```
"use client"


import { useState, useCallback, useEffect } from "react"

import { Search, Upload, AlertCircle } from "lucide-react"

import { useQuery, useMutation, useQueryClient } from "react-query"

import { useToast } from "../ui/Toasts/use-toast"

import { Input } from "../spread_sheet_swarm/ui/input"

import { Alert, AlertDescription } from "../ui/alert"

import { useSession, useSupabaseClient } from '@supabase/auth-helpers-react'

import { useDropzone } from "react-dropzone"

import { z } from "zod"

import { v4 as uuidv4 } from 'uuid'

import { AddDocumentDialog, DocumentCard, DocumentDetailsModal } from "./doccard"

import { useRouter } from 'next/navigation'

import { supabaseAdmin } from "@/shared/utils/supabase/admin"

import { createClientComponentClient } from "@supabase/auth-helpers-nextjs"



// Constants

const MAX_FILE_SIZE = 50 * 1024 * 1024 // 50MB

const ALLOWED_FILE_TYPES = ['text/plain', 'text/csv', 'application/pdf', 'image/jpeg', 'image/png',

'application/json']

const LOCAL_STORAGE_KEY = "documentHubCache"

const RETRY_ATTEMPTS = 3

const RETRY_DELAY = 1000
```

```
const createClient = () => {
    return createClientComponentClient({
      supabaseUrl: process.env.NEXT_PUBLIC_SUPABASE_URL!,
      supabaseKey: process.env.NEXT_PUBLIC_SUPABASE_ANON_KEY!,
    })
}



const supabase = createClient()

// User Schema
const UserSchema = z.object({
  id: z.string(),
  email: z.string().email(),
  role: z.enum(['admin', 'user']),
  organization_id: z.string().optional(),
})

type User = z.infer<typeof UserSchema>

// Document Schema with User Mapping
const DocumentSchema = z.object({
  id: z.string().uuid(),
  name: z.string().min(1).max(255),
  type: z.string(),
```

```
  size: z.string(),
  user_id: z.string(),
  organization_id: z.string().optional(),
  uploaded_by_email: z.string().email(),
  upload_date: z.string(),
  content: z.string(),
  checksum: z.string(),
  version: z.number(),
  last_modified: z.string(),
  status: z.enum(['processing', 'completed', 'error']),
  metadata: z.record(z.string(), z.any()).optional(),
  permissions: z.object({
    can_read: z.array(z.string()),
    can_write: z.array(z.string()),
    can_delete: z.array(z.string()),
    is_public: z.boolean(),
  }),
})

type Document = z.infer<typeof DocumentSchema>

interface DocumentError extends Error {
  code?: string;
  details?: string;
}
```

```
// Access Control
const AccessControl = {
  canRead(document: Document, userId: string): boolean {
    return (
      document.user_id === userId ||
      document.permissions.can_read.includes(userId) ||
      document.permissions.is_public
    )
  },

  canWrite(document: Document, userId: string): boolean {
    return document.user_id === userId || document.permissions.can_write.includes(userId)
  },

  canDelete(document: Document, userId: string): boolean {
    return document.user_id === userId || document.permissions.can_delete.includes(userId)
  }
}

// Utility Functions
const generateChecksum = async (content: string): Promise<string> => {
  const msgBuffer = new TextEncoder().encode(content);
  const hashBuffer = await crypto.subtle.digest('SHA-256', msgBuffer);
  return Array.from(new Uint8Array(hashBuffer))
    .map(b => b.toString(16).padStart(2, '0'))
    .join('');
```

```
}

const retryOperation = async <T,>(
  operation: () => Promise<T>,
  attempts: number = RETRY_ATTEMPTS,
  delay: number = RETRY_DELAY
): Promise<T> => {
  try {
    return await operation();
  } catch (error) {
    if (attempts <= 1) throw error;
    await new Promise(resolve => setTimeout(resolve, delay));
    return retryOperation(operation, attempts - 1, delay * 2);
  }
}

// Supabase Operations
const createSupabaseOperations = (supabaseClient: any, user: User | null) => ({
  async fetchDocuments(): Promise<Document[]> {
    if (!supabaseClient || !user) throw new Error('Not authenticated');

    const { data, error } = await retryOperation(async () =>
      await supabaseClient
        .from("documents")
        .select("*")
```

```
.or(`user_id.eq.${user.id},permissions->can_read.cs.{${user.id}},permissions->is_public.eq.true`)
        .order("upload_date", { ascending: false })
    );

    if (error) throw error;
    return data?.map((doc: unknown) => DocumentSchema.parse(doc)) ?? [];
  },

  async addDocument(document: Document): Promise<Document> {
    if (!supabaseClient || !user) throw new Error('Not authenticated');

    const validatedDoc = DocumentSchema.parse({
      ...document,
      user_id: user.id,
      organization_id: user.organization_id,
      uploaded_by_email: user.email,
    });

    const { data, error } = await retryOperation(async () =>
      await supabaseClient
        .from("documents")
        .insert(validatedDoc)
        .single()
    );

    if (error) throw error;
```

```
  return DocumentSchema.parse(data);
},

async updateDocument(document: Document): Promise<Document> {
  if (!supabaseClient || !user) throw new Error('Not authenticated');
  if (!AccessControl.canWrite(document, user.id)) {
    throw new Error('Unauthorized to update document');
  }

  const validatedDoc = DocumentSchema.parse(document);
  const { data, error } = await retryOperation(async () =>
    await supabaseClient
      .from("documents")
      .update(validatedDoc)
      .match({ id: document.id })
      .single()
  );

  if (error) throw error;
  return DocumentSchema.parse(data);
},

async deleteDocument(documentId: string): Promise<void> {
  if (!supabaseClient || !user) throw new Error('Not authenticated');

  const { data: document } = await supabaseClient
```

```
      .from("documents")
      .select("*")
      .eq("id", documentId)
      .single();

    if (!AccessControl.canDelete(document, user.id)) {
      throw new Error('Unauthorized to delete document');
    }

    const { error } = await retryOperation(async () =>
      await supabaseClient
        .from("documents")
        .delete()
        .match({ id: documentId })
    );

    if (error) throw error;
  },

  async shareDocument(documentId: string, shareWith: { email: string, permissions: ('read' | 'write' |
'delete')[] }): Promise<Document> {
    if (!supabaseClient || !user) throw new Error('Not authenticated');

    const { data: document } = await supabaseClient
      .from("documents")
      .select("*")
```

```
  .eq("id", documentId)
  .single();

if (!AccessControl.canWrite(document, user.id)) {
  throw new Error('Unauthorized to share document');
}

const { data: sharedWithUser } = await supabaseClient
  .from("users")
  .select("id")
  .eq("email", shareWith.email)
  .single();

if (!sharedWithUser) throw new Error('User not found');

const updatedPermissions = {
  ...document.permissions,
  can_read: shareWith.permissions.includes('read')
    ? [...document.permissions.can_read, sharedWithUser.id]
    : document.permissions.can_read,
  can_write: shareWith.permissions.includes('write')
    ? [...document.permissions.can_write, sharedWithUser.id]
    : document.permissions.can_write,
  can_delete: shareWith.permissions.includes('delete')
    ? [...document.permissions.can_delete, sharedWithUser.id]
    : document.permissions.can_delete,
```

```
    };

    return await this.updateDocument({
      ...document,
      permissions: updatedPermissions,
    });
  }
});

// Local Storage Operations with User Context
const createLocalStorageOperations = (userId: string) => ({
  save(documents: Document[]): void {
    if (typeof window === 'undefined') return;
    try {
      localStorage.setItem(`${LOCAL_STORAGE_KEY}_${userId}`, JSON.stringify(documents));
    } catch (error) {
      console.error('Error saving to localStorage:', error);
    }
  },

  get(): Document[] {
    if (typeof window === 'undefined') return [];
    try {
      const cached = localStorage.getItem(`${LOCAL_STORAGE_KEY}_${userId}`);
      return cached ? JSON.parse(cached) : [];
    } catch (error) {
```

```
      console.error('Error reading from localStorage:', error);
      return [];
    }
  },

  clear(): void {
    if (typeof window === 'undefined') return;
    localStorage.removeItem(`${LOCAL_STORAGE_KEY}_${userId}`);
  }
});

// Logging Service
const createLogService = (supabaseClient: any, user: User | null) => ({
  async logError(error: DocumentError, context: string): Promise<void> {
    const errorLog = {
      timestamp: new Date().toISOString(),
      user_id: user?.id,
      organization_id: user?.organization_id,
      error: {
        message: error.message,
        code: error.code,
        details: error.details,
        stack: error.stack
      },
      context,
    };
```

```
    await supabaseClient?.from('error_logs').insert(errorLog);
    console.error(`[${context}]`, errorLog);
  },

  async logActivity(action: string, details: any): Promise<void> {
    const activityLog = {
      timestamp: new Date().toISOString(),
      user_id: user?.id,
      organization_id: user?.organization_id,
      action,
      details,
    };

    await supabaseClient?.from('activity_logs').insert(activityLog);
  }
});

// React Component
export default function EnterpriseDataHub() {
  const { toast } = useToast();
  const session = useSession();
  const supabaseClient = useSupabaseClient();
  const [searchQuery, setSearchQuery] = useState("");
  const [selectedDocument, setSelectedDocument] = useState<Document | null>(null);
  const [isOffline, setIsOffline] = useState(!navigator.onLine);
```

```
const queryClient = useQueryClient();
const router = useRouter();
// Removed duplicate declaration of isLoading state

const user: any = session?.user ? {
  id: session.user.id,
  email: session.user.email!,
  role: 'user',
  organization_id: session.user.user_metadata.organization_id,
} : null;

const supabaseOperations = createSupabaseOperations(supabaseClient, user);
const localStorageOperations = user ? createLocalStorageOperations(user.id) : null;
const logService = createLogService(supabaseClient, user);


// Auth state check
useEffect(() => {
  const checkAuth = async () => {
    try {
      const { data: { session }, error } = await supabaseAdmin.auth.getSession()
      if (error) throw error
      if (!session) {
        router.push('/auth/signin') // Redirect to your sign-in page
      }
    } catch (error) {
```

```
      console.error('Auth error:', error)
      toast({
        title: "Authentication Error",
        description: "Please try signing in again.",
        variant: "destructive",
      })
      router.push('/auth/signin')
    }
  }

  checkAuth()
}, [router, toast])

// Subscription to auth changes
useEffect(() => {
  const {
    data: { subscription },
  } = supabaseAdmin.auth.onAuthStateChange((_event, session) => {
    if (!session) {
      router.push('/auth/signin')
    }
  })

  return () => subscription.unsubscribe()
}, [router])
```

```
// Network status monitoring
useEffect(() => {
  const handleOnline = () => setIsOffline(false);
  const handleOffline = () => setIsOffline(true);

  window.addEventListener('online', handleOnline);
  window.addEventListener('offline', handleOffline);

  return () => {
    window.removeEventListener('online', handleOnline);
    window.removeEventListener('offline', handleOffline);
  };
}, []);

// Document fetching with error handling and offline support
const { data: documents, isLoading, isError } = useQuery<Document[], Error>(
  ["documents", user?.id],
  async () => {
    if (!user || !localStorageOperations) throw new Error('Not authenticated');

    try {
      const docs = await supabaseOperations.fetchDocuments();
      localStorageOperations.save(docs);
```

```
      return docs;
    } catch (error) {
      if (isOffline) {
        return localStorageOperations.get();
      }
      throw error;
    }
  },
  {
    enabled: !!user,
    retry: RETRY_ATTEMPTS,
    retryDelay: RETRY_DELAY,
    onError: async (error) => {
      await logService.logError(error as DocumentError, 'document_fetch');
      toast({
        title: "Error fetching documents",
        description: "Using cached data. Please check your connection.",
        variant: "destructive",
      });
      if (localStorageOperations) {
        queryClient.setQueryData(["documents", user?.id], localStorageOperations.get());
      }
    },
  }
);
```

```
// Document addition mutation
const addDocumentMutation = useMutation(
  async (newDoc: Omit<Document, 'user_id' | 'organization_id' | 'uploaded_by_email'>) => {
    if (!user || !localStorageOperations) throw new Error('Not authenticated');

    const checksum = await generateChecksum(newDoc.content);
    const documentWithUser = {
      ...newDoc,
      checksum,
      user_id: user.id,
      organization_id: user.organization_id,
      uploaded_by_email: user.email,
    } as Document;

    // Save locally first
    const localDocs = localStorageOperations.get();
    localStorageOperations.save([...localDocs, documentWithUser]);

    // Then save to Supabase if online
    if (!isOffline) {
      return await supabaseOperations.addDocument(documentWithUser);
    }
    return documentWithUser;
  },
  {
    onSuccess: async (doc) => {
```

```
        await logService.logActivity('document_added', { documentId: doc.id });
        queryClient.invalidateQueries(["documents", user?.id]);
        toast({
          title: "Document added successfully",
          description: isOffline ? "Document saved locally. Will sync when online." : "Document saved to cloud.",
        });
      },
      onError: async (error) => {
        await logService.logError(error as DocumentError, 'document_add');
        toast({
          title: "Error adding document",
          description: "There was a problem adding your document. Please try again.",
          variant: "destructive",
        });
      },
    }
  );

  // Share document mutation
  const shareDocumentMutation = useMutation(
    async ({ documentId, shareWith }: { documentId: string, shareWith: { email: string, permissions: ('read' | 'write' | 'delete')[] } }) => {
      return await supabaseOperations.shareDocument(documentId, shareWith);
    },
    {
```

```
    onSuccess: async (doc) => {
      await logService.logActivity('document_shared', { documentId: doc.id });
      queryClient.invalidateQueries(["documents", user?.id]);
      toast({
        title: "Document shared successfully",
        description: "The selected users now have access to this document.",
      });
    },
    onError: async (error) => {
      await logService.logError(error as DocumentError, 'document_share');
      toast({
        title: "Error sharing document",
        description: "There was a problem sharing your document. Please try again.",
        variant: "destructive",
      });
    },
  }
);

// File upload handling
const onDrop = useCallback(async (acceptedFiles: File[]) => {
  if (!user) {
    toast({
      title: "Authentication required",
      description: "Please sign in to upload documents.",
      variant: "destructive",
```

```
  });
  return;
}

for (const file of acceptedFiles) {
  if (file.size > MAX_FILE_SIZE) {
    toast({
      title: "File too large",
      description: `${file.name} exceeds the 50MB limit`,
      variant: "destructive",
    });
    continue;
  }

  if (!ALLOWED_FILE_TYPES.includes(file.type)) {
    toast({
      title: "Invalid file type",
      description: `${file.name} is not a supported file type`,
      variant: "destructive",
    });
    continue;
  }

  try {
    const reader = new FileReader();
    const content = await new Promise<string>((resolve, reject) => {
```

```
  reader.onload = () => resolve(reader.result as string);
  reader.onerror = () => reject(reader.error);
  reader.readAsDataURL(file);
});

const newDocument = {
  id: uuidv4(),
  name: file.name,
  type: file.type.split('/')[1] || 'unknown',
  size: `${(file.size / 1024 / 1024).toFixed(2)} MB`,
  upload_date: new Date().toISOString(),
  content,
  checksum: '',
  version: 1,
  last_modified: new Date().toISOString(),
  status: 'processing' as const,
  metadata: {
    originalName: file.name,
    lastModified: file.lastModified,
    type: file.type,
  },
  permissions: {
    can_read: [],
    can_write: [],
    can_delete: [],
    is_public: false,
```

```
        },
      };

      await addDocumentMutation.mutateAsync(newDocument);
    } catch (error) {
      await logService.logError(error as DocumentError, 'file_upload');
      toast({
        title: "Upload Error",
        description: `Error uploading ${file.name}. Please try again.`,
        variant: "destructive",
      });
    }
  }
}, [addDocumentMutation, toast, user]);

const { getRootProps, getInputProps, isDragActive } = useDropzone({
  onDrop,
  accept: {
    'text/plain': ['.txt'],
    'text/csv': ['.csv'],
    'application/pdf': ['.pdf'],
    'image/jpeg': ['.jpg', '.jpeg'],
    'image/png': ['.png'],
    'application/json': ['.json']
  },
  maxSize: MAX_FILE_SIZE,
```

```
});


if (!user) {
  return (
    <div className="container mx-auto p-6">
      <Alert>
        <AlertCircle className="h-4 w-4" />
        <AlertDescription>
          Please sign in to access the document hub.
        </AlertDescription>
      </Alert>
    </div>
  );
}


return (
  <div className="container mx-auto p-6 space-y-6">
    {isOffline && (
      <Alert variant="default">
        <AlertCircle className="h-4 w-4" />
        <AlertDescription>
          You are currently offline. Changes will be saved locally and synced when you're back online.
        </AlertDescription>
      </Alert>
    )}
```

```
<div className="flex justify-between items-center">
  <h1 className="text-3xl font-bold">Enterprise Data Hub</h1>
  <div className="flex gap-2">
    <AddDocumentDialog onAddDocument={(doc: Omit<{ id: string; name: string; type: string; status: "completed" | "error" | "processing"; user_id: string; content: string; size: string; version: number; uploaded_by_email: string; upload_date: string; checksum: string; last_modified: string; permissions: { can_read: string[]; can_write: string[]; can_delete: string[]; is_public: boolean }; metadata?: Record<string, any> | undefined; organization_id?: string | undefined }, "user_id" | "organization_id" | "uploaded_by_email">) => addDocumentMutation.mutateAsync(doc)} />
    <span className="text-sm text-gray-500">
      Signed in as {user.email}
    </span>
  </div>
</div>

<div className="relative">
  <Search className="absolute left-2 top-1/2 transform -translate-y-1/2 text-gray-400" />
  <Input
    type="search"
    placeholder="Search documents..."
    className="pl-10"
    value={searchQuery}
    onChange={(e) => setSearchQuery(e.target.value)}
  />
</div>
```

```
<div
  {...getRootProps()}
  className={`border-2 border-dashed rounded-lg p-10 text-center transition-colors ${
    isDragActive ? "border-primary bg-primary/10" : "border-gray-300"
  }`}
>
  <input {...getInputProps()} />
  <Upload className="mx-auto h-12 w-12 text-gray-400" />
  <p className="mt-2 text-sm text-gray-600">
    Drag 'n' drop some files here, or click to select files
  </p>
</div>

{isLoading ? (
  <div className="text-center">Loading documents...</div>
) : isError ? (
  <Alert variant="destructive">
    <AlertCircle className="h-4 w-4" />
    <AlertDescription>
      Error loading documents. Please try again later.
    </AlertDescription>
  </Alert>
) : documents?.length === 0 ? (
  <div className="text-center text-gray-500">No documents found. Try uploading some!</div>
) : (
  <div className="grid grid-cols-1 md:grid-cols-2 lg:grid-cols-3 xl:grid-cols-4 gap-6">
```

```
        {documents?.map((doc) => (
          <DocumentCard
            key={doc.id}
            document={doc}
            currentUser={user}
                  onShare={(shareWith: any) => shareDocumentMutation.mutate({ documentId: doc.id,
shareWith })}
            onClick={() => setSelectedDocument(doc)}
          />
        ))}
      </div>
    )}

    {selectedDocument && (
      <DocumentDetailsModal
        document={selectedDocument}
        currentUser={user}
        onClose={() => setSelectedDocument(null)}
        onShare={(shareWith: any) => shareDocumentMutation.mutate({
          documentId: selectedDocument.id,
          shareWith
        })}
      />
    )}
  </div>
);
```

```
}
```