```
import { router, userProcedure } from '@/app/api/trpc/trpc-router';
import { z } from 'zod';
import { TRPCError } from '@trpc/server';
import { User } from '@supabase/supabase-js';

export const dndRouter = router({
  saveFlow: userProcedure
    .input(
      z.object({
        flow_id: z.string().optional(),
        nodes: z.array(
          z.object({
            id: z.string(),
            type: z.string(),
            position: z.object({
              x: z.number(),
              y: z.number(),
            }),
            data: z.object({
              id: z.string(),
              name: z.string(),
              type: z.string(),
              model: z.string(),
              systemPrompt: z.string(),
              clusterId: z.string().nullable().optional(),
              isProcessing: z.boolean().nullable().optional(),
```

```
      lastResult: z.string().nullable().optional(),
      dataSource: z.string().nullable().optional(),
      dataSourceInput: z.string().nullable().optional(),
    }).passthrough(),
  }).passthrough()
),
edges: z.array(
  z.object({
    id: z.string(),
    source: z.string(),
    target: z.string(),
    type: z.string().optional(),
    animated: z.boolean().optional(),
    style: z.object({
      stroke: z.string(),
    }).optional(),
    markerEnd: z.object({
      type: z.string(),
      color: z.string(),
    }).optional(),
    data: z.object({
      label: z.string(),
    }).optional(),
  }).passthrough()
),
architecture: z.string(),
```

```
    results: z.record(z.any()),
  })
)
.mutation(async ({ ctx, input }) => {
  try {
    const user_id = ctx.session.data.session?.user?.id;

    if (!user_id) {
      throw new TRPCError({
        code: 'UNAUTHORIZED',
        message: 'User not authenticated',
      });
    }

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

    const flow_data = {
      nodes: input.nodes,
      edges: input.edges,
      architecture: input.architecture,
      results: input.results,
    };
```

```
let result;

if (input.flow_id) {
  // Update existing flow
  const { data, error } = await ctx.supabase
    .from('drag_and_drop_flows')
    .update({
      flow_data,
      current: true,
      updated_at: new Date().toISOString(),
    })
    .eq('id', input.flow_id)
    .eq('user_id', user_id) // Ensure user owns this flow
    .select()
    .single();

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

  result = data;
} else {
```

```
    // Create new flow
    const { data, error } = await ctx.supabase
      .from('drag_and_drop_flows')
      .insert({
        user_id,
        flow_data,
        current: true,
      })
      .select()
      .single();

    if (error) {
      throw new TRPCError({
        code: 'INTERNAL_SERVER_ERROR',
        message: 'Error creating flow',
      });
    }

    result = data;
  }

  return result;
} catch (error) {
  console.error('Error in saveFlow:', error);
  throw error instanceof TRPCError ? error : new TRPCError({
    code: 'INTERNAL_SERVER_ERROR',
```

```
        message: 'Error saving flow',
      });
    }
  }),

getCurrentFlow: userProcedure
  .input(z.object({ flowId: z.string().optional() }))
  .query(async ({ ctx, input }) => {
    const user_id = ctx.session.data.session?.user?.id;

    if (!user_id) {
      throw new TRPCError({
        code: 'UNAUTHORIZED',
        message: 'User not authenticated',
      });
    }

    let query = ctx.supabase
      .from('drag_and_drop_flows')
      .select('*');

    // If flowId is provided, use it to fetch specific flow
    if (input.flowId) {
      query = query.eq('id', input.flowId);
    } else {
      // Otherwise, get the current flow
```

```
      query = query.eq('current', true);
    }

    // Add user_id filter and get single result
    const { data, error } = await query
      .eq('user_id', user_id)
      .single();

    if (error) {
      console.error('Error fetching flow:', error);
      throw new TRPCError({
        code: 'INTERNAL_SERVER_ERROR',
        message: 'Error fetching flow',
      });
    }

    return data?.flow_data;
  }),

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

  const { data, error } = await ctx.supabase
    .from('drag_and_drop_flows')
    .select('*')
    .eq('user_id', user_id)
```

```
    .order('created_at', { ascending: false });

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

  return data;
}),

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

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

    const { error } = await ctx.supabase
      .from('drag_and_drop_flows')
      .update({ current: true })
      .eq('id', input.flow_id)
```

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

      if (error) {
        throw new TRPCError({
          code: 'INTERNAL_SERVER_ERROR',
          message: 'Error setting current flow',
        });
      }

      return true;
    }),
});
```