```
import {
  publicProcedure,
  router,
  userProcedure,
} from '@/app/api/trpc/trpc-router';
import {
  getSwarmPullRequestStatus,
  publishSwarmToGithub,
} from '@/shared/utils/api/swarm-publisher-github';
import { Tables } from '@/types_db';
import { TRPCError } from '@trpc/server';
import { z } from 'zod';

const explorerRouter = router({
  getModels: publicProcedure.query(async ({ ctx }) => {
    const models = await ctx.supabase
      .from('swarms_cloud_models')
      .select(

'id,name,unique_name,model_type,description,tags,slug,price_million_input,price_million_output',
      )
      .eq('enabled', true)
      .order('created_at', { ascending: false });
    return models;
  }),
  getModelBySlug: publicProcedure
```

```
    .input(z.string())
    .query(async ({ input, ctx }) => {
      const model = await ctx.supabase
        .from('swarms_cloud_models')
        .select(

'id,name,unique_name,model_type,description,tags,use_cases,model_card_md,slug,price_million_i
nput,price_million_output',
        )
        .eq('slug', input)
        .eq('enabled', true)
        .single();
      return model.data;
    }),

  // swarm
  validateSwarmName: userProcedure
    .input(z.string())
    .mutation(async ({ ctx, input }) => {
      const name = input;
      // name validation
      // only a-z, 0-9, _
      if (!/^[a-zA-Z0-9_ ]+$/.test(name)) {
        return {
          error: 'Invalid name, only a-z, 0-9, _ are allowed',
          valid: false,
```

```
      };
    }
    // at least 5 characters
    if (name.length < 5) {
      return {
        error: 'Name should be at least 5 characters',
        valid: false,
      };
    }

    const user_id = ctx.session.data.session?.user?.id || '';
    const swarm = await ctx.supabase
      .from('swarms_cloud_user_swarms')
      .select('*')
      .eq('name', name)
      .eq('user_id', user_id);
    const exists = (swarm.data ?? [])?.length > 0;
    return {
      valid: !exists,
      error: exists ? 'Name already exists' : '',
    };
  }),
addSwarm: userProcedure
  .input(
    z.object({
      name: z.string(),
```

```
    description: z.string().optional(),
    useCases: z.array(z.any()),
    tags: z.string().optional(),
    code: z.string(),
  }),
)
.mutation(async ({ ctx, input }) => {
  let name = input.name;

  const ownerEmail = ctx.session.data.session?.user?.email || '';
  // convert email to name , fully , replace any non a-z, 0-9, _ with _
  const ownerName = ownerEmail.replace(/[^a-zA-Z0-9_]/g, '_');

  // validate name , it should be a valid directroy name, a-z, 0-9, _
  if (!/^[a-zA-Z0-9_ ]+$/.test(name)) {
    throw 'Invalid name, only a-z, 0-9, _ are allowed';
  }
  // replace none a-z 0-9 space,  with _
  name = name.replace(/[^a-zA-Z0-9_]/g, '_');
  name = name.replace(' ', '_');

  // rate limiter - 1 swarm per hour
  const user_id = ctx.session.data.session?.user?.id;
  const lastSubmites = await ctx.supabase
    .from('swarms')
    .select('*')
```

```
    .eq('user_id', user_id)
    .order('created_at', { ascending: false })
    .limit(1);
  if ((lastSubmites?.data ?? [])?.length > 0) {
    const lastSubmit = lastSubmites.data?.[0];
    const lastSubmitTime = new Date(lastSubmit.created_at);
    const currentTime = new Date();
    const diff = currentTime.getTime() - lastSubmitTime.getTime();
    const diffHours = diff / (1000 * 60 * 60); // 1h
    if (diffHours < 1) {
      throw 'You can only submit one swarm per hour';
    }
  }
  // github stuff , make branch , add commits , create PR
  try {
    const res = await publishSwarmToGithub({
      name,
      code: input.code,
      ownerName,
      ownerEmail,
    });

    const swarm = await ctx.supabase
      .from('swarms_cloud_user_swarms')
      .insert([
        {
```

```
            name: name,
            use_cases: input.useCases,
            user_id: user_id,
            code: input.code,
            tags: input.tags,
            pr_id: res?.number || '',
            pr_link: res?._links.issue.href || '',
            description: input.description,
            status: 'pending',
          } as Tables<'swarms_cloud_user_swarms'>,
        ]);
      if (swarm.error) {
        throw swarm.error;
      }
      return true;
    } catch (e) {
      console.error(e);
    }
  }),
// Validate prompt
validatePrompt: userProcedure
  .input(z.string())
  .mutation(async ({ ctx, input }) => {
    const prompt = input;
    // at least 5 characters
    if (prompt.length < 5) {
```

```
      return {
        error: 'Prompt should be at least 5 characters',
        valid: false,
      };
    }

    const user_id = ctx.session.data.session?.user?.id || '';
    const promptData = await ctx.supabase
      .from('swarms_cloud_prompts')
      .select('*')
      .eq('prompt', prompt)
      .eq('user_id', user_id);
    const exists = (promptData.data ?? [])?.length > 0;
    return {
      valid: !exists,
      error: exists ? 'Prompt already exists' : '',
    };
  }),

// Add prompt
addPrompt: userProcedure
  .input(
    z.object({
      name: z.string().optional(),
      prompt: z.string(),
      description: z.string().optional(),
```

```
    useCases: z.array(z.any()),
    tags: z.string().optional(),
  }),
)
.mutation(async ({ ctx, input }) => {
  if (!input.prompt) {
    throw 'Prompt is required';
  }

  // at least 5 characters
  if (!input.name || input.name.trim()?.length < 2) {
    throw 'Name should be at least 2 characters';
  }

  // rate limiter - 1 prompt per minute
  const user_id = ctx.session.data.session?.user?.id ?? '';
  const lastSubmits = await ctx.supabase
    .from('swarms_cloud_prompts')
    .select('*')
    .eq('user_id', user_id)
    .order('created_at', { ascending: false })
    .limit(1);

  if ((lastSubmits?.data ?? [])?.length > 0) {
    const lastSubmit = lastSubmits.data?.[0] || { created_at: new Date() };
    const lastSubmitTime = new Date(lastSubmit.created_at);
```

```
    const currentTime = new Date();
    const diff = currentTime.getTime() - lastSubmitTime.getTime();
    const diffMinutes = diff / (1000 * 60); // 1 minute
  }

  try {
    const prompts = await ctx.supabase.from('swarms_cloud_prompts').insert([
      {
        name: input.name,
        use_cases: input.useCases,
        prompt: input.prompt,
        description: input.description,
        user_id: user_id,
        tags: input.tags,
        status: 'pending',
      } as Tables<'swarms_cloud_prompts'>,
    ]);
    if (prompts.error) {
      throw prompts.error;
    }
    return true;
  } catch (e) {
    console.error(e);
    throw "Couldn't add prompt";
  }
}),
```

```
// Update prompt
updatePrompt: userProcedure
  .input(
    z.object({
      id: z.string(),
      name: z.string(),
      prompt: z.string().optional(),
      description: z.string().optional(),
      useCases: z.array(z.any()),
      tags: z.string().optional(),
    }),
  )
  .mutation(async ({ ctx, input }) => {
    if (!input.prompt) {
      throw 'Prompt is required';
    }

    // at least 5 characters
    if (!input.name || input.name.trim()?.length < 2) {
      throw 'Name should be at least 2 characters';
    }

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

    try {
```

```
      const prompt = await ctx.supabase
        .from('swarms_cloud_prompts')
        .update({
          name: input.name,
          use_cases: input.useCases,
          prompt: input.prompt,
          description: input.description,
          tags: input.tags,
        } as Tables<'swarms_cloud_prompts'>)
        .eq('user_id', user_id)
        .eq('id', input.id)
        .select('*');

      if (prompt.error) {
        throw prompt.error;
      }
      return true;
    } catch (e) {
      console.error(e);
      throw 'Prompt could not be updated';
    }
  }),
getAllPrompts: publicProcedure
  .input(
    z.object({
      limit: z.number().default(6),
```

```
    offset: z.number().default(1),
    search: z.string().optional(), // Add search as an optional parameter
  }),
)
.query(async ({ ctx, input }) => {
  const { limit, offset, search } = input;

  let query = ctx.supabase
    .from('swarms_cloud_prompts')
    .select('*')
    .order('created_at', { ascending: false });

  // If a search query is provided, filter based on name or prompt fields
  if (search) {
    query = query
      .ilike('name', `%${search}%`)
      .or(`prompt.ilike.%${search}%`);
  }

  const prompts = await query.range(offset, offset + limit - 1);

  if (prompts.error) {
    console.error(prompts.error);
    throw prompts.error.message;
  }
```

```
    return prompts;
  }),
getPromptById: publicProcedure
  .input(z.string())
  .query(async ({ input, ctx }) => {
    const model = await ctx.supabase
      .from('swarms_cloud_prompts')
      .select('*')
      .eq('id', input)
      .single();
    return model.data;
  }),

reloadSwarmStatus: userProcedure
  .input(z.string())
  .mutation(async ({ ctx, input }) => {
    const swarm = await ctx.supabase
      .from('swarms_cloud_user_swarms')
      .select('*')
      .eq('id', input);
    if (swarm.data?.length === 0) {
      throw 'Swarm not found';
    }
    // if its already approved, no need to check
    const oldStatus = swarm.data?.[0].status;
    if (oldStatus === 'approved') {
```

```
      return 'approved';
    }
    const pr_id = swarm.data?.[0].pr_id;
    if (!pr_id) {
      throw 'PR not found';
    }
    const pr_status = await getSwarmPullRequestStatus(pr_id);
    let status: 'approved' | 'pending' | 'rejected' = 'pending';
    if (typeof pr_status != 'boolean' && pr_status.closed_at) {
      if (pr_status.merged) {
        status = 'approved';
      } else {
        status = 'rejected';
      }
    }
    // update status
    await ctx.supabase
      .from('swarms_cloud_user_swarms')
      .update({ status })
      .eq('id', input);
    return status;
  }),
getMyPendingSwarms: userProcedure.query(async ({ ctx }) => {
  const user_id = ctx.session.data.session?.user?.id || '';
  const swarms = await ctx.supabase
    .from('swarms_cloud_user_swarms')
```

```
    .select('*')
    .eq('user_id', user_id)
    .eq('status', 'pending');
  return swarms;
}),
getAllApprovedSwarms: userProcedure.query(async ({ ctx }) => {
  const swarms = await ctx.supabase
    .from('swarms_cloud_user_swarms')
    .select('id,name,description')
    .eq('status', 'approved');
  return swarms;
}),
getSwarmByName: publicProcedure
  .input(z.string())
  .query(async ({ ctx, input }) => {
    const swarm = await ctx.supabase
      .from('swarms_cloud_user_swarms')
      .select('id,name,description,use_cases,tags,status, user_id')
      .eq('name', input)
      .eq('status', 'approved');
    return swarm.data?.[0];
  }),
//agents
validateAgent: userProcedure
  .input(z.string())
  .mutation(async ({ ctx, input }) => {
```

```
    const agent = input;
    // at least 5 characters
    if (agent.length < 5) {
      return {
        error: 'Agent should be at least 5 characters',
        valid: false,
      };
    }

    const user_id = ctx.session.data.session?.user?.id || '';
    const agentData = await ctx.supabase
      .from('swarms_cloud_agents')
      .select('*')
      .eq('agent', agent)
      .eq('user_id', user_id);
    const exists = (agentData.data ?? [])?.length > 0;
    return {
      valid: !exists,
      error: exists ? 'Agent already exists' : '',
    };
  }),
// Add agent
addAgent: userProcedure
  .input(
    z.object({
      name: z.string(),
```

```
    agent: z.string(),
    language: z.string().optional(),
    description: z.string().optional(),
    requirements: z.array(z.any()),
    useCases: z.array(z.any()),
    tags: z.string().optional(),
  }),
)
.mutation(async ({ ctx, input }) => {
  if (!input.description) {
    throw 'Description is required';
  }

  // at least 5 characters
  if (!input.name || input.name.trim()?.length < 2) {
    throw 'Name should be at least 2 characters';
  }

  // rate limiter - 1 agent per minute
  const user_id = ctx.session.data.session?.user?.id ?? '';
  const lastSubmits = await ctx.supabase
    .from('swarms_cloud_agents')
    .select('*')
    .eq('user_id', user_id)
    .order('created_at', { ascending: false })
    .limit(1);
```

```
if ((lastSubmits?.data ?? [])?.length > 0) {
  const lastSubmit = lastSubmits.data?.[0] || { created_at: new Date() };
  const lastSubmitTime = new Date(lastSubmit.created_at);
  const currentTime = new Date();
  const diff = currentTime.getTime() - lastSubmitTime.getTime();
  const diffMinutes = diff / (1000 * 60); // 1 minute
  if (diffMinutes < 1) {
    throw 'You can only submit one agent per minute';
  }
}

try {
  const agents = await ctx.supabase.from('swarms_cloud_agents').insert([
    {
      name: input.name || null,
      description: input.description || null,
      user_id: user_id,
      use_cases: input.useCases,
      agent: input.agent,
      requirements: input.requirements,
      tags: input.tags || null,
      language: input.language,
      status: 'pending',
    } as Tables<'swarms_cloud_agents'>,
  ]);
```

```
      if (agents.error) {
        throw agents.error;
      }
      return true;
    } catch (e) {
      console.error(e);
      throw "Couldn't add agent";
    }
  }),
// Update agent
updateAgent: userProcedure
  .input(
    z.object({
      id: z.string(),
      name: z.string(),
      agent: z.string().optional(),
      language: z.string().optional(),
      description: z.string().optional(),
      requirements: z.array(z.any()).optional(),
      useCases: z.array(z.any()),
      tags: z.string().optional(),
    }),
  )
  .mutation(async ({ ctx, input }) => {
    if (!input.description) {
      throw 'Description is required';
```

```
}

// at least 5 characters
if (!input.name || input.name.trim()?.length < 2) {
  throw 'Name should be at least 2 characters';
}

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

try {
  const agent = await ctx.supabase
    .from('swarms_cloud_agents')
    .update({
      name: input.name,
      description: input.description,
      use_cases: input.useCases,
      agent: input.agent,
      requirements: input.requirements,
      tags: input.tags,
      language: input.language,
    } as Tables<'swarms_cloud_agents'>)
    .eq('user_id', user_id)
    .eq('id', input.id)
    .select('*');

  if (agent.error) {
```

```
        throw agent.error;
      }

      if (!agent.data?.length) {
        throw new Error('Agent not found');
      }

      return true;
    } catch (e) {
      console.error(e);
      throw "Couldn't add agent";
    }
  }),
getAllAgents: publicProcedure.query(async ({ ctx }) => {
  const agents = await ctx.supabase
    .from('swarms_cloud_agents')
    .select('*')
    .order('created_at', { ascending: false });

  return agents;
}),
getAgentById: publicProcedure
  .input(z.string())
  .query(async ({ input, ctx }) => {
    const agents = await ctx.supabase
      .from('swarms_cloud_agents')
```

```
      .select('*')
      .eq('id', input)
      .single();
    return agents.data;
  }),
getAgentsByUserId: userProcedure
  .input(z.string())
  .query(async ({ input, ctx }) => {
    const agents = await ctx.supabase
      .from('swarms_cloud_agents')
      .select('*')
      .eq('user_id', input)
      .order('created_at', { ascending: false });
    return agents;
  }),
//tools
validateTool: userProcedure
  .input(z.string())
  .mutation(async ({ ctx, input }) => {
    const tool = input;
    if (tool.length < 3) {
      return {
        error: 'Tool should be at least 3 characters',
        valid: false,
      };
    }
```

```
    const user_id = ctx.session.data.session?.user?.id || '';
    const toolData = await ctx.supabase
      .from('swarms_cloud_tools')
      .select('*')
      .eq('tool', tool)
      .eq('user_id', user_id);
    const exists = (toolData.data ?? [])?.length > 0;
    return {
      valid: !exists,
      error: exists ? 'Tool already exists' : '',
    };
  }),
// Add tool
addTool: userProcedure
  .input(
    z.object({
      name: z.string(),
      tool: z.string(),
      language: z.string().optional(),
      description: z.string().optional(),
      requirements: z.array(z.any()),
      useCases: z.array(z.any()),
      tags: z.string().optional(),
    }),
  )
```

```
.mutation(async ({ ctx, input }) => {
  if (!input.description) {
    throw 'Description is required';
  }

  if (!input.name || input.name.trim()?.length < 2) {
    throw 'Name should be at least 2 characters';
  }

  // rate limiter - 1 tool per 30 secs
  const user_id = ctx.session.data.session?.user?.id ?? '';
  const lastSubmits = await ctx.supabase
    .from('swarms_cloud_tools')
    .select('*')
    .eq('user_id', user_id)
    .order('created_at', { ascending: false })
    .limit(1);

  if ((lastSubmits?.data ?? [])?.length > 0) {
    const lastSubmit = lastSubmits.data?.[0] || { created_at: new Date() };
    const lastSubmitTime = new Date(lastSubmit.created_at);
    const currentTime = new Date();
    const diff = currentTime.getTime() - lastSubmitTime.getTime();
    const diffMinutes = diff / (1000 * 30); // 30 secs
    if (diffMinutes < 1) {
      throw 'You can only submit one tool per 30 secs';
```

```
    }
  }

  try {
    const tools = await ctx.supabase.from('swarms_cloud_tools').insert([
      {
        name: input.name || null,
        description: input.description || null,
        user_id: user_id,
        use_cases: input.useCases,
        tool: input.tool,
        requirements: input.requirements,
        tags: input.tags || null,
        language: input.language,
        status: 'pending',
      } as Tables<'swarms_cloud_tools'>,
    ]);
    if (tools.error) {
      throw tools.error;
    }
    return true;
  } catch (e) {
    console.error(e);
    throw "Couldn't add tool";
  }
}),
```

```
// Update tool
updateTool: userProcedure
  .input(
    z.object({
      id: z.string(),
      name: z.string(),
      tool: z.string().optional(),
      language: z.string().optional(),
      description: z.string().optional(),
      requirements: z.array(z.any()).optional(),
      useCases: z.array(z.any()),
      tags: z.string().optional(),
    }),
  )
  .mutation(async ({ ctx, input }) => {
    if (!input.description) {
      throw 'Description is required';
    }

    // at least 5 characters
    if (!input.name || input.name.trim()?.length < 2) {
      throw 'Name should be at least 2 characters';
    }

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

```
try {
  const tool = await ctx.supabase
    .from('swarms_cloud_tools')
    .update({
      name: input.name,
      description: input.description,
      use_cases: input.useCases,
      tool: input.tool,
      requirements: input.requirements,
      tags: input.tags,
      language: input.language,
    } as Tables<'swarms_cloud_tools'>)
    .eq('user_id', user_id)
    .eq('id', input.id)
    .select('*');

  if (tool.error) {
    throw tool.error;
  }

  if (!tool.data?.length) {
    throw new Error('Tool not found');
  }

  return true;
} catch (e) {
```

```
      console.error(e);
      throw "Couldn't add tool";
    }
  }),
getAllTools: publicProcedure.query(async ({ ctx }) => {
  const tools = await ctx.supabase
    .from('swarms_cloud_tools')
    .select('*')
    .order('created_at', { ascending: false });

  return tools;
}),
getToolById: publicProcedure
  .input(z.string())
  .query(async ({ input, ctx }) => {
    const tool = await ctx.supabase
      .from('swarms_cloud_tools')
      .select('*')
      .eq('id', input)
      .single();
    return tool.data;
  }),
checkReview: userProcedure
  .input(
    z.object({
      modelId: z.string(),
```

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

  try {
    const { data, error } = await ctx.supabase
      .from('swarms_cloud_reviews')
      .select('id')
      .eq('user_id', user_id)
      .eq('model_id', modelId)
      .single();

    if (error && error.code === 'PGRST116') {
      return { hasReviewed: false };
    }

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

    return { hasReviewed: !!data.id };
```

```
    } catch (error) {
      throw new TRPCError({
        code: 'INTERNAL_SERVER_ERROR',
        message: `Failed to check review`,
      });
    }
  }),

addReview: userProcedure
  .input(
    z.object({
      model_type: z.string(),
      model_id: z.string(),
      rating: z.number(),
      comment: z.string(),
    }),
  )
  .mutation(async ({ input, ctx }) => {
    const { model_id, model_type, rating, comment } = input;
    const user_id = ctx.session.data.session?.user?.id ?? '';

    try {
      const { data: existingReview, error: existingReviewError } =
        await ctx.supabase
          .from('swarms_cloud_reviews')
          .select('id')
```

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

if (existingReview?.id) {
  throw new TRPCError({
    code: 'BAD_REQUEST',
    message: 'User has already reviewed this model',
  });
}

// Insert new review
const newReview = await ctx.supabase
  .from('swarms_cloud_reviews')
  .insert([
    {
      user_id,
      model_id,
      model_type,
      rating,
      comment,
    },
  ]);

if (newReview.error) {
  throw new TRPCError({
```

```
          code: 'INTERNAL_SERVER_ERROR',
          message: 'Failed to insert review',
        });
      }

      return true;
    } catch (error) {
      console.error(error);
      throw new TRPCError({
        code: 'INTERNAL_SERVER_ERROR',
        message: `Failed to insert review`,
      });
    }
  }),
getReviews: publicProcedure
  .input(z.string())
  .query(async ({ input, ctx }) => {
    const modelId = input;

    try {
      const { data: reviews, error: reviewsError } = await ctx.supabase
        .from('swarms_cloud_reviews')
        .select(
          `
          id,
          comment,
```

```
      model_id,
      user_id,
      model_type,
      created_at,
      rating,
      users (
      full_name,
      username,
      email,
      avatar_url
      )
    `,
    )
    .eq('model_id', modelId)
    .order('created_at', { ascending: false });

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

  return reviews;
} catch (error) {
  console.error(error);
```

```
        throw new TRPCError({
          code: 'INTERNAL_SERVER_ERROR',
          message: `Failed to fetch reviews`,
        });
      }
    }),
});

export default explorerRouter;
```