) {
      throw new Error(usage.message);
    }

    if (isEmpty(usage.organization)) {
      return null;
    }

    return usage.organization;
```

```
  }),

// SPREADSHEET SWARMS

createSession: userProcedure
  .input(
    z.object({
      task: z.string().optional(),
      output: z.any().optional(),
      tasks_executed: z.number().optional(),
      time_saved: z.number().optional(),
    }),
  )
  .mutation(async ({ ctx, input }) => {
    const user_id = ctx.session.data.session?.user?.id || '';

    // Set all other sessions to non-current
    await ctx.supabase
      .from('swarms_spreadsheet_sessions')
      .update({ current: false })
      .eq('user_id', user_id);

    const { data, error } = await ctx.supabase
      .from('swarms_spreadsheet_sessions')
      .insert({
        user_id,
```

```
        task: input.task,
        current: true,
        output: input.output ?? {},
        tasks_executed: input.tasks_executed ?? 0,
        time_saved: input.time_saved ?? 0,
      })
      .select()
      .single();

    if (error) throw error;
    return data;
  }),

getSessionWithAgents: userProcedure
  .input(
    z.object({
      session_id: z.string().uuid(),
    }),
  )
  .query(async ({ ctx, input }) => {
    const user_id = ctx.session.data.session?.user?.id || '';

    const { data: session, error: sessionError } = await ctx.supabase
      .from('swarms_spreadsheet_sessions')
      .select('*')
      .eq('id', input.session_id)
```

```
      .eq('user_id', user_id)
      .single();

    if (sessionError) throw sessionError;

    const { data: agents, error: agentsError } = await ctx.supabase
      .from('swarms_spreadsheet_session_agents')
      .select('*')
      .eq('session_id', input.session_id)
      .eq('user_id', user_id)
      .order('created_at', { ascending: true });

    if (agentsError) throw agentsError;

    return { ...session, agents };
  }),

getAllSessionsWithAgents: userProcedure.query(async ({ ctx }) => {
  const user_id = ctx.session.data.session?.user?.id || '';

  const { data: sessions, error: sessionsError } = await ctx.supabase
    .from('swarms_spreadsheet_sessions')
    .select('*')
    .eq('user_id', user_id)
    .order('created_at', { ascending: true });
```

```
  if (sessionsError) throw sessionsError;

  const { data: agents, error: agentsError } = await ctx.supabase
    .from('swarms_spreadsheet_session_agents')
    .select('*')
    .eq('user_id', user_id)
    .order('created_at', { ascending: true });

  if (agentsError) throw agentsError;

  const sessionsWithAgents = sessions.map((session) => {
    return {
      ...session,
      agents: agents.filter((agent) => agent.session_id === session.id),
    };
  });

  return sessionsWithAgents;
}),

addAgent: userProcedure
  .input(
    z.object({
      session_id: z.string().uuid(),
      name: z.string(),
      description: z.string(),
```

```
      system_prompt: z.string(),
      llm: z.string(),
      original_agent_id: z.string().uuid().optional(),
    }),
  )
  .mutation(async ({ ctx, input }) => {
    const user_id = ctx.session.data.session?.user?.id || '';

    const { data, error } = await ctx.supabase
      .from('swarms_spreadsheet_session_agents')
      .insert({
        ...input,
        user_id,
        status: 'idle',
      })
      .select()
      .single();

    if (error) throw error;
    return data;
  }),

updateAgentStatus: userProcedure
  .input(
    z.object({
      agent_id: z.string().uuid(),
```

```
      status: z.enum(['idle', 'running', 'completed', 'error']),
      output: z.string().optional(),
    }),
  )
  .mutation(async ({ ctx, input }) => {
    const user_id = ctx.session.data.session?.user?.id || '';

    const { error } = await ctx.supabase
      .from('swarms_spreadsheet_session_agents')
      .update({
        status: input.status,
        output: input.output,
      })
      .eq('id', input.agent_id)
      .eq('user_id', user_id);

    if (error) throw error;
    return true;
  }),

deleteAgent: userProcedure
  .input(
    z.object({
      agent_id: z.string().uuid(),
    }),
  )
```

```
.mutation(async ({ ctx, input }) => {
  const user_id = ctx.session.data.session?.user?.id || '';

  const { data: agent } = await ctx.supabase
    .from('swarms_spreadsheet_session_agents')
    .select('original_agent_id')
    .eq('id', input.agent_id)
    .eq('user_id', user_id)
    .single();

  if (agent) {
    if (!agent.original_agent_id) {
      await ctx.supabase
        .from('swarms_spreadsheet_session_agents')
        .delete()
        .eq('original_agent_id', input.agent_id)
        .eq('user_id', user_id);
    }
  }

  // Delete the agent itself
  const { error } = await ctx.supabase
    .from('swarms_spreadsheet_session_agents')
    .delete()
    .eq('id', input.agent_id)
    .eq('user_id', user_id);
```

```
    if (error) throw error;
    return true;
  }),

updateSessionTask: userProcedure
  .input(
    z.object({
      session_id: z.string().uuid(),
      task: z.string(),
    }),
  )
  .mutation(async ({ ctx, input }) => {
    const user_id = ctx.session.data.session?.user?.id || '';

    const { error } = await ctx.supabase
      .from('swarms_spreadsheet_sessions')
      .update({ task: input.task })
      .eq('id', input.session_id)
      .eq('user_id', user_id);

    if (error) throw error;
    return true;
  }),

updateSessionMetrics: userProcedure
```

```
  .input(
    z.object({
      session_id: z.string().uuid(),
      tasksExecuted: z.number(),
      timeSaved: z.number(),
    }),
  )
  .mutation(async ({ ctx, input }) => {
    const user_id = ctx.session.data.session?.user?.id || '';

    const { error } = await ctx.supabase
      .from('swarms_spreadsheet_sessions')
      .update({
        tasks_executed: input.tasksExecuted,
        time_saved: input.timeSaved,
      })
      .eq('id', input.session_id)
      .eq('user_id', user_id);

    if (error) throw error;
    return true;
  }),

updateSessionOutput: userProcedure
  .input(
    z.object({
```

```
      session_id: z.string().uuid(),
      output: z.record(z.any()),
    }),
  )
  .mutation(async ({ ctx, input }) => {
    const user_id = ctx.session.data.session?.user?.id || '';

    const { error } = await ctx.supabase
      .from('swarms_spreadsheet_sessions')
      .update({ output: input.output })
      .eq('id', input.session_id)
      .eq('user_id', user_id);

    if (error) throw error;
    return true;
  }),

getAllSessions: userProcedure.query(async ({ ctx }) => {
  const user_id = ctx.session.data.session?.user?.id || '';
  const { data, error } = await ctx.supabase
    .from('swarms_spreadsheet_sessions')
    .select('*')
    .eq('user_id', user_id)
    .order('created_at', { ascending: false });

  if (error) throw error;
```

```
  return data;
}),

setCurrentSession: userProcedure
  .input(z.object({ session_id: z.string().uuid() }))
  .mutation(async ({ ctx, input }) => {
    const user_id = ctx.session.data.session?.user?.id || '';

    await ctx.supabase
      .from('swarms_spreadsheet_sessions')
      .update({ current: false })
      .eq('user_id', user_id);

    const { error } = await ctx.supabase
      .from('swarms_spreadsheet_sessions')
      .update({ current: true })
      .eq('id', input.session_id)
      .eq('user_id', user_id);

    if (error) throw error;
    return true;
  }),

getDuplicateCount: userProcedure
  .input(z.object({ original_agent_id: z.string().uuid() }))
  .query(async ({ ctx, input }) => {
```

```
      const user_id = ctx.session.data.session?.user?.id || '';

      const { data, error } = await ctx.supabase
        .from('swarms_spreadsheet_session_agents')
        .select('id')
        .eq('original_agent_id', input.original_agent_id)
        .eq('user_id', user_id);

      if (error) throw error;
      return data.length;
    }),
});

export default panelRouter;
```