Universidad Autónoma de Madrid

Escuela Politécnica Superior

DISEÑO DE APLICACIONES COLABORATIVAS EN LA NUBE

Video Editor

Equipo I:
Zhijie Qian
Martín Soto Alandete
Junhan Li
Dong Xu

12 de Mayo de 2025

Índice

1. Requisitos

1.1. Base

En esta práctica, se intenta resolver el problema de sincronización de documentos a tiempo real para múltiples usuarios. Este problema surge cuando existe un entorno donde haya más de un usuario activo conectado a la red, y todos los usuarios quiere editar un mismo fichero o documento. Para poder lograr este objetivo, hemos decidido implementar un editor de video a tiempo real, pero tomando un código base existente para ahorrar el tiempo de desarrollo, ya que nuestro objetivo es resolver el problema de sincronización y no tanto en implementar un editor de video.
La forma de resolver la sincronización sería parecido a Google Doc Wave, donde el cliente puede realizar múltiples cambios dando múltiples estados pero el servidor siempre tendrá un único estado común a todo el mundo, por lo que sería en el lado de cliente que se resuelva todo tipo de conflictos y se subirá los cambios al servidor.
El código base que se ha utilizado es fabric-video-editor, que se base en NextJs con estilo de código en React. De manera resumida, el código que nos importa es `store/Store.ts`, donde se define cómo debería ser la estructura y el funcionamiento de los datos.

```
export class Store {
  canvas: fabric.Canvas | null

  backgroundColor: string;

  selectedMenuOption: MenuOption;
  audios: string[]
  videos: string[]
  images: string[]
  editorElements: EditorElement[]
  selectedElement: EditorElement | null;

  maxTime: number
  animations: Animation[]
  animationTimeLine: anime.AnimeTimelineInstance;
  playing: boolean;

  currentKeyFrame: number;
  fps: number;

  possibleVideoFormats: string[] = ['mp4', 'webm'];
  selectedVideoFormat: 'mp4' | 'webm';

  constructor() {
    this.canvas = null;
    this.videos = [];
    this.images = [];
    this.audios = [];
    this.editorElements = [];
    this.backgroundColor = '#111111';
    this.maxTime = 30 * 1000;
    this.playing = false;
    this.currentKeyFrame = 0;
    this.selectedElement = null;
    this.fps = 60;
    this.animations = [];
    this.animationTimeLine = anime.timeline();
    this.selectedMenuOption = 'Video';
    this.selectedVideoFormat = 'mp4';
    makeAutoObservable(this);
  }
};
```

1.2. Autenticación

El proyecto implementa un sistema completo de autenticación utilizando Firebase Authentication, que ofrece dos métodos principales de inicio de sesión:

1. Inicio de sesión con correo electrónico/contraseña - Permite a los usuarios crear cuentas y acceder mediante correo electrónico y contraseña.

2. Inicio de sesión con Google OAuth - Permite a los usuarios acceder rápidamente utilizando sus cuentas de Google.

La implementación central del sistema de autenticación se encuentra en el archivo src/utils/firebaseConfig.ts, que inicializa los servicios de Firebase, incluyendo el servicio de autenticación.
La funcionalidad de inicio de sesión se implementa en src/app/login/page.tsx , que incluye tanto el inicio de sesión con correo electrónico como con Google:

```
const handleLogin = async (e: React.FormEvent) => {
  e.preventDefault();
  setIsLoading(true);
  setError("");

  try {
    // Implement email/password login
    const userCredential = await signInWithEmailAndPassword(auth, email, password);
    const user = userCredential.user;
    // Redirect to editor page after successful login
    window.location.href = "/workspace";
  } catch (error: any) {
    console.error("Login failed:", error);
    const errorCode = error.code;
    const errorMessage = error.message;
    console.log(errorCode, errorMessage);
    setError("Failed to sign in. Please check your credentials and try again.");
  } finally {
    setIsLoading(false);
  }
};
const handleGoogleLogin = async (e: React.FormEvent) => {
  // Implement Google OAuth login
  e.preventDefault();
  setIsLoading(true);

  try {
    const provider = new GoogleAuthProvider();
    const result = await signInWithPopup(auth, provider);
    const credential = GoogleAuthProvider.credentialFromResult(result);
    const token = credential?.accessToken;
    const user = result.user;
    window.location.href = "/workspace";
  } catch (error: any) {
    console.error("Google login failed:", error);
    const errorCode = error.code;
    const errorMessage = error.message;
    const email = error.customData?.email;
    const credential = GoogleAuthProvider.credentialFromError(error);
    console.log(errorCode, errorMessage, email, credential);
  } finally {
    setIsLoading(false);
  }
};
```

La funcionalidad de registro se implementa en src/app/signup/page.tsx, que también soporta registro por correo electrónico y por Google:

```
const handleSignup = async (e: React.FormEvent) => {
  e.preventDefault();
  setError("");
  // Basic validation
  if (password !== confirmPassword) {
    setError("Passwords do not match");
    return;
  }
  setIsLoading(true);
  try {
    // Implement email/password signup with full name
    const userCredential = await createUserWithEmailAndPassword(auth, email,
  password);
    const user = userCredential.user;
    // Update the user profile with the full name
    if (user) {
      await updateProfile(user, {
        displayName: name
      });
    }
    window.location.href = "/workspace";
  } catch (error: any) {
    const errorCode = error.code;
    const errorMessage = error.message;
    setError("Failed to create account. Please try again.");
  } finally {
    setIsLoading(false);
  }
};
const handleGoogleSignup = async (e: React.FormEvent) => {
  // Implement Google OAuth signup
  e.preventDefault();
  setIsLoading(true);
  setError("");
  try {
    const provider = new GoogleAuthProvider();
    const result = await signInWithPopup(auth, provider);
    const credential = GoogleAuthProvider.credentialFromResult(result);
    const token = credential?.accessToken;
    const user = result.user;
    // Store the user's full name from Google account
    if (user && !user.displayName) {
      // If for some reason the user doesn't have a display name from Google
      await updateProfile(user, {
        displayName: "UserNoName"
      });
    }
    window.location.href = "/workspace";
  } catch (error: any) {
    const errorCode = error.code;
    const errorMessage = error.message;
    const email = error.customData?.email;
    const credential = GoogleAuthProvider.credentialFromError(error);
    console.log(errorCode, errorMessage, email, credential);
    setError("Failed to sign up with Google. Please try again.");
  } finally {
    setIsLoading(false);
  }
};
```

Además, el proyecto implementa un **AuthContext** para gestionar el estado de autenticación en toda la aplicación.

1.3. Workspace

El Workspace es el área central donde los usuarios gestionan sus proyectos, permitiéndoles ver, crear y administrar sus proyectos de edición de video. Esta funcionalidad incluye los siguientes componentes principales:

1. Visualización de lista de proyectos - Muestra los proyectos creados por el usuario y los proyectos compartidos con el usuario
2. Funcionalidad de creación de proyectos - Permite a los usuarios crear nuevos proyectos de edición de vídeo.
3. Funcionalidad de gestión de proyectos - Permite a los usuarios abrir o eliminar proyectos

El componente **ProjectCard** se utiliza para mostrar la información de un proyecto individual y proporciona funcionalidades para abrir y eliminar proyectos:

```
export const ProjectCard: React.FC<ProjectCardProps> = ({ project, onDelete }) => {
    const { currentUser } = useAuth();
    const isOwner = currentUser?.uid === project.ownerId;

    // check if the current user is a collaborator
    const userRole = currentUser?.email && project.collaborators?.[currentUser.email.
    toLowerCase()]?.role;
    const isCollaborator = !!userRole;
    // get user role display text
    const getRoleText = () => {
        if (isOwner) return 'Owner';
        if (userRole === 'editor') return 'Editor';
        if (userRole === 'viewer') return 'Viewer';
        return '';
    };
    const handleDelete = async () => {
        if (!confirm('Are you sure you want to delete this project?')) return;
        await deleteProject(project.id);
        onDelete();
    };
    // if the current user is neither the owner nor a collaborator, do not display
    the project card
    if (!isOwner && !isCollaborator) {
        return null;
    }
    return (
    //divs...
    );
)
```

El componente modal **CreateProjectModal** permite a los usuarios ingresar un nombre y descripción para el proyecto, y guarda el nuevo proyecto en la base de datos Firestore:

```
  const handleCreateProject = async (e: React.FormEvent) => {
    e.preventDefault();
    if (!currentUser) {
      setError('You must be logged in to create a project');
      return;
    }
    if (!projectName.trim()) {
      setError('Project name is required');
      return;
    }
    setIsLoading(true);
    setError(null);

```

```
    try {
      const timestamp = Date.now();
      // create a new Project object
      const newProject: Omit<Project, 'id'> = {
        name: projectName.trim(),
        description: projectDescription.trim(),
        createdAt: timestamp,
        updatedAt: timestamp,
        ownerId: currentUser.uid,
        ownerName: currentUser.displayName || currentUser.email?.split('@')[0] || '
User',
        collaborators: {},
        thumbnailUrl: ''
      };
      // Use Firestore to create the project
      console.log('Creating new project:', newProject);
      let projectId;
      try {
        // use addDoc to create project
        const docRef = await addDoc(collection(projectFirestore, 'projects'),
newProject);
        projectId = docRef.id;
        console.log('Project created with ID:', projectId);
      } catch (error) {
        console.error('Error in project creation:', error);
        throw error;
      }
      setProjectName('');
      setProjectDescription('');
      onClose();
      if (onProjectCreated && projectId) {
        onProjectCreated(projectId);
      }
      if (projectId) {
        router.push(`/editor/${projectId}`);
      }
    } catch (err) {
      console.error('Error creating project:', err);
      setError('Failed to create project. Please try again.');
    } finally {
      setIsLoading(false);
    }
  };
```

1.4. Chat

La funcionalidad de Chat es uno de los componentes centrales de colaboración en tiempo real del proyecto, permitiendo a los usuarios comunicarse mientras editan proyectos de video. Esta funcionalidad está implementada utilizando Firebase Realtime Database, asegurando que los mensajes se entreguen en tiempo real a todos los colaboradores.

Implementación Técnica

1. Firebase Realtime Database - Utilizado para almacenar y sincronizar mensajes de chat
2. React Hooks - Para gestionar el estado y ciclo de vida de los componentes
3. Firebase SDK - Para interactuar con los servicios de Firebase

La implementación principal de la funcionalidad de chat se encuentra en el archivo src/services/-chatService.ts, que proporciona funciones para enviar y suscribirse a mensajes:

```
import {
  ref,
  push,
  onValue,
  off,
  query,
  orderByChild,
  limitToLast,
  set,
  serverTimestamp,
  DatabaseReference
} from 'firebase/database';
import { database } from '@/utils/firebaseConfig';
import { ChatMessage } from '@/types/chat';
// Obtener referencia al chat del proyecto
export const getProjectChatRef = (projectId: string): DatabaseReference => {
  return ref(database, `chats/${projectId}/messages`);
};
// Enviar mensaje
export const sendMessage = async (
  projectId: string,
  text: string,
  senderId: string,
  senderName: string,
  senderPhotoURL?: string | null
): Promise<void> => {
  try {
    const chatRef = getProjectChatRef(projectId);
    const newMessageRef = push(chatRef);
    await set(newMessageRef, {
      id: newMessageRef.key,
      text,
      senderId,
      senderName,
      senderPhotoURL,
      timestamp: serverTimestamp(),
      projectId
    });
  } catch (error) {
    console.error('Error sending message:', error);
    throw error;
  }
};
// Suscribirse a mensajes
export const subscribeToMessages = (
  projectId: string,
  callback: (messages: ChatMessage[]) => void,
  limit: number = 50
): () => void => {
  const chatRef = getProjectChatRef(projectId);
  const messagesQuery = query(
    chatRef,
    orderByChild('timestamp'),
    limitToLast(limit)
  );
  const handleMessages = (snapshot: any) => {
    const data = snapshot.val();
    if (!data) {
      callback([]);
      return;
    }
    const messages = Object.values(data) as ChatMessage[];
    // Ordenar por tiempo
    messages.sort((a, b) => (a.timestamp || 0) - (b.timestamp || 0));
```

```
    callback(messages);
  };
  onValue(messagesQuery, handleMessages);
  return () => off(messagesQuery, 'value', handleMessages);
};
```

La definición de tipos para los mensajes de chat:

```
export interface ChatMessage {
  id: string;
  text: string;
  senderId: string;
  senderName: string;
  senderPhotoURL?: string | null;
  timestamp: number;
  projectId: string;
}
export interface ChatState {
  messages: ChatMessage[];
  loading: boolean;
  error: string | null;
}
```

1.5. Drag and Drop

Para poder reordenar dinámicamente los tracks del proyecto se ha utilizado la biblioteca dnd-kit. Se ha envuelto el listado de pistas dentro del componente DndContext de @dnd-kit/core que servirá como proveedor del contexto de arrastre ademas de asignare la siguiente configuración:

- sensors: utilizando PointerSensor para detectar interacción con el mouse.
- collisionDetection: closestCenter: estrategia que identifica el destino del elemento arrastrado con base en la cercanía al centro de los otros elementos.
- onDragEnd: manejador que actualiza el orden de los elementos en el estado global.

Los elementos a ordenar (store.editorElements y store.conflit) se combinan y ordenan por su propiedad order, para mantener un orden visual coherente. Luego se pasan a SortableContext, que habilita su reordenamiento usando la estrategia verticalListSortingStrategy.
Cada elemento es representado por el componente SortableTimeFrameView, que está envuelto en el hook useSortable. Este hook:

1. Asigna referencias DOM (setNodeRef).
2. Aplica estilos animados con transformaciones (transform, transition).
3. Proporciona attributes y listeners para habilitar la interacción por teclado y mouse.
4. Expone isDragging para manejar estilos condicionales durante el movimiento.

El div que actúa como "handle"de arrastre es visualmente una burbuja azul al costado de cada pista. Cuando se completa un arrastre, handleDragEnd compara las posiciones inicial y final. Si hay un cambio:

1. Se crea un nuevo arreglo con los elementos reordenados usando arrayMove.
2. Se detectan los elementos cuyo índice cambió (basado en su propiedad order).
3. Proporciona attributes y listeners para habilitar la interacción por teclado y mouse.

4. Se actualiza el estado global (store.updateEditorElement) solo para los elementos que realmente cambiaron de lugar.

Esto garantiza que el estado se mantenga coherente y que no se hagan escrituras innecesarias.

```
<DndContext
  sensors={sensors}
  collisionDetection={closestCenter}
  onDragEnd={handleDragEnd}
>
  <SortableContext
    items={allElements.map((el) => el.id)}
    strategy={verticalListSortingStrategy}
  >
    <div className="flex flex-col">
    {allElements.map((element) => (
      <SortableTimeFrameView
        key={element.id}
        element={element}
      />
    ))}
    </div>
  </SortableContext>
</DndContext>
```

Para realizar el re ordenamiento de los elementos se ejecuta la siguiente función cada vez que son soltados.

En primer lugar se obtiene el index antiguo y nuevo del elemento arrastrado. Luego generamos una copia del array elementos ordenados por el campo order. Descartamos de la copia el elemento con el index antiguo (popElementByIndex) y lo insertamos con el nuevo index (insertElementAtIndex).

Con el objetivo de hacer mas eficiente el reordenado, obtenemos un nuevo listado solo con los elementos que han cambiado de orden (change) finalmente son actualizados en el store

```
const handleDragEnd = (event: DragEndEvent) => {
    const { active, over } = event;
    if (!over || active.id === over.id) return;

    var oldIndex = store.editorElements.find((el) => el.id ===  active.id)!.order;
    var newIndex = store.editorElements.find((el) => el.id === over.id)!.order;

    const elements = store.editorElements.sort((a, b) => a.order - b.order);
    const oldEle = popElementByIndex(elements, oldIndex);
    const reordered = insertElementAtIndex(elements, newIndex, oldEle!);
    const changed:EditorElement[] = [];
    reordered.forEach((el, idx) => {
      if (el.order != idx) {
        changed.push({ ...el, order: idx });
      }
    });

    // Iterate through changed elements and call updateEditorElement
    changed.forEach((element) => {
      store.updateEditorElement(element);
    });
  };

  function popElementByIndex(arr: EditorElement[], index: number) {
    if (index >= 0 && index < arr.length) {
      return arr.splice(index, 1)[0]; // Remove 1 element at the given index and
    return the removed element
```

```
    } else {
      return undefined; // Or throw an error, depending on your preference for out-of-bounds indices
    }
  }

  function insertElementAtIndex(arr: EditorElement[], index: number, element: EditorElement) {
    arr.splice(index, 0, element);
    return arr; // Returns the modified array for convenience
}
```

1.6. Almacenamiento

Para el almacenamiento, en el código original teníamos siguientes atributos: `videos`, `audios` e `images`, donde almacena unas rutas locales que es donde se encuentra esos ficheros. El motivo de esto es porque cuando se adjunta un fichero, se genera una copia en el lado del cliente, por lo que podemos generar una ruta fácilmente con el código `URL.createObjectURL(file)`.
Ahora, como queremos poder editar todos los elementos vía web, no nos vale guardar todo en local, sino tener una zona común de trabajo y almacenar los ficheros allí. Para ello, se ha utilizado `Firebase Storage` y se ha creado funciones de ayuda para agilizar la gestión.

```
const uploadFile = async (file: File, folder: string = "uploads"): Promise<string> => {
  const storageRef = ref(storage, `${folder}/${file.name}`);
  try {
    // Upload the file to Firebase Storage
    await uploadBytes(storageRef, file);

    // Get the file's download URL
    const downloadURL = await getDownloadURL(storageRef);
    return downloadURL;
  } catch (error) {
    console.error("Error uploading file:", error);
    throw error;
  }
};

const getFilesFromFolder = async (folder: string = "uploads"): Promise<string[]> => {
  const folderRef = ref(storage, folder);
  try {
    const result = await listAll(folderRef);
    const urls = await Promise.all(result.items.map(async (itemRef) => {
      return getDownloadURLFromRef(itemRef);
    }));
    return urls;
  } catch (error) {
    console.error("Error retrieving files from folder:", error);
    throw error;
  }
};
```

1.7. Sincronización

Llegamos a la parte más importante y quizá el trabajo más tedioso, la sincronización de todo el proyecto. Como idea inicial, como tenemos los ficheros alojados en Firebase Storage, podemos simplemente gestionar el resto de elementos como documentos en `Firebase Database`, y dejamos que Firebase resuelva todo el tema de merge por nosotros, pero como la práctica se centra en soluciones de sincronización, se ha tenido que implementar uno por uno, todos los posibles datos, pero

por cuestión de tiempo, sólo se ha implementado los más importantes que afecta al funcionamiento principal del programa.

1.7.1. Ficheros

Para la sincronización de ficheros, primero debemos definir un subscriptor de escucha de los eventos, porque cuando un usuario añade un fichero a Storage, todos los usuarios también debería tener ese fichero en un local, y por otra parte, como cuando se lance ese subscriptor se cogerá todos los elementos que existe, justo nos sirve para la primera carga de datos cuando un usuario abre un projecto/worskpace. En este código de abajo no existirá un evento de eliminación, porque el repositorio original no se ha implementado y no tuvimos tampoco el tiempo de desarrollo.

```
onSnapshot(collection(db, `projects/${this.projectId}/videos`), (snapshot) => {
  snapshot.docChanges().forEach((change) => {
    const data: string = change.doc.data().url as unknown as string;

    if (change.type === "added") {
      this.addVideoResource(data, false);
      console.log("New animation: ", change.doc.data());
    }
  });
});

onSnapshot(collection(db, `projects/${this.projectId}/audios`), (snapshot) => {
  snapshot.docChanges().forEach((change) => {
    const data: string = change.doc.data().url as unknown as string;

    if (change.type === "added") {
      this.addAudioResource(data, false);
      console.log("New animation: ", change.doc.data());
    }
  });
});

onSnapshot(collection(db, `projects/${this.projectId}/images`), (snapshot) => {
  snapshot.docChanges().forEach((change) => {
    const data: string = change.doc.data().url as unknown as string;

    if (change.type === "added") {
      this.addImageResource(data, false);
      console.log("New animation: ", change.doc.data());
    }
  });
});
```

1.7.2. Animaciones

Antes de entrar a sincronización de elementos, veamos primero sincronización de animación, ya que es un caso más fácil. Según la definición de tipo, la animación contiene siguientes campos:

```
type AnimationBase<T, P = {}> = {
  uid: string | null,
  id: string;
  targetId: string;
  duration: number;
  type: T;
  properties: P;
}
```

Y además, existe una sección propia para la edición de datos, donde todos los valores son de key:value. Con esta facilidad, y como sólo hay X animaciones definidas, es muy complicado tener un problema

de sincronización a tiempo real, porque todos los datos son de un único click o de único cambio, y es prácticamente imposible que 2 personas edite en un mismo momento, por lo que hemos decidido utilizar la configuración de `merge` definida por Firebase para evitar ese problema.
A nivel de implementación, parecido a las animaciones, se ha tenido que crear un subscriptor que esté a la escucha de cualquier cambio, aunque esta vez, con un código ligeramente más complejo

```
onSnapshot(collection(db, `projects/${this.projectId}/animations`), (snapshot) => {
  snapshot.docChanges().forEach((change) => {
    const data: Animation = {
      ...change.doc.data(),
      uid: change.doc.id,
    } as Animation;

    if (change.type === "added") {
      this.addAnimation(data, false);
      console.log("New animation: ", change.doc.data());
    }
    if (change.type === "modified") {
      this.updateAnimation(data.id, data, false);
      console.log("Modified animation: ", change.doc.data());
    }
    if (change.type === "removed") {
      this.removeAnimation(data.id);
      console.log("Removed animation: ", change.doc.data());
    }
  });
});
```

Luego, se debe cambiar ciertas lógicas existentes para subir el cambio a firebase. Como tras subir el cambio se ejecuta un evento del subscriptor, se ha incluido el flag `localChange` para indicar cuál es un cambio local y cuál es un cambio que proviene del remoto para no entrar en un bucle infinito.

```
// Store.ts
async addAnimation(animation: Animation, localChange: boolean = true) {
    // ...
    addAnimationToFirestore(animation, this.projectId);
    this.animations = [...this.animations, animation];
    this.refreshAnimations();
}
updateAnimation(id: string, animation: Animation, localChange: boolean = true) {
    // ...
    uploadAnimationToFirebase(animation, this.projectId);
    const index = this.animations.findIndex((a) => a.id === id);
    this.animations[index] = animation;
    this.refreshAnimations();
}
```

```
// Store.ts
async addAnimation(animation: Animation, localChange: boolean = true) {
    if(!localChange){
      const ele = this.animations.find((e) => e.id === animation.id);
      if(ele){
        return;
      }
    }

    addAnimationToFirestore(animation, this.projectId);
    this.animations = [...this.animations, animation];
    this.refreshAnimations();
}
updateAnimation(id: string, animation: Animation, localChange: boolean = true) {
    if(!localChange){
      const ele = this.animations.find((e) => e.id === animation.id);
      if (!ele) {
```

```
        return;
      }

      const dif = diff(ele, animation);
      if (Object.keys(dif).length === 0) {
        return;
      }
    }

    uploadAnimationToFirebase(animation, this.projectId);
    const index = this.animations.findIndex((a) => a.id === id);
    this.animations[index] = animation;
    this.refreshAnimations();
}

// Utils.ts
const addAnimationToFirestore = async function (animation: Animation, projectId:
    string | null) {
  if (!projectId) {
    console.error('Project ID is null. Cannot add animation to Firestore.');
    return;
  }

  const db = getFirestore();
  const collec = collection(db, `projects/${projectId}/animations`);
  try {
    if (animation.uid == null) {
      const docRef = await addDoc(collec, animation);
      animation.uid = docRef.id;
    } else {
      const docRef = doc(db, `projects/${projectId}/animations`, animation.uid);
      await setDoc(docRef, animation, { merge: true });
    }
  } catch (error) {
    console.error('Error adding animation to Firestore:', error);
    alert('Error synchronizing animation data.');
    return;
  }
};

const uploadAnimationToFirebase = function (animation: Animation, projectId: string |
     null) {
  if (!projectId) {
    console.error('Project ID is null. Cannot upload animation to Firebase.');
    alert('Error: Project ID is missing.');
    return;
  }
  if (animation.uid == null) {
    console.log('Element UID is null');
    return;
  }
  try {
    const db = getFirestore();
    const docRef = doc(db, `projects/${projectId}/animations`, animation.uid);
    updateDoc(docRef, animation as any, { merge: true })
      .then(() => console.log(`Document with UID ${animation.uid} updated
    successfully`))
      .catch((error) => console.error('Error updating document in Firebase:', error))
    ;
  } catch (error) {
    console.error('Error updating document in Firebase:', error);
  }
};
```

1.7.3. Elementos

LLegamos a la parte más compleja, y es compleja por diferentes motivos. La primera es que la estructura de código no es simplemente key:value estandar, sino el value puede ser a su vez un diccionario y a su vez, otro diccionario, por lo tanto, o se crea una sincronización para cada tipo de elemento o habría que generalizar de alguna manera los valores.

```
export type EditorElementBase<T extends string, P> = {
  uid: string | null;
  id: string;
  conflitId: string | null;
  fabricObject?: fabric.Object;
  name: string;
  readonly type: T;
  order: number;
  placement: Placement;
  timeFrame: TimeFrame;
  properties: P;
};

export type EditorElement =
  | VideoEditorElement
  | ImageEditorElement
  | AudioEditorElement
  | TextEditorElement;
```

En nuestro caso, decidimos generalizarlo y se ha creado siguiente código para poder realizar una copia completa y eliminación de ciertos valores innecesarios de un elemento cualquiera.

```
function deepCopy(element: EditorElement): EditorElement {
  switch (element.type) {
    case "video":
      return {
        ...element,
        fabricObject: undefined, // Exclude fabricObject
        properties: {
          ...element.properties,
          imageObject: undefined // Exclude imageObject
        },
        placement: { ...element.placement },
        timeFrame: { ...element.timeFrame },
      } as VideoEditorElement;
    case "image":
      return {
        ...element,
        fabricObject: undefined, // Exclude fabricObject
        properties: {
          ...element.properties,
          imageObject: undefined, // Exclude imageObject
          effect: { ...element.properties.effect }, // Clone the proxy effect
        },
        placement: { ...element.placement },
        timeFrame: { ...element.timeFrame },
      } as ImageEditorElement;
    case "audio":
      return {
        ...element,
        fabricObject: undefined, // Exclude fabricObject
        properties: { ...element.properties },
        placement: { ...element.placement },
        timeFrame: { ...element.timeFrame },
      } as AudioEditorElement;
    case "text":
      return {
```

```
        ...element,
        fabricObject: undefined, // Exclude fabricObject
        properties: {
          ...element.properties,
          splittedTexts: element.properties.splittedTexts.map((text) => ({ ...text }))
    ),
        },
        placement: { ...element.placement },
        timeFrame: { ...element.timeFrame },
      } as TextEditorElement;
    default:
      throw new Error(`Unsupported EditorElement type: ${(element as EditorElement).
    type}`);
  }
}

function removeUndefinedFields(obj: any): any {
  if (Array.isArray(obj)) {
    return obj.map(removeUndefinedFields); // Recursively clean arrays
  } else if (obj && typeof obj === "object") {
    return Object.entries(obj).reduce((acc, [key, value]) => {
      if (value !== undefined) {
        acc[key] = removeUndefinedFields(value); // Recursively clean nested objects
      }
      return acc;
    }, {} as any);
  }
  return obj; // Return primitive values as-is
}
```

El segundo problema es la decisión que se debe tomar cuando múltiples usuarios edite sobre mismo elemento, que a diferencia de las animaciones, una persona puede estar modificando la posición de elemento, una persona modificando el texto de un elemento y una persona modificando el momento de aparición de un elemento.

Para lograr el segundo problema, se ha creado un sistema de `Merge` local, donde los campos que se puede mergear sin problema, se harán de manera automática, pero para los campos donde ambas personas han editado a la vez, se generará un conflicto y creará un track adicional en el usuario afecto. Una vez generado el track adicional, el usuario tiene que eliminar uno de ellos y el sistema se subirá el otro como bueno para la sincronización a todos los usuarios. En el caso de edición sobre un elemento eliminado por otro usuario, por simplicidad, se añade de nuevo ese elemento.

```
// Utils.ts
const mergeField = function (
  element: EditorElement,
  from: EditorElement,
  to: EditorElement,
  fieldName: string,
  diffFrom: Record<string, any>,
  diffTo: Record<string, any>
): boolean {
  if (fieldName in diffFrom && fieldName in diffTo) {
    const diffFieldFrom: Record<string, any> = diff((element as any)[fieldName], (
    from as any)[fieldName]);
    const diffFieldTo: Record<string, any> = diff((element as any)[fieldName], (to as
     any)[fieldName]);
    const combinedDiff: Record<string, any> = diff(diffFieldFrom, diffFieldTo);
    if (Object.keys(combinedDiff).length === 0) {
      for (const key in diffFieldFrom) {
        if (diffFieldFrom[key] !== undefined) {
          (element as any)[fieldName][key] = diffFieldFrom[key];
        }
```

```
      }
      for (const key in diffFieldTo) {
        if (diffFieldTo[key] !== undefined) {
          (element as any)[fieldName][key] = diffFieldTo[key];
        }
      }
    } else {
      return false;
    }
  } else if (fieldName in diffFrom) {
    (element as any)[fieldName] = (from as any)[fieldName];
  } else if (fieldName in diffTo) {
    (element as any)[fieldName] = (to as any)[fieldName];
  }
  return true;
};

const mergeElementUpdate = function (original: EditorElement, from: EditorElement, to
    : EditorElement) {
  const diffFrom: Record<string, any> = diff(original, from);
  const diffTo: Record<string, any> = diff(original, to);
  if ('fabricObject' in diffFrom) {
    delete diffFrom.fabricObject;
  }
  if ('fabricObject' in diffTo) {
    delete diffTo.fabricObject;
  }

  const element = removeUndefinedFields(deepCopy(original));
  const normalChanges = ['order', 'placement', 'timeFrame', 'properties'];
  for (const change of normalChanges) {
    if (
      !mergeField(
        element,
        from,
        to,
        change,
        diffFrom,
        diffTo
      )
    ) {
      return null;
    }
  }

  return element;
};

const mergeElementDelete = function (original: EditorElement, from: EditorElement, to
    : EditorElement, projectId: string | null) {
  addElementToFirestore(to, projectId);
  return to;
};

// Store.ts
mergeElement(original: EditorElement, from: EditorElement, to: EditorElement, type: '
    deleted' | 'updated') {
    if(original === undefined || to === undefined) {
      return from;
    }

    if(type == 'updated'){
      return mergeElementUpdate(original, from, to);
    }else{
```

```
      return mergeElementDelete(original, from, to, this.projectId);
    }
}

setSelectedElement(selectedElement: EditorElement | null) {
    var refresh = false;
    if (this.canvas) {
      if (selectedElement?.fabricObject){
        this.canvas.setActiveObject(selectedElement.fabricObject);
      }
      else{
        if(this.selectedElement != null){
          const element = this.mergeElement(
            this.pendingMerge[this.selectedElement.id]?.from,
            this.selectedElement,
            this.pendingMerge[this.selectedElement.id]?.to,
            this.pendingMerge[this.selectedElement.id]?.type
          );
          if(element){
            if(this.pendingMerge[this.selectedElement.id]){
              delete this.pendingMerge[this.selectedElement.id];
              this.updateEditorElement(element);
              // Refresh to update the element's appearance (remove editor color)
              this.refreshElements();
            }
          }else{
            const ele = removeUndefinedFields(deepCopy(this.selectedElement));
            ele.conflitId = this.selectedElement.id;
            ele.id = getUid();
            ele.name = `${this.selectedElement.name} (conflit)`;
            this.conflit[ele.id] = ele;
            selectedElement = this.pendingMerge[this.selectedElement.id]?.to;
            refresh = true;

            alert("There is a conflict with the element. Pls, review the conflict
    track and delete one of them to sryncronize all data.")
          }
        }
        this.canvas.discardActiveObject();
      }
    }
    this.selectedElement = selectedElement;
    return refresh;
}

async removeEditorElement(id: string | undefined) {
    if (id === undefined) {
      alert("Element ID is undefined");
      return;
    }

    if (this.conflit[id] != undefined) {
      delete this.conflit[id];
      return;
    }

    const elementToRemove = this.editorElements.find(
      (editorElement) => editorElement.id === id
    );

    if (!elementToRemove || !elementToRemove.uid) {
      return;
    }

```

```
    if (!this.projectId) {
      console.error("Project ID is null. Cannot remove element from Firestore.");
      return;
    }

    var hasConflict = false;
    for(const [key, value] of Object.entries(this.conflit)){
      if(value.conflitId == id){
        value.id = elementToRemove.id;
        value.name = elementToRemove.name;
        value.conflitId = null;
        this.updateEditorElement(value);

        hasConflict = true;
        delete this.conflit[key];
        break;
      }
    }
    if(hasConflict){
      return;
    }

    const db = getFirestore();
    const docRef = doc(db, `projects/${this.projectId}/videoEditor`, elementToRemove.
    uid);
    try {
      await deleteDoc(docRef);

      this.setEditorElements(
        this.editorElements.filter(
          (editorElement) => editorElement.id !== id
        )
      );
      this.refreshElements();
    } catch (error) {
      console.error("Error deleting document from Firebase:", error);
      return;
    }
}
```

Una cosa importante que no se ha comentado es que todo el tema de update de un elemento, porque el sistema se comprobará si ese elemento existe un cambios remotos pendiente en mergear, y en función de eso, decidirá publicar o no su cambio. Y tras deseleccionar el elemento con el método `setSelectedElement`, es cuando mergeamos todo y se comprueba los conflictos.

```
updateEditorElement(editorElement: EditorElement, localChange: boolean = true) {
    if(this.conflit[editorElement.id] != undefined){
      this.conflit[editorElement.id] = editorElement;
      return;
    }

    if(this.pendingMerge[editorElement.id] == undefined) {
      if(!localChange){
        const ele = this.editorElements.find((e) => e.id === editorElement.id);
        if (!ele) {
          return;
        }
        const dif = diff(ele, editorElement);
        if ('fabricObject' in dif) {
          delete dif.fabricObject;
        }

        if (Object.keys(dif).length === 0) {
          return;
```

```
        }
      }else{
        uploadElementToFirebase(editorElement, this.projectId);
      }
    }
    this.setEditorElements(this.editorElements.map((element) =>
      element.id === editorElement.id ? editorElement : element
    ));
}
```

Finalmente, para el resto de código ya es muy parecido que otras sincronizaciones, que sería definir los subscriptores de evento. El único cambio diferente es que aquí se comprueba si el elemento remoto es el elemento que está editando ahora mismo el usuario, en el caso de que sí, se guardará ese cambio como cambio pendiente de mergear.

```
onSnapshot(collection(db, `projects/${this.projectId}/videoEditor`), (snapshot) => {
  snapshot.docChanges().forEach((change) => {
    const data = change.doc.data();
    const element: EditorElement = {
      uid: change.doc.id,
      id: data.id,
      conflitId: null,
      name: data.name,
      type: data.type,
      order: data.order,
      placement: data.placement,
      timeFrame: data.timeFrame,
      properties: data.properties,
      editPersonsId: data.editPersonsId,
    };
    if(data.order >= this.order){
      this.order = data.order + 1;
    }
    if (change.type === "added") {
      this.addEditorElement(element, false);
      console.log("New element: ", change.doc.data());
    }
    if (change.type === "modified") {
      if (this.selectedElement?.id === element.id) {
        const dif = diff(this.selectedElement, element);
        if ("fabricObject" in dif) {
          delete (dif as { fabricObject?: unknown }).fabricObject;
        }
        if (Object.keys(dif).length === 0) {
          return;
        }
        if (this.pendingMerge[element.id] == undefined) {
          this.pendingMerge[element.id] = {
            from: this.selectedElement,
            to: element,
            type: "updated",
          };
        } else {
          this.pendingMerge[element.id].to = element;
          this.pendingMerge[element.id].type = "updated";
        }
      } else {
        this.updateEditorElement(element, false);
        console.log("Modified element: ", change.doc.data());
      }
    }
    if (change.type === "removed") {
      if (this.selectedElement?.id === element.id) {
        if (this.pendingMerge[element.id] == undefined) {
```

```
          this.pendingMerge[element.id] = {
            from: this.selectedElement,
            to: element,
            type: "deleted",
          };
        } else {
          this.pendingMerge[element.id].to = element;
          this.pendingMerge[element.id].type = "deleted";
        }
      } else {
        this.removeEditorElement(change.doc.data().id);
      }
    }
  });
});
```

1.7.4. Otros

Para terminar con las sincronizaciones y por falta de tiempo, se ha sincronizado el background del editor y el tiempo máximo del video, que es una implementación práticamente igual que la de animación.

```
// Utils.ts
const addBackgroundToFirestore = async function (background: string, projectId:
    string | null) {
  if (!projectId) {
    console.error('Project ID is null. Cannot update background in Firestore.');
    return;
  }

  const db = getFirestore();
  const projectDocRef = doc(db, `projects/${projectId}`);
  try {
    await updateDoc(projectDocRef, { background });
    console.log('Background updated successfully.');
  } catch (error) {
    console.error('Error updating background in Firestore:', error);
  }
};

const addTimesToFirestore = async function (times: number, projectId: string | null)
    {
  if (!projectId) {
    console.error('Project ID is null. Cannot update times in Firestore.');
    return;
  }

  const db = getFirestore();
  const projectDocRef = doc(db, `projects/${projectId}`);
  try {
    await updateDoc(projectDocRef, { times });
    console.log('Times updated successfully.');
  } catch (error) {
    console.error('Error updating times in Firestore:', error);
  }
};


// Store.ts
const projectDocRef = doc(db, `projects/${this.projectId}`);
onSnapshot(projectDocRef, (docSnapshot) => {
  if (docSnapshot.exists()) {
    const data = docSnapshot.data();
```

```
40
41      if (data.background !== undefined) {
42        this.setBackgroundColor(data.background, false);
43        console.log("Background updated: ", data.background);
44      }
45
46      if (data.times !== undefined) {
47        this.setMaxTime(data.times, false);
48        console.log("Times updated: ", data.times);
49      }
50    } else {
51      console.error("Project document does not exist.");
52    }
53  });
```

2. Hosting

Dado que la aplicación es un proyecto de NextJs hemos decidido realizar el despliegue en Vercel. Vercel es una plataforma de hosting que se integra muy bien con este tipo de proyectos. Ofrece varias funcionalidades entre las cuales, conectarse al repositorio del proyecto en github. Luego en cada cambio subido a la rama principal, se realiza un redespliegue automático.

Desde firebase habrá que configurar la política de CORS para que acepte peticiones desde el dominio proporcionado por Vercel.

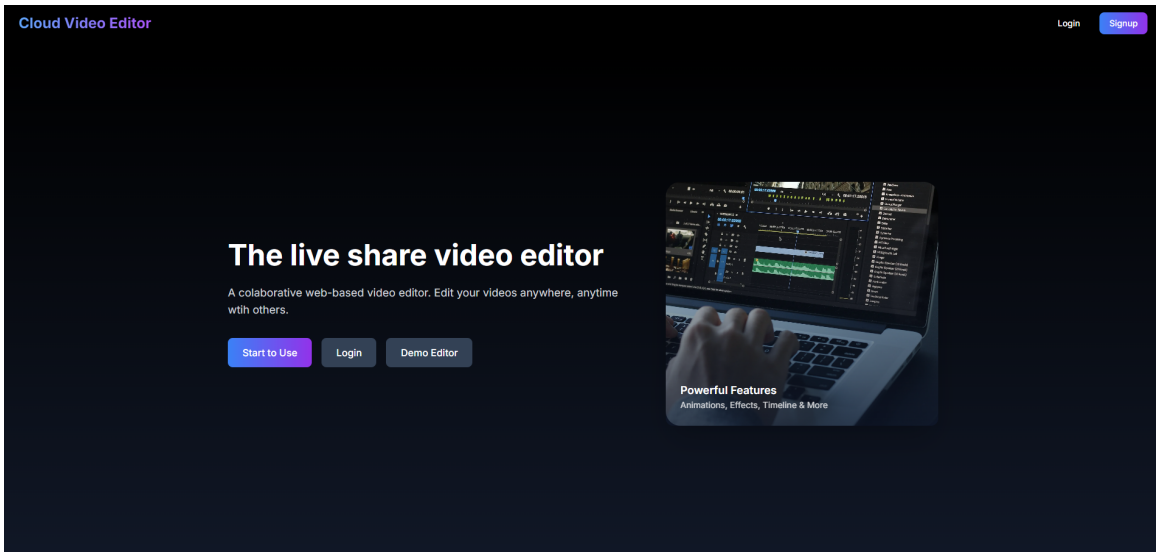

Figura 1: Haz clic en la imagen para visitar Cloud Video Editor o accede desde https://cloud-video-editor.vercel.app

3. Créditos

El código fuente de este proyecto fue modificado y adaptado de este repositorio.