

•

Contents

/

|-

|-

|-

|-

|-

|-

|-

/

|-

|-

|-

|-

|-

/

|-

|-

|-

/

|-

/

/

/

/

|-

|-

|-

|-

/

|-

|-

|-

|-

|-

/

-
/
-
-
-
-
/
-
-
-
/
-
-
/
-
-

convex\authconfig.ts

[to top](#)

```
export default {  
  providers: [  
    {  
      domain: "https://smart-griffon-59.clerk.accounts.dev/",  
      applicationID: "convex",  
    },  
  ],  
};
```

convex\auth.ts

[to top](#)

```

import { mutation, query } from "../_generated/server";
import { v } from "convex/values";
import { Id } from "../_generated/dataModel";
import bcrypt from "bcryptjs";

// We'll use synchronous versions of bcrypt functions to avoid setTimeout
// incompatibility
// These are slower but more compatible with Convex's limitations

// Password management using bcryptjs (synchronous version)
function hashPasswordSync(password: string): string {
  // Use 10 rounds (standard secure value)
  const salt = bcrypt.genSaltSync(10);
  return bcrypt.hashSync(password, salt);
}

// Verify password against stored hash (synchronous version)
function verifyPasswordSync(password: string, hash: string): boolean {
  return bcrypt.compareSync(password, hash);
}

/**
 * Authenticate a user with username and password.
 */
export const login = mutation({
  args: {
    username: v.string(),
    password: v.string(),
  },
  returns: v.object({
    userId: v.id("users"),
    username: v.string(),
  }),
  handler: async (ctx, args) => {
    // Look for an existing user with this username
    const existingUser = await ctx.db
      .query("users")
      .withIndex("by_username", (q) => q.eq("username", args.username))
      .unique();

    // If user exists, verify password
    if (existingUser) {
      // If the user was created before passwords were implemented, they may not have a
      // password
      if (!existingUser.password) {
        // Update the user with a password
        const passwordHash = hashPasswordSync(args.password);
        await ctx.db.patch(existingUser._id, { password: passwordHash });
        return {
          userId: existingUser._id,
          username: existingUser.username,
        };
      }
    }
  }
});

```

```

    }

    // Verify password
    const isValidPassword = verifyPasswordSync(
      args.password,
      existingUser.password
    );
    if (!isValidPassword) {
      throw new Error("Invalid password");
    }

    return {
      userId: existingUser._id,
      username: existingUser.username,
    };
  }
}

// If user doesn't exist, throw an error instead of creating a new account
throw new Error(
  "Account not found. Please register before attempting to sign in."
);
},
});

/**
 * Register a new user
 */
export const register = mutation({
  args: {
    username: v.string(),
    password: v.string(),
  },
  returns: v.object({
    userId: v.id("users"),
    username: v.string(),
  }),
  handler: async (ctx, args) => {
    // Check if username already exists
    const existingUser = await ctx.db
      .query("users")
      .withIndex("by_username", (q) => q.eq("username", args.username))
      .unique();

    if (existingUser) {
      throw new Error("Username already exists");
    }

    // Create a new user
    const passwordHash = hashPasswordSync(args.password);
    const userId = await ctx.db.insert("users", {
      username: args.username,
      password: passwordHash,
      createdAt: Date.now(),
    });
  },
});

```

```

    });

    return {
      userId,
      username: args.username,
    };
  },
});

/**
 * Get the current user if logged in
 */
export const getUser = query({
  args: {
    userId: v.optional(v.id("users")),
  },
  returns: v.union(
    v.object({
      userId: v.id("users"),
      username: v.string(),
    }),
    v.null()
  ),
  handler: async (ctx, args) => {
    if (!args.userId) {
      return null;
    }

    const user = await ctx.db.get(args.userId);
    if (!user) {
      return null;
    }

    return {
      userId: user._id,
      username: user.username,
    };
  },
});

/**
 * Get or create a user with Clerk authentication
 * This function is called when a user signs in with Google via Clerk
 */
export const getUserFromClerk = mutation({
  args: {
    clerkId: v.string(),
    username: v.string(),
  },
  returns: v.object({
    userId: v.id("users"),
    username: v.string(),
  }),
});

```

```
handler: async (ctx, args) => {
  // Check if user with this Clerk ID already exists
  const existingUser = await ctx.db
    .query("users")
    .withIndex("by_clerkId", (q) => q.eq("clerkId", args.clerkId))
    .unique();

  if (existingUser) {
    return {
      userId: existingUser._id,
      username: existingUser.username,
    };
  }

  // Otherwise, create a new user
  const userId = await ctx.db.insert("users", {
    username: args.username,
    clerkId: args.clerkId,
    createdAt: Date.now(),
  });

  return {
    userId,
    username: args.username,
  };
},
});
```

convex\chatts

[to top](#)

```

import { internalAction, mutation, query } from "../_generated/server";
import { v } from "convex/values";

export const sendMessage = mutation({
  args: {
    user: v.string(),
    body: v.string(),
    sessionId: v.optional(v.string()),
  },
  handler: async (ctx, args) => {
    console.log("This TypeScript function is running on the server.");
    await ctx.db.insert("messages", {
      user: args.user,
      body: args.body,
      sessionId: args.sessionId,
    });
  },
});

export const getMessages = query({
  args: {},
  handler: async (ctx) => {
    const messages = await ctx.db.query("messages").order("desc").take(50);

    return messages.reverse();
  },
});

export const getSessionMessages = query({
  args: {
    sessionId: v.string(),
  },
  handler: async (ctx, args) => {
    if (!args.sessionId) {
      return [];
    }

    const messages = await ctx.db
      .query("messages")
      .withIndex("by_sessionId", (q) => q.eq("sessionId", args.sessionId))
      .order("desc")
      .take(100);

    return messages.reverse();
  },
});

// export const getWikipediaSummary = internalAction({
//   args: { topic: v.string() },
//   handler: async (ctx, args) => {
//     const response = await fetch(
//       "https://en.wikipedia.org/w/api.php?

```

```
format=json&action=query&prop=extracts&exintro&explaintext&redirects=1&titles="+  
//      args.topic,  
//    );  
  
//    return getSummaryFromJSON(await response.json());  
//  },  
// });  
  
// function getSummaryFromJSON(data: any) {  
//   const firstPageId = Object.keys(data.query.pages)[0];  
//   return data.query.pages[firstPageId].extract;  
// }
```

convex\eventsts

[to top](#)


```

import { v } from "convex/values";
import { mutation, query } from "../_generated/server";

// Create a new event
export const create = mutation({
  args: {
    userId: v.string(),
    title: v.string(),
    start: v.string(),
    end: v.optional(v.string()),
  },
  handler: async (ctx, args) => {
    // Validate that the start date is not in the past
    const startDate = new Date(args.start);
    const currentDate = new Date();

    console.log("Create Session - Validation:", {
      startDate,
      currentDate,
      isPastDate: startDate < currentDate,
      userId: args.userId,
      title: args.title
    });

    if (startDate < currentDate) {
      console.log("Create Session - Rejected: Past date detected");
      throw new Error("Cannot create session with a past date. Please select a future date.");
    }

    // Check for overlapping sessions for the same user
    const userEvents = await ctx.db
      .query("events")
      .withIndex("by_userId", (q) => q.eq("userId", args.userId))
      .collect();

    // Parse the new session's time range
    const newStart = new Date(args.start).getTime();
    const newEnd = args.end ? new Date(args.end).getTime() : newStart + 60 * 60 * 1000;
    // Default to 1 hour if no end provided

    // Check each existing event for overlap
    for (const event of userEvents) {
      const existingStart = new Date(event.start).getTime();
      const existingEnd = event.end ? new Date(event.end).getTime() : existingStart +
60 * 60 * 1000;

      // Check for overlap:
      // New event starts during existing event OR
      // New event ends during existing event OR
      // New event completely contains existing event
      const overlap = (newStart >= existingStart && newStart < existingEnd) ||

```

```

        (newEnd > existingStart && newEnd <= existingEnd) ||
        (newStart <= existingStart && newEnd >= existingEnd);

    if (overlap) {
        console.log("Create Session - Rejected: Overlapping session detected", {
            newSession: { start: args.start, end: args.end },
            existingSession: { start: event.start, end: event.end, title: event.title }
        });
        throw new Error("Cannot create overlapping sessions. You already have a session
scheduled during this time.");
    }
}

const eventId = await ctx.db.insert("events", {
    userId: args.userId,
    title: args.title,
    start: args.start,
    end: args.end,
    createdAt: Date.now(),
});

console.log("Create Session - Success:", {
    eventId,
    title: args.title,
});

return eventId;
},
});

// Get all events for a user
export const getByUserId = query({
    args: {
        userId: v.string(),
    },
    handler: async (ctx, args) => {
        return await ctx.db
            .query("events")
            .withIndex("by_userId", (q) => q.eq("userId", args.userId))
            .collect();
    },
});

// Get all events from all users with creator username
export const getAllEvents = query({
    args: {},
    handler: async (ctx) => {
        // First, get all users and create a userId -> username map
        const allUsers = await ctx.db.query("users").collect();
        const userMap = new Map();

        // Create a map of user IDs to usernames
        allUsers.forEach((user) => {

```

```

    // Try adding both _id and clerkId to the map
    userMap.set(user._id, user.username);
    if (user.clerkId) {
        userMap.set(user.clerkId, user.username);
    }
});

// Get all events
const events = await ctx.db.query("events").collect();

// Add username to each event
const eventsWithUserInfo = events.map((event) => {
    return {
        ...event,
        creatorName: userMap.get(event.userId) || "Unknown User",
    };
});

return eventsWithUserInfo;
},
});

// Update an existing event
export const update = mutation({
    args: {
        id: v.id("events"),
        title: v.string(),
        start: v.string(),
        end: v.optional(v.string()),
    },
    handler: async (ctx, args) => {
        const event = await ctx.db.get(args.id);
        if (!event) {
            console.log("Update Session - Failed: Event not found", args.id);
            throw new Error("Event not found");
        }

        // Validate that the start date is not in the past
        const startDate = new Date(args.start);
        const currentDate = new Date();

        console.log("Update Session - Validation:", {
            eventId: args.id,
            startDate,
            currentDate,
            isPastDate: startDate < currentDate,
            title: args.title
        });

        if (startDate < currentDate) {
            console.log("Update Session - Rejected: Past date detected");
            throw new Error("Cannot update session to a past date. Please select a future date.");
        }
    }
});

```

```

}

// Check for overlapping sessions for the same user
const userEvents = await ctx.db
  .query("events")
  .withIndex("by_userId", (q) => q.eq("userId", event.userId))
  .collect();

// Parse the updated session's time range
const newStart = new Date(args.start).getTime();
const newEnd = args.end ? new Date(args.end).getTime() : newStart + 60 * 60 * 1000;

// Check each existing event for overlap
for (const existingEvent of userEvents) {
  // Skip checking against the event being updated
  if (existingEvent._id === args.id) continue;

  const existingStart = new Date(existingEvent.start).getTime();
  const existingEnd = existingEvent.end ? new Date(existingEvent.end).getTime() :
existingStart + 60 * 60 * 1000;

  // Check for overlap
  const overlap = (newStart >= existingStart && newStart < existingEnd) ||
    (newEnd > existingStart && newEnd <= existingEnd) ||
    (newStart <= existingStart && newEnd >= existingEnd);

  if (overlap) {
    console.log("Update Session - Rejected: Overlapping session detected", {
      updatedSession: { id: args.id, start: args.start, end: args.end },
      existingSession: { id: existingEvent._id, start: existingEvent.start, end:
existingEvent.end, title: existingEvent.title }
    });
    throw new Error("Cannot create overlapping sessions. You already have a session
scheduled during this time.");
  }
}

await ctx.db.patch(args.id, {
  title: args.title,
  start: args.start,
  end: args.end,
});

console.log("Update Session - Success:", {
  eventId: args.id,
  title: args.title,
});

return args.id;
},
});

// Mutation to join a session (add user to participant list)

```

```

export const joinSession = mutation({
  args: { eventId: v.id("events") },
  handler: async (ctx, args) => {
    const identity = await ctx.auth.getUserIdentity();
    if (!identity) {
      throw new Error("User must be authenticated to join a session.");
    }
    const userId = identity.subject; // Use the subject as the stable user ID

    const event = await ctx.db.get(args.eventId);
    if (!event) {
      throw new Error("Event not found.");
    }

    // Ensure participantIds array exists and add user if not already present
    const currentParticipants = event.participantIds || [];
    if (!currentParticipants.includes(userId)) {
      const updatedParticipants = [...currentParticipants, userId];
      await ctx.db.patch(args.eventId, { participantIds: updatedParticipants });
      console.log(`User ${userId} joined event ${args.eventId}`);
    } else {
      console.log(`User ${userId} is already in event ${args.eventId}`);
    }
  },
});

// Get a single event by ID
export const getEventById = query({
  args: { id: v.string() }, // Changed to string, will validate if it's an ID format
  handler: async (ctx, args) => {
    if (!args.id) {
      console.log("getEventById - No ID provided");
      return null;
    }

    try {
      // First, get all users and create a userId -> username map
      const allUsers = await ctx.db.query("users").collect();
      const userMap = new Map();

      // Create a map of user IDs to usernames
      allUsers.forEach((user) => {
        // Try adding both _id and clerkId to the map
        userMap.set(user._id, user.username);
        if (user.clerkId) {
          userMap.set(user.clerkId, user.username);
        }
      });

      // Get the event by ID
      try {
        const eventId = args.id as any; // Convert to any to avoid type issues
        const event = await ctx.db.get(eventId);
      }
    }
  }
});

```

```

if (!event) {
  console.log("getEventById - Event not found", eventId);
  return null;
}

// Check if it's actually an event by looking for event-specific fields
if ("title" in event && "start" in event) {
  // Log session timing information for debugging
  const now = new Date();
  const startTime = new Date(event.start);
  const endTime = new Date(event.end || event.start);
  const timeUntilStart = startTime.getTime() - now.getTime();

  // Early join threshold - 10 minutes
  const earlyJoinThreshold = 10 * 60 * 1000; // 10 minutes in milliseconds
  const canEarlyJoin = timeUntilStart > 0 && timeUntilStart <=
earlyJoinThreshold;

  console.log("Session timing info:", {
    eventId: event._id,
    title: event.title,
    startTime: startTime.toISOString(),
    endTime: endTime.toISOString(),
    currentTime: now.toISOString(),
    timeUntilStart,
    minutesUntilStart: Math.ceil(timeUntilStart / (1000 * 60)),
    isActive: now >= startTime && now <= endTime,
    canEarlyJoin,
    isPast: now > endTime
  });

  // Add creator name to the event
  return {
    ...event,
    creatorName: userMap.get(event.userId) || "Unknown User",
  };
}

// TODO: Decide if we should return null or the document if it's not an event?
// Returning the document might be confusing if the caller expects an Event
type.
// Returning null might be safer if the caller specifically expects an event.
// For now, let's assume the caller might handle non-event docs, but log a
warning.
else {
  console.warn(`Document found for ID ${eventId} is not an event document.
Type might be ${typeof event}.`);
}

return event; // Return the raw document
} catch (error) {
  console.error("Error getting event:", error);

```

```

        return null;
    }
} catch (error) {
    console.error("Error fetching event by ID:", error);
    return null;
}
},
});

// Delete an event by ID
export const deleteEvent = mutation({
  args: {
    id: v.id("events"),
  },
  handler: async (ctx, args) => {
    const event = await ctx.db.get(args.id);
    if (!event) {
      console.log("Delete Event - Failed: Event not found", args.id);
      throw new Error("Event not found");
    }

    console.log("Delete Event - Deleting:", {
      eventId: args.id,
      title: event.title,
    });

    await ctx.db.delete(args.id);

    console.log("Delete Event - Success: Event deleted", args.id);
    return args.id;
  },
});

```

convex\README.md

[to top](#)

Welcome to your Convex functions directory!

Write your Convex functions here.

See <https://docs.convex.dev/functions> for more.

A query function that takes two arguments looks like:

```
// functions.js
import { query } from "../_generated/server";
import { v } from "convex/values";

export const myQueryFunction = query({
  // Validators for arguments.
  args: {
    first: v.number(),
    second: v.string(),
  },

  // Function implementation.
  handler: async (ctx, args) => {
    // Read the database as many times as you need here.
    // See https://docs.convex.dev/database/reading-data.
    const documents = await ctx.db.query("tablename").collect();

    // Arguments passed from the client are properties of the args object.
    console.log(args.first, args.second);

    // Write arbitrary JavaScript here: filter, aggregate, build derived data,
    // remove non-public properties, or create new objects.
    return documents;
  },
});
```

Using this query function in a React component looks like:

```
const data = useQuery(api.functions.myQueryFunction, {
  first: 10,
  second: "hello",
});
```

A mutation function looks like:


```
// functions.js
import { mutation } from "../_generated/server";
import { v } from "convex/values";

export const myMutationFunction = mutation({
  // Validators for arguments.
  args: {
    first: v.string(),
    second: v.string(),
  },

  // Function implementation.
  handler: async (ctx, args) => {
    // Insert or modify documents in the database here.
    // Mutations can also read from the database like queries.
    // See https://docs.convex.dev/database/writing-data.
    const message = { body: args.first, author: args.second };
    const id = await ctx.db.insert("messages", message);

    // Optionally, return a value from your mutation.
    return await ctx.db.get(id);
  },
});
```

Using this mutation function in a React component looks like:

```
const mutation = useMutation(api.functions.myMutationFunction);
function handleButtonPress() {
  // fire and forget, the most common way to use mutations
  mutation({ first: "Hello!", second: "me" });
  // OR
  // use the result once the mutation has completed
  mutation({ first: "Hello!", second: "me" }).then((result) =>
    console.log(result),
  );
}
```

Use the Convex CLI to push your functions to a deployment. See everything the Convex CLI can do by running `npx convex -h` in your project root directory. To learn more, launch the docs with `npx convex docs`.

convex\schemats

[to top](#)

```

import { defineSchema, defineTable } from "convex/server";
import { v } from "convex/values";

export default defineSchema({
  messages: defineTable({
    user: v.string(),
    body: v.string(),
    sessionId: v.optional(v.string()),
  }).index("by_sessionId", ["sessionId"]),

  users: defineTable({
    username: v.string(),
    password: v.optional(v.string()),
    createdAt: v.number(),
    clerkId: v.optional(v.string()),
  })
    .index("by_username", ["username"])
    .index("by_clerkId", ["clerkId"]),

  events: defineTable({
    userId: v.string(),
    title: v.string(),
    start: v.string(), // Store start time as ISO string
    end: v.optional(v.string()), // Store end time as ISO string, optional
    createdAt: v.number(),
    participantIds: v.optional(v.array(v.string())), // Array of user IDs in the
session
  })
    .index("by_userId", ["userId"])
    .index("by_userId_start", ["userId", "start"]),

  // New table for WebRTC signaling
  videoSignals: defineTable({
    sessionId: v.string(), // ID of the session the signal belongs to
    userId: v.string(), // ID of the user sending the signal
    targetUserId: v.string(), // ID of the user the signal is intended for
    type: v.union( // Type of signal
      v.literal("offer"),
      v.literal("answer"),
      v.literal("candidate")
    ),
    signal: v.string(), // The signaling data (SDP offer/answer or ICE
candidate)
  })
    .index("by_session_and_targetUser", ["sessionId", "targetUserId"])
    .index("by_session_and_user", ["sessionId", "userId"]),
});

```

```

import { v } from "convex/values";
import { mutation, query } from "../_generated/server";

/**
 * Send a WebRTC signaling message (offer, answer, or candidate)
 * to a specific user within a session.
 */
export const sendSignal = mutation({
  args: {
    sessionId: v.string(),
    targetUserId: v.string(),
    type: v.union(
      v.literal("offer"),
      v.literal("answer"),
      v.literal("candidate")
    ),
    signal: v.string(),
  },
  handler: async (ctx, { sessionId, targetUserId, type, signal }) => {
    const identity = await ctx.auth.getUserIdentity();
    if (!identity) {
      throw new Error("User must be authenticated to send signals.");
    }
    const senderUserId = identity.subject; // Get user ID from identity

    // Basic validation (could add checks if users are part of the session)
    if (senderUserId === targetUserId) {
      console.warn("Attempting to send signal to self");
      // Usually, you don't send signals to yourself, but might depend on logic
      // Decide if this should be an error or just logged
    }

    await ctx.db.insert("videoSignals", {
      sessionId,
      userId: senderUserId, // Use the authenticated user's ID as the sender
      targetUserId,
      type,
      signal,
    });

    console.log(`Signal sent: ${type} from ${senderUserId} to ${targetUserId} in session ${sessionId}`);
  },
});

/**
 * Query for WebRTC signaling messages directed at the current user
 * within a specific session.
 */
export const getSignals = query({
  args: {
    sessionId: v.string(),
  },
});

```

```

},
handler: async (ctx, { sessionId }) => {
  const identity = await ctx.auth.getUserIdentity();
  if (!identity) {
    // Return empty array or throw error if user not authenticated?
    // Returning empty might be safer for client-side logic.
    console.warn("Unauthenticated user attempting to get signals.")
    return [];
  }
  const currentUserId = identity.subject; // Get user ID from identity

  // Fetch signals where the targetUserId matches the current authenticated user's ID
  const signals = await ctx.db
    .query("videoSignals")
    .withIndex("by_session_and_targetUser", (q) =>
      q.eq("sessionId", sessionId).eq("targetUserId", currentUserId)
    )
    .collect();

  return signals;
},
});

/**
 * Mutation to delete signals once they have been processed by the recipient.
 * This prevents reprocessing old signals.
 */
export const deleteSignal = mutation({
  args: { signalId: v.id("videoSignals") },
  handler: async (ctx, { signalId }) => {
    const identity = await ctx.auth.getUserIdentity();
    if (!identity) {
      throw new Error("User must be authenticated to delete signals.");
    }
    const currentUserId = identity.subject;

    // Fetch the signal document first
    const signal = await ctx.db.get(signalId);

    // Only attempt deletion if the signal exists
    if (signal !== null) {
      // Optional: You could re-add the check here if needed:
      // if (signal.targetUserId !== currentUserId) {
      //   console.warn(`User ${currentUserId} attempted to delete signal
      ${signalId} not targeted at them.`);
      //   // Decide whether to throw an error or just log and exit
      //   return;
      // }

      await ctx.db.delete(signalId);
      console.log(`Signal ${signalId} deleted by user ${currentUserId}`);
    } else {
      console.log(`Signal ${signalId} not found, likely already deleted.`);
    }
  }
});

```

```
    }  
  }  
});
```

convex_generated\apid.ts

[to top](#)

```

/* eslint-disable */
/**
 * Generated `api` utility.
 *
 * THIS CODE IS AUTOMATICALLY GENERATED.
 *
 * To regenerate, run `npx convex dev`.
 * @module
 */

import type {
  ApiFromModules,
  FilterApi,
  FunctionReference,
} from "convex/server";
import type * as auth from "../auth.js";
import type * as chat from "../chat.js";
import type * as events from "../events.js";
import type * as video from "../video.js";

/**
 * A utility for referencing Convex functions in your app's API.
 *
 * Usage:
 * ```js
 * const myFunctionReference = api.myModule.myFunction;
 * ```
 */
declare const fullApi: ApiFromModules<{
  auth: typeof auth;
  chat: typeof chat;
  events: typeof events;
  video: typeof video;
}>;
export declare const api: FilterApi<
  typeof fullApi,
  FunctionReference<any, "public">
>;
export declare const internal: FilterApi<
  typeof fullApi,
  FunctionReference<any, "internal">
>;

```

convex_generated\apijs

[to top](#)

```
/* eslint-disable */
/**
 * Generated `api` utility.
 *
 * THIS CODE IS AUTOMATICALLY GENERATED.
 *
 * To regenerate, run `npx convex dev`.
 * @module
 */

import { anyApi } from "convex/server";

/**
 * A utility for referencing Convex functions in your app's API.
 *
 * Usage:
 * ```js
 * const myFunctionReference = api.myModule.myFunction;
 * ```
 */
export const api = anyApi;
export const internal = anyApi;
```

convex_generated\dataModeld.ts

[to top](#)

```

/* eslint-disable */
/**
 * Generated data model types.
 *
 * THIS CODE IS AUTOMATICALLY GENERATED.
 *
 * To regenerate, run `npx convex dev`.
 * @module
 */

import type {
  DataModelFromSchemaDefinition,
  DocumentByName,
  TableNamesInDataModel,
  SystemTableNames,
} from "convex/server";
import type { GenericId } from "convex/values";
import schema from "../schema.js";

/**
 * The names of all of your Convex tables.
 */
export type TableNames = TableNamesInDataModel<DataModel>;

/**
 * The type of a document stored in Convex.
 *
 * @typeParam TableName - A string literal type of the table name (like "users").
 */
export type Doc<TableName extends TableNames> = DocumentByName<
  DataModel,
  TableName
>;

/**
 * An identifier for a document in Convex.
 *
 * Convex documents are uniquely identified by their `Id`, which is accessible
 * on the `_id` field. To learn more, see [Document IDs]
 * (https://docs.convex.dev/using/document-ids).
 *
 * Documents can be loaded using `db.get(id)` in query and mutation functions.
 *
 * IDs are just strings at runtime, but this type can be used to distinguish them from
 * other
 * strings when type checking.
 *
 * @typeParam TableName - A string literal type of the table name (like "users").
 */
export type Id<TableName extends TableNames | SystemTableNames> =
  GenericId<TableName>;

```



```
/**
 * A type describing your Convex data model.
 *
 * This type includes information about what tables you have, the type of
 * documents stored in those tables, and the indexes defined on them.
 *
 * This type is used to parameterize methods like `queryGeneric` and
 * `mutationGeneric` to make them type-safe.
 */
export type DataModel = DataModelFromSchemaDefinition<typeof schema>;
```

convex_generated\serverd.ts

[to top](#)

```

/* eslint-disable */
/**
 * Generated utilities for implementing server-side Convex query and mutation
functions.
 *
 * THIS CODE IS AUTOMATICALLY GENERATED.
 *
 * To regenerate, run `npx convex dev`.
 * @module
 */

import {
  ActionBuilder,
  HttpActionBuilder,
  MutationBuilder,
  QueryBuilder,
  GenericActionCtx,
  GenericMutationCtx,
  GenericQueryCtx,
  GenericDatabaseReader,
  GenericDatabaseWriter,
} from "convex/server";
import type { DataModel } from "../dataModel.js";

/**
 * Define a query in this Convex app's public API.
 *
 * This function will be allowed to read your Convex database and will be accessible
from the client.
 *
 * @param func - The query function. It receives a {@link QueryCtx} as its first
argument.
 * @returns The wrapped query. Include this as an `export` to name it and make it
accessible.
 */
export declare const query: QueryBuilder<DataModel, "public">;

/**
 * Define a query that is only accessible from other Convex functions (but not from the
client).
 *
 * This function will be allowed to read from your Convex database. It will not be
accessible from the client.
 *
 * @param func - The query function. It receives a {@link QueryCtx} as its first
argument.
 * @returns The wrapped query. Include this as an `export` to name it and make it
accessible.
 */
export declare const internalQuery: QueryBuilder<DataModel, "internal">;

/**

```

```

* Define a mutation in this Convex app's public API.
*
* This function will be allowed to modify your Convex database and will be accessible
from the client.
*
* @param func - The mutation function. It receives a {@link MutationCtx} as its first
argument.
* @returns The wrapped mutation. Include this as an `export` to name it and make it
accessible.
*/
export declare const mutation: MutationBuilder<DataModel, "public">;

/**
* Define a mutation that is only accessible from other Convex functions (but not from
the client).
*
* This function will be allowed to modify your Convex database. It will not be
accessible from the client.
*
* @param func - The mutation function. It receives a {@link MutationCtx} as its first
argument.
* @returns The wrapped mutation. Include this as an `export` to name it and make it
accessible.
*/
export declare const internalMutation: MutationBuilder<DataModel, "internal">;

/**
* Define an action in this Convex app's public API.
*
* An action is a function which can execute any JavaScript code, including non-
deterministic
* code and code with side-effects, like calling third-party services.
* They can be run in Convex's JavaScript environment or in Node.js using the "use
node" directive.
* They can interact with the database indirectly by calling queries and mutations
using the {@link ActionCtx}.
*
* @param func - The action. It receives an {@link ActionCtx} as its first argument.
* @returns The wrapped action. Include this as an `export` to name it and make it
accessible.
*/
export declare const action: ActionBuilder<DataModel, "public">;

/**
* Define an action that is only accessible from other Convex functions (but not from
the client).
*
* @param func - The function. It receives an {@link ActionCtx} as its first argument.
* @returns The wrapped function. Include this as an `export` to name it and make it
accessible.
*/
export declare const internalAction: ActionBuilder<DataModel, "internal">;

```

```

/**
 * Define an HTTP action.
 *
 * This function will be used to respond to HTTP requests received by a Convex
 * deployment if the requests matches the path and method where this action
 * is routed. Be sure to route your action in `convex/http.js`.
 *
 * @param func - The function. It receives an {@link ActionCtx} as its first argument.
 * @returns The wrapped function. Import this function from `convex/http.js` and route
it to hook it up.
 */
export declare const httpAction: HttpActionBuilder;

/**
 * A set of services for use within Convex query functions.
 *
 * The query context is passed as the first argument to any Convex query
 * function run on the server.
 *
 * This differs from the {@link MutationCtx} because all of the services are
 * read-only.
 */
export type QueryCtx = GenericQueryCtx<DataModel>;

/**
 * A set of services for use within Convex mutation functions.
 *
 * The mutation context is passed as the first argument to any Convex mutation
 * function run on the server.
 */
export type MutationCtx = GenericMutationCtx<DataModel>;

/**
 * A set of services for use within Convex action functions.
 *
 * The action context is passed as the first argument to any Convex action
 * function run on the server.
 */
export type ActionCtx = GenericActionCtx<DataModel>;

/**
 * An interface to read from the database within Convex query functions.
 *
 * The two entry points are {@link DatabaseReader.get}, which fetches a single
 * document by its {@link Id}, or {@link DatabaseReader.query}, which starts
 * building a query.
 */
export type DatabaseReader = GenericDatabaseReader<DataModel>;

/**
 * An interface to read from and write to the database within Convex mutation
 * functions.
 *

```

```
* Convex guarantees that all writes within a single mutation are
* executed atomically, so you never have to worry about partial writes leaving
* your data in an inconsistent state. See [the Convex Guide]
(https://docs.convex.dev/understanding/convex-fundamentals/functions#atomicity-and-optimistic-concurrency-control)
* for the guarantees Convex provides your functions.
*/
export type DatabaseWriter = GenericDatabaseWriter<DataModel>;
```

convex_generated\serverjs

[to top](#)

```

/* eslint-disable */
/**
 * Generated utilities for implementing server-side Convex query and mutation
functions.
 *
 * THIS CODE IS AUTOMATICALLY GENERATED.
 *
 * To regenerate, run `npx convex dev`.
 * @module
 */

import {
  actionGeneric,
  httpActionGeneric,
  queryGeneric,
  mutationGeneric,
  internalActionGeneric,
  internalMutationGeneric,
  internalQueryGeneric,
} from "convex/server";

/**
 * Define a query in this Convex app's public API.
 *
 * This function will be allowed to read your Convex database and will be accessible
from the client.
 *
 * @param func - The query function. It receives a {@link QueryCtx} as its first
argument.
 * @returns The wrapped query. Include this as an `export` to name it and make it
accessible.
 */
export const query = queryGeneric;

/**
 * Define a query that is only accessible from other Convex functions (but not from the
client).
 *
 * This function will be allowed to read from your Convex database. It will not be
accessible from the client.
 *
 * @param func - The query function. It receives a {@link QueryCtx} as its first
argument.
 * @returns The wrapped query. Include this as an `export` to name it and make it
accessible.
 */
export const internalQuery = internalQueryGeneric;

/**
 * Define a mutation in this Convex app's public API.
 *
 * This function will be allowed to modify your Convex database and will be accessible

```

```

from the client.
*
* @param func - The mutation function. It receives a {@link MutationCtx} as its first
argument.
* @returns The wrapped mutation. Include this as an `export` to name it and make it
accessible.
*/
export const mutation = mutationGeneric;

/**
 * Define a mutation that is only accessible from other Convex functions (but not from
the client).
 *
 * This function will be allowed to modify your Convex database. It will not be
accessible from the client.
 *
 * @param func - The mutation function. It receives a {@link MutationCtx} as its first
argument.
 * @returns The wrapped mutation. Include this as an `export` to name it and make it
accessible.
 */
export const internalMutation = internalMutationGeneric;

/**
 * Define an action in this Convex app's public API.
 *
 * An action is a function which can execute any JavaScript code, including non-
deterministic
 * code and code with side-effects, like calling third-party services.
 * They can be run in Convex's JavaScript environment or in Node.js using the "use
node" directive.
 * They can interact with the database indirectly by calling queries and mutations
using the {@link ActionCtx}.
 *
 * @param func - The action. It receives an {@link ActionCtx} as its first argument.
 * @returns The wrapped action. Include this as an `export` to name it and make it
accessible.
 */
export const action = actionGeneric;

/**
 * Define an action that is only accessible from other Convex functions (but not from
the client).
 *
 * @param func - The function. It receives an {@link ActionCtx} as its first argument.
 * @returns The wrapped function. Include this as an `export` to name it and make it
accessible.
 */
export const internalAction = internalActionGeneric;

/**
 * Define a Convex HTTP action.
 *

```

```
* @param func - The function. It receives an {@link ActionCtx} as its first argument,
and a `Request` object
* as its second.
* @returns The wrapped endpoint function. Route a URL path to this function in
`convex/http.js`.
*/
export const httpAction = httpActionGeneric;
```

dist\indexhtml

[to top](#)

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
    <title>Convex Chat</title>
    <script type="module" crossorigin src="/assets/index-S7rK6oc0.js"></script>
    <link rel="stylesheet" crossorigin href="/assets/index-Ca1NoLCf.css">
  </head>
  <body>
    <div id="root"></div>
  </body>
</html>
```

indexhtml

[to top](#)

```
<!doctype html>
<html lang="en">
  <head>
    <meta charset="UTF-8" />
    <title>Workoutmate</title>
    <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  </head>
  <body>
    <div id="root"></div>
    <script type="module" src="/src/main.tsx"></script>
  </body>
</html>
```

README.md

[to top](#)

Workoutmate

A workout companion app powered by Convex.

Follow the tutorial at

docs.convex.dev/tutorial (<https://docs.convex.dev/tutorial>) for instructions.

src\Apptsx

[to top](#)

```

import { useEffect } from "react";
import { Routes, Route, useLocation, useNavigate } from "react-router-dom";
import { Login, ProtectedRoute, OAuthCallback } from "../components/auth";
import { Calendar } from "../components/calendar";
import { Session } from "../components/session";
import { useUser } from "@clerk/clerk-react";

export default function App() {
  const location = useLocation();
  const navigate = useNavigate();
  const { isLoading: clerkLoaded, isSignedIn, user } = useUser();

  // When auth state changes, redirect appropriately
  useEffect(() => {
    // Skip redirects if Clerk is still loading or we're on the OAuth callback page
    if (!clerkLoaded || location.pathname === "/oauth-callback") {
      return;
    }

    if (!isSignedIn) {
      // Only redirect if we're not already on a auth page
      if (
        location.pathname !== "/login" &&
        location.pathname !== "/register" &&
        location.pathname !== "/oauth-callback"
      ) {
        console.log("Not authenticated, redirecting to login");
        navigate("/login");
      }
    } else if (
      location.pathname === "/login" ||
      location.pathname === "/register"
    ) {
      // If logged in but on auth page, redirect to home
      console.log("Already authenticated, redirecting to home");
      navigate("/");
    }
  }, [isSignedIn, navigate, location.pathname, clerkLoaded]);

  // Main route structure
  return (
    <Routes>
      <Route path="/login" element={<Login isRegistering={false} />} />
      <Route path="/register" element={<Login isRegistering={true} />} />
      <Route path="/oauth-callback" element={<OAuthCallback />} />
      <Route
        path="/"
        element={
          <ProtectedRoute>
            <Calendar />
          </ProtectedRoute>
        }
      />
    </Routes>
  );
}

```

```
    />
    <Route
      path="/session/:sessionId"
      element={
        <ProtectedRoute>
          <Session />
        </ProtectedRoute>
      }
    />
  </Routes>
);
}
```

src\components\auth\index.ts

[to top](#)

```
export { Login } from "./Login";
export { ProtectedRoute } from "./ProtectedRoute";
export { OAuthCallback } from "./OAuthCallback";
```

src\components\auth\Logintsx

[to top](#)

```

import { useState, useEffect } from "react";
import { useNavigate } from "react-router-dom";
import styles from "../Login.module.css";
import workoutImage from "../../assets/images/workoutmate.webp";
import { useSignIn, useClerk, useUser, SignIn, SignUp } from "@clerk/clerk-react";

interface LoginProps {
  isRegistering: boolean;
}

export function Login({ isRegistering }: LoginProps) {
  const [isProcessingGoogle, setIsProcessingGoogle] = useState(false);
  const navigate = useNavigate();
  const { signIn, isLoading: clerkLoaded } = useSignIn();
  const clerk = useClerk();
  const { isSignedIn } = useUser();

  const handleGoogleSignIn = async () => {
    try {
      if (!clerkLoaded || !signIn) {
        console.error("Clerk not ready for Google Sign-In");
        return;
      }

      setIsProcessingGoogle(true);
      console.log("Starting Google sign-in process");

      if (clerk.session) {
        console.log("Clearing existing session");
        await clerk.signOut();
      }

      console.log("Redirecting to Google OAuth");
      await signIn.authenticateWithRedirect({
        strategy: "oauth_google",
        redirectUrl: window.location.origin + "/oauth-callback",
        redirectUrlComplete: window.location.origin + "/", // Redirect to home after
completion
      });
    } catch (error) {
      console.error("Google sign-in failed:", error);
      setIsProcessingGoogle(false);
    }
  };

  return (
    <div className={styles.loginContainer}>
      <div className={styles.loginWrapper}>
        <div className={styles.loginCard}>
          <h1>Workoutmate</h1>

          {isRegistering ? (

```

```

    <SignUp
      path="/register"
      routing="path"
      signInUrl="/login" // URL to navigate to for sign-in
      forceRedirectUrl="/" // Redirect after successful sign-up
    />
  ) : (
    <SignIn
      path="/login"
      routing="path"
      signUpUrl="/register" // URL to navigate to for sign-up
      forceRedirectUrl="/" // Redirect after successful sign-in
    />
  )}

  {/* <div className={styles.divider}>
    <span>OR</span>
  </div>

  <button
    onClick={handleGoogleSignIn}
    className={` ${styles.loginButton} ${styles.googleButton}`}
    disabled={!clerkLoaded || isProcessingGoogle}
  >
    {isProcessingGoogle ? "Processing..." : "Continue with Google"}
  </button> */}
</div>
<div className={styles.imageContainer}>
  <img src={workoutImage} alt="Workout illustration" className=
{styles.workoutImage} />
</div>
</div>
</div>
);
}

```

src\components\auth\OAuthCallbacktsx

[to top](#)

```

import { useEffect, useState } from "react";
import { useNavigate } from "react-router-dom";
import { useAuth } from "../../contexts/AuthContext";
import { useClerk, useUser } from "@clerk/clerk-react";
import { useMutation } from "convex/react";
import { api } from "../../convex/_generated/api";

export function OAuthCallback() {
  const navigate = useNavigate();
  const { isSignedIn, user, isLoading: userIsLoaded } = useUser();
  const { login, isAuthenticated } = useAuth();
  const getUserFromClerk = useMutation(api.auth.getUserFromClerk);
  const clerk = useClerk();
  const [isProcessing, setIsProcessing] = useState(true);
  const [error, setError] = useState<string | null>(null);
  const [statusMessage, setStatusMessage] = useState("Initializing authentication...");

  useEffect(() => {
    // Debug logging to track the flow
    console.log("OAuth callback - Auth state:", {
      isSignedIn,
      isLoading: userIsLoaded,
      hasUser: !!user,
      userId: user?.id,
      isAuthenticatedInContext: isAuthenticated
    });
  });

  async function handleOAuthCallback() {
    if (!userIsLoaded) {
      setStatusMessage("Loading user data...");
      return; // Wait for user data to load
    }

    // If already authenticated in our context, we can navigate directly
    if (isAuthenticated) {
      console.log("Already authenticated in context, navigating to home");
      setStatusMessage("Already authenticated, redirecting...");
      setTimeout(() => navigate("/"), 500);
      return;
    }

    try {
      setIsProcessing(true);

      if (isSignedIn && user) {
        setStatusMessage("User authenticated with Clerk, creating Convex user...");

        // Get email or fall back to username
        const userIdentifier = user.emailAddresses.length > 0
          ? user.emailAddresses[0].emailAddress
          : (user.username || `user-${user.id}`);
      }
    }
  }
}

```

```

    console.log("Creating/getting Convex user with identifier:", userIdentifier);

    // Create or get user in Convex
    const convexUser = await getUserFromClerk({
      clerkId: user.id,
      username: userIdentifier,
    });

    console.log("Convex user created/retrieved:", convexUser);

    // Login with AuthContext
    login(convexUser.userId, convexUser.username);
    setStatusMessage("Authentication successful, redirecting...");

    // Redirect to the home page with a slight delay to ensure state updates
    setTimeout(() => navigate("/"), 1000);
  } else if (userIsLoaded && !isSignedIn) {
    // User loaded but not signed in
    console.log("User data loaded but not signed in");
    throw new Error("OAuth authentication failed - Not signed in after loading user data");
  }
} catch (error) {
  console.error("Error in OAuth callback:", error);
  setError("Authentication failed. Please try again.");
  setStatusMessage("Authentication error, redirecting to login...");
  setTimeout(() => navigate("/login"), 2000);
} finally {
  setIsProcessing(false);
}
}

handleOAuthCallback();
}, [isSignedIn, user, userIsLoaded, login, getUserFromClerk, navigate, clerk, isAuthenticated]));

if (error) {
  return (
    <div style={{
      display: 'flex',
      justifyContent: 'center',
      alignItems: 'center',
      height: '100vh',
      flexDirection: 'column'
    }}>
      <h2>Authentication Error</h2>
      <p>{error}</p>
      <p>Redirecting you back to login...</p>
    </div>
  );
}

return (

```

```

    <div style={{
      display: 'flex',
      justifyContent: 'center',
      alignItems: 'center',
      height: '100vh',
      flexDirection: 'column'
    }}>
      <h2>Completing login...</h2>
      <p>{statusMessage}</p>
    </div>
  );
}

```

src\components\auth\ProtectedRoutes.tsx

[to top](#)

```

import { Navigate } from "react-router-dom";
import { useUser } from "@clerk/clerk-react";

interface ProtectedRouteProps {
  children: React.ReactNode;
}

export const ProtectedRoute = ({ children }: ProtectedRouteProps) => {
  const { isLoading, isSignedIn } = useUser();

  if (!isLoading) {
    return <div>Loading authentication...</div>;
  }

  if (!isSignedIn) {
    return <Navigate to="/login" replace />;
  }

  return <>{children}</>;
};

```

src\components\calendar\Calendar.tsx

[to top](#)


```

import { useEffect, useState } from "react";
import FullCalendar from "@fullcalendar/react";
import dayGridPlugin from "@fullcalendar/daygrid";
import timeGridPlugin from "@fullcalendar/timegrid";
import interactionPlugin, { DateClickArg } from "@fullcalendar/interaction";
import { Header } from "../layout";
import { useUser } from "@clerk/clerk-react";
import { useMutation, useQuery } from "convex/react";
import { api } from "../../convex";
import { CalendarOptions } from "@fullcalendar/core";
import { CustomEvent } from "../CustomEvent";
import { EventModal } from "../EventModal";
import { SessionDetailsModal } from "../SessionDetailsModal";
import { Id } from "../../convex/_generated/dataModel";
import { showToast } from "../../utils/toast";
import styles from "../Calendar.module.css";

interface CalendarProps {}

interface CalendarEvent {
  id: Id<"events">;
  title: string;
  start: string;
  end: string;
  creatorName: string;
}

export function Calendar({}: CalendarProps) {
  const { isLoading, isSignedIn, user } = useUser();

  const currentUserId = isLoading && isSignedIn ? user.id : null;
  const currentUsername = isLoading && isSignedIn ? user.username : null;

  const [isModalOpen, setIsModalOpen] = useState(false);
  const [isDetailsModalOpen, setIsDetailsModalOpen] = useState(false);
  const [selectedDate, setSelectedDate] = useState<string>("");
  const [selectedEvent, setSelectedEvent] = useState<CalendarEvent | null>(
    null
  );
  const [viewingEvent, setViewingEvent] = useState<CalendarEvent | null>(null);

  // Use Convex to manage events
  const createEvent = useMutation(api.events.create);
  const updateEvent = useMutation(api.events.update);
  const deleteEvent = useMutation(api.events.deleteEvent);
  const allEvents = useQuery(api.events.getAllEvents) || [];

  // Transform Convex events to the format FullCalendar expects
  const events = allEvents.map((event: any) => ({
    id: event._id,
    creatorName: event.creatorName, // Keep the original creator name
    // Include creator name with the title, show "You" for current user's events

```

```

    title:
      event.creatorName === currentUsername
        ? `You: ${event.title}`
        : event.creatorName && event.creatorName !== "Unknown User"
          ? `${event.creatorName}: ${event.title}`
          : event.title,
    start: event.start,
    end: event.end,
  }));

// Handle date click - open modal for new event creation
const handleDateClick = (info: DateClickArg) => {
  const clickedDate = new Date(info.dateStr);
  const now = new Date();

  // Check if the clicked date is in the past
  if (clickedDate < now) {
    console.log("Calendar - Rejected date click: Past date detected", info.dateStr);
    showToast.session.pastDateError();
    return; // Early return without opening the modal
  }

  // Only executed for future dates
  setSelectedEvent(null); // Clear any selected event
  setSelectedDate(info.dateStr);
  setIsModalOpen(true);
};

// Handle event click - show details modal for all events
const handleEventClick = (info: any) => {
  const event = events.find((e) => e.id === info.event.id);
  if (!event) return;

  const [, ...titleParts] = event.title.split(": ");
  const eventTitle =
    titleParts.length > 0 ? titleParts.join(": ") : event.title;
  const eventData = {
    id: event.id,
    title: eventTitle,
    start: event.start,
    end: event.end,
    creatorName: event.creatorName,
  };

  setViewingEvent(eventData);
  setIsDetailsModalOpen(true);
};

// Handle direct edit click from the event
const handleEventEditClick = (info: any) => {
  const event = events.find((e) => e.id === info.event.id);
  if (!event) return;

```

```

const [, ...titleParts] = event.title.split(": ");
const eventTitle =
  titleParts.length > 0 ? titleParts.join(": ") : event.title;
const eventData = {
  id: event.id,
  title: eventTitle,
  start: event.start,
  end: event.end,
  creatorName: event.creatorName,
};

setSelectedEvent(eventData);
setIsModalOpen(true);
};

// Handle switching from details to edit mode
const handleEditEvent = () => {
  if (viewingEvent) {
    setSelectedEvent(viewingEvent);
    setIsDetailsModalOpen(false);
    setIsModalOpen(true);
  }
};

// Handle event deletion
const handleEventDelete = async (eventId: Id<"events">) => {
  try {
    console.log("Calendar - Deleting event:", eventId);
    await deleteEvent({ id: eventId });
    console.log("Calendar - Event deleted successfully");
    showToast.session.deleted();
    return Promise.resolve();
  } catch (error: any) {
    console.error("Calendar - Error deleting event:", error);
    showToast.error(error.message || "Failed to delete session");
    return Promise.reject(error);
  }
};

// Handle event creation/update from modal
const handleEventSubmit = async ({
  title,
  start,
  end,
}: {
  title: string;
  start: string;
  end: string;
}): Promise<void> => {
  // Add guard clause for authentication
  if (!isSignedIn || !user) {
    console.error("User not authenticated, cannot submit event.");
    showToast.error("Authentication error, please log in again.");
  }
};

```

```

    return;
}

try {
  console.log("Calendar - Submitting event:", {
    isEdit: !!selectedEvent?.id,
    title,
    start,
    end
  });

  if (selectedEvent?.id) {
    // Update existing event
    console.log("Calendar - Updating existing event:", selectedEvent.id);
    await updateEvent({
      id: selectedEvent.id,
      title,
      start,
      end,
    });
    console.log("Calendar - Event updated successfully");
    showToast.session.updated();
  } else if (user) {
    // Create new event
    console.log("Calendar - Creating new event for user:", user.id);
    const eventId = await createEvent({
      userId: user.id,
      title,
      start,
      end,
    });
    console.log("Calendar - Event created successfully:", eventId);
    showToast.session.created();
  }
} catch (error: any) {
  console.error("Calendar - Error submitting event:", error);
  if (error.message.includes("past date")) {
    showToast.session.pastDateError();
  } else if (error.message.includes("overlapping")) {
    showToast.session.overlapError();
  } else {
    showToast.error(error.message || "Failed to create/update session");
  }
  return Promise.reject(error);
}
};

// Configure calendar options
const calendarOptions: CalendarOptions = {
  plugins: [dayGridPlugin, timeGridPlugin, interactionPlugin],
  initialView: "timeGridWeek",
  weekends: true,
  events: events,

```

```

height: "100%",
headerToolbar: {
  left: "prev,next today",
  center: "title",
  right: "dayGridMonth,timeGridWeek,timeGridDay",
},
editable: true,
selectable: true,
dateClick: handleDateClick,
eventClick: handleEventClick,
nowIndicator: true,
scrollTime: "08:00:00",
// Additional time slot configuration options
slotDuration: "00:15:00", // 30-minute slots (default)
slotLabelInterval: "01:00:00", // Show labels every hour
slotMinTime: "00:00:00", // Start at midnight
slotMaxTime: "24:00:00", // End at midnight
eventContent: (arg) => {
  const timeText =
    arg.timeText ||
    new Date(arg.event.startStr).toLocaleTimeString([], {
      hour: "2-digit",
      minute: "2-digit",
    });
  const event = events.find((e) => e.id === arg.event.id);
  const isCurrentUser = event?.creatorName === currentUsername;
  return (
    <CustomEvent
      event={{ title: arg.event.title, timeText }}
      isCurrentUser={isCurrentUser}
      onEditClick={() => handleEventEditClick(arg)}
    />
  );
},
};

// Handle loading state from Clerk
if (!isLoading) {
  return (
    <div className={styles.calendarContainer}>
      <Header />
      <div className={styles.loading}>Loading user information...</div>
    </div>
  );
}

// Handle unauthenticated state (though ProtectedRoute should prevent this)
if (!isSignedIn) {
  return (
    <div className={styles.calendarContainer}>
      <Header />
      <div className={styles.loading}>Please log in to view the calendar.</div>
    </div>
  );
}

```

```

    );
}

return (
    <div className={styles.calendarContainer}>
        <Header />
        <main className={styles.calendar}>
            <FullCalendar {...calendarOptions} />
        </main>
        <EventModal
            isOpen={isModalOpen}
            onClose={() => {
                setIsModalOpen(false);
                setSelectedEvent(null);
            }}
            onSubmit={handleEventSubmit}
            onDelete={selectedEvent?.creatorName === currentUsername ? handleEventDelete :
undefined}
            dateStr={selectedDate}
            event={selectedEvent}
            userId={user?.id ?? ''}
        />
        <SessionDetailsModal
            isOpen={isDetailsModalOpen}
            onClose={() => {
                setIsDetailsModalOpen(false);
                setViewingEvent(null);
            }}
            event={viewingEvent}
            isOwnEvent={viewingEvent?.creatorName === currentUsername}
            onEdit={handleEditEvent}
        />
    </div>
);
};

```

src\components\calendar\CustomEventtsx

[to top](#)

```

import React from "react";
import styles from "../CustomEvent.module.css";

interface CustomEventProps {
  event: {
    title: string;
    timeText: string;
  };
  isCurrentUser: boolean;
  onEditClick?: (e: React.MouseEvent) => void;
}

export const CustomEvent: React.FC<CustomEventProps> = ({
  event,
  isCurrentUser,
  onEditClick,
}) => {
  // Split the title to separate creator name and event title if present
  const [creatorName, ...titleParts] = event.title.split(": ");
  const eventTitle =
    titleParts.length > 0 ? titleParts.join(": ") : creatorName;

  const handleEditClick = (e: React.MouseEvent) => {
    e.stopPropagation(); // Prevent event click from triggering
    onEditClick?.(e);
  };

  return (
    <div
      className={` ${styles.customEvent} ${isCurrentUser ? styles.currentUserEvent :
""}` >
      <div className={styles.eventTime}>{event.timeText}</div>
      {titleParts.length > 0 && (
        <div className={styles.eventCreator}>{creatorName}</div>
      )}
      <div className={styles.eventTitle}>{eventTitle}</div>
      {isCurrentUser && (
        <button className={styles.editButton} onClick={handleEditClick}>
          <span>✎</span>
          <span>Edit</span>
        </button>
      )}
    </div>
  );
};

```

src\components\calendar\EventModaltsx

[to top](#)

```

import React, { useEffect, useState } from "react";
import styles from "../EventModal.module.css";
import { useQuery } from "convex/react";
import { api } from "../../convex";
import { showToast } from "../../utils/toast";

// Helper function to format date for datetime-local input
const formatDateForInput = (date: Date): string => {
  const year = date.getFullYear();
  const month = String(date.getMonth() + 1).padStart(2, "0");
  const day = String(date.getDate()).padStart(2, "0");
  const hours = String(date.getHours()).padStart(2, "0");
  const minutes = String(date.getMinutes()).padStart(2, "0");
  return `${year}-${month}-${day}T${hours}:${minutes}`;
};

// Helper function to get initial date value
const getInitialDate = (dateStr: string, addHour = false): string => {
  try {
    // Handle both ISO string and FullCalendar's date string formats
    // FullCalendar format example: "2024-03-15T09:00:00"
    const date = new Date(dateStr);

    if (isNaN(date.getTime())) {
      // If parsing fails, try parsing as a FullCalendar date string
      const [datePart, timePart] = dateStr.split("T");
      if (datePart && timePart) {
        const [year, month, day] = datePart.split("-").map(Number);
        const [hours, minutes] = timePart.split(":").map(Number);
        const newDate = new Date(year, month - 1, day, hours, minutes);
        if (!isNaN(newDate.getTime())) {
          if (addHour) {
            newDate.setHours(newDate.getHours() + 1);
          }
          return formatDateForInput(newDate);
        }
      }
    }

    // If all parsing fails, use current time
    const now = new Date();
    if (addHour) {
      now.setHours(now.getHours() + 1);
    }
    return formatDateForInput(now);
  }

  if (addHour) {
    date.setHours(date.getHours() + 1);
  }
  return formatDateForInput(date);
} catch {
  // Fallback to current time if any error occurs

```



```

    const now = new Date();
    if (addHour) {
        now.setHours(now.getHours() + 1);
    }
    return formatDateForInput(now);
}
};

interface EventModalProps {
    isOpen: boolean;
    onClose: () => void;
    onSubmit: (eventData: { title: string; start: string; end: string }) =>
    Promise<void>;
    onDelete?: (eventId: any) => Promise<void>;
    dateStr: string;
    event: {
        id?: any; // Convex ID type
        title: string;
        start: string;
        end?: string;
        creatorName?: string;
    } | null;
    userId: string; // Add userId to props for checking user's events
}

export const EventModal: React.FC<EventModalProps> = ({
    isOpen,
    onClose,
    onSubmit,
    onDelete,
    dateStr,
    event,
    userId,
}) => {
    const [title, setTitle] = React.useState("");
    const [startDate, setStartDate] = React.useState(() =>
        getInitialDate(dateStr)
    );
    const [endDate, setEndDate] = React.useState(() =>
        getInitialDate(dateStr, true)
    );
    const [error, setError] = React.useState<string | null>(null);
    const [isDeleting, setIsDeleting] = React.useState(false);
    const [overlappingSession, setOverlappingSession] = useState<any | null>(null);

    // Get user's existing events to check for overlaps
    const userEvents = useQuery(api.events.getByUserId, { userId }) || [];

    // Check for overlapping sessions
    useEffect(() => {
        if (!isOpen || !userId || !startDate || !endDate) return;

        try {

```

```

const newStart = new Date(startDate).getTime();
const newEnd = new Date(endDate).getTime();

// Reset overlap state
setOverlappingSession(null);

// Check each existing event for overlap
for (const existingEvent of userEvents) {
  // Skip the current event being edited
  if (event && existingEvent._id === event.id) continue;

  const existingStart = new Date(existingEvent.start).getTime();
  const existingEnd = existingEvent.end
    ? new Date(existingEvent.end).getTime()
    : existingStart + 60 * 60 * 1000; // Default to 1 hour if no end time

  // Check for overlap
  const overlap = (newStart >= existingStart && newStart < existingEnd) ||
    (newEnd > existingStart && newEnd <= existingEnd) ||
    (newStart <= existingStart && newEnd >= existingEnd);

  if (overlap) {
    setOverlappingSession(existingEvent);
    break;
  }
}
} catch (error) {
  console.error("Error checking for overlapping sessions:", error);
}
}, [isOpen, userEvents, startDate, endDate, userId, event]);

// Reset form when modal opens
React.useEffect(() => {
  if (isOpen) {
    setError(null);
    setOverlappingSession(null);
    if (event) {
      setTitle(event.title);
      setStartDate(getInitialDate(event.start));
      setEndDate(getInitialDate(event.end || event.start));
    } else {
      setTitle("");
      setStartDate(getInitialDate(dateStr));
      setEndDate(getInitialDate(dateStr, true));
    }
  }
}, [dateStr, isOpen, event]);

if (!isOpen) return null;

const handleSubmit = (e: React.FormEvent) => {
  e.preventDefault();
  try {

```

```

setError(null);
const start = new Date(startDate);
const end = new Date(endDate);
const now = new Date();

console.log("EventModal - Form Submission:", {
  title,
  start,
  end,
  now,
  isPastDate: start < now
});

if (isNaN(start.getTime()) || isNaN(end.getTime())) {
  console.error("EventModal - Invalid date format detected");
  const errorMsg = "Invalid date format. Please check your input.";
  setError(errorMsg);
  showToast.error(errorMsg);
  throw new Error(errorMsg);
}

// Check if start date is in the past
if (start < now) {
  console.log("EventModal - Rejected: Past date detected");
  setError("Cannot create session with a past date. Please select a future
date.");
  showToast.session.pastDateError();
  return;
}

// Check for overlapping sessions
if (overlappingSession) {
  console.log("EventModal - Rejected: Overlapping session detected",
overlappingSession);
  const errorMsg = `This session overlaps with your existing session
"${overlappingSession.title}"`;
  setError(errorMsg);
  showToast.session.overlapError();
  return;
}

console.log("EventModal - Submitting form:", {
  title,
  start: start.toISOString(),
  end: end.toISOString(),
  isEdit: !!event
});

const eventData = {
  title,
  start: start.toISOString(),
  end: end.toISOString(),
};

```

```

// We need to use a Promise for async handling
onSubmit(eventData)
  .then(() => {
    setTitle("");
    onClose();
    console.log("EventModal - Form submitted successfully");
    // Toast notifications are handled in the Calendar component
  })
  .catch((err: any) => {
    console.error("EventModal - Error from backend:", err);
    setError(err.message || "Failed to create/update session. Please try
again.");
    // Toast notifications for backend errors are handled in the Calendar
component
  });
} catch (error: any) {
  console.error("EventModal - Error processing dates:", error);
  setError(error.message || "Invalid date format. Please check your input.");
}
};

const handleDelete = async () => {
  if (!event?.id || !onDelete) return;

  try {
    setIsDeleting(true);
    setError(null);
    console.log("EventModal - Deleting session:", event.id);

    await onDelete(event.id);
    console.log("EventModal - Session deleted successfully");

    onClose();
    // Toast notification handled in Calendar component
  } catch (error: any) {
    console.error("EventModal - Error deleting session:", error);
    const errorMsg = error.message || "Failed to delete session. Please try again.";
    setError(errorMsg);
    showToast.error(errorMsg);
    setIsDeleting(false);
  }
};

const handleDeleteConfirm = () => {
  if (window.confirm("Are you sure you want to delete this session? This action
cannot be undone.)) {
    handleDelete();
  }
};

const isValidDateRange = new Date(startDate) < new Date(endDate);
const now = new Date();

```

```

const minDateTimeValue = formatDateForInput(now);

return (
  <div className={styles.modalOverlay}>
    <div className={styles.modal}>
      <h2>{event ? "Edit Session" : "Create New Session"}</h2>
      {error && <div className={styles.errorMessage}>{error}</div>}
      {overlappingSession && (
        <div className={styles.warningMessage}>
          Warning: This session overlaps with your existing session "
{overlappingSession.title}"
        </div>
      )}
    </div>
    <form onSubmit={handleSubmit}>
      <div className={styles.formGroup}>
        <label htmlFor="title">Session Title</label>
        <input
          id="title"
          type="text"
          value={title}
          onChange={(e) => setTitle(e.target.value)}
          placeholder="Enter session title"
          autoFocus
          required
        />
      </div>
      <div className={styles.formGroup}>
        <label htmlFor="startDate">Start Date & Time</label>
        <input
          id="startDate"
          type="datetime-local"
          value={startDate}
          onChange={(e) => setStartDate(e.target.value)}
          min={!event ? minDateTimeValue : undefined}
          required
        />
        {new Date(startDate) < now && !event && (
          <div className={styles.errorMessage}>
            Start time cannot be in the past
          </div>
        )}
      </div>
      <div className={styles.formGroup}>
        <label htmlFor="endDate">End Date & Time</label>
        <input
          id="endDate"
          type="datetime-local"
          value={endDate}
          onChange={(e) => setEndDate(e.target.value)}
          min={startDate}
          required
        />
        {!isValidDateRange && (

```

```

        <div className={styles.errorMessage}>
            End time must be after start time
        </div>
    )}
</div>
<div className={styles.buttonGroup}>
    {event && onDelete && (
        <button
            type="button"
            onClick={handleDeleteConfirm}
            className={styles.deleteButton}
            disabled={isDeleting}
        >
            {isDeleting ? "Deleting..." : "Delete Session"}
        </button>
    )}
    <div className={styles.rightButtons}>
        <button
            type="button"
            onClick={onClose}
            className={styles.cancelButton}
        >
            Cancel
        </button>
        <button
            type="submit"
            className={styles.submitButton}
            disabled={!title.trim() || !isValidDateRange || overlappingSession !==
null}
        >
            {event ? "Update Session" : "Create Session"}
        </button>
    </div>
</div>
</form>
</div>
</div>
    );
};

```

src\components\calendar\index.ts

[to top](#)

```
export { Calendar } from "../Calendar";
```

src\components\calendar\SessionDetailsModal.tsx

[to top](#)

```

import React, { useState, useEffect } from "react";
import { useNavigate } from "react-router-dom";
import styles from "../SessionDetailsModal.module.css";
import { showToast } from "../../utils/toast";

// Constants
const EARLY_JOIN_MINUTES = 10;

interface SessionDetailsModalProps {
  isOpen: boolean;
  onClose: () => void;
  event: {
    title: string;
    start: string;
    end: string;
    creatorName: string;
    id: string;
  } | null;
  isOwnEvent?: boolean;
  onEdit?: () => void;
}

export const SessionDetailsModal: React.FC<SessionDetailsModalProps> = ({
  isOpen,
  onClose,
  event,
  isOwnEvent = false,
  onEdit,
}) => {
  const navigate = useNavigate();
  const [currentTime, setCurrentTime] = useState(new Date());
  const [canEnterSession, setCanEnterSession] = useState(false);
  const [sessionStatus, setSessionStatus] = useState<'upcoming' | 'active' | 'ended' | 'early'>('upcoming');
  const [timeRemaining, setTimeRemaining] = useState('');

  // Update current time every second
  useEffect(() => {
    if (!isOpen) return;

    const timer = setInterval(() => {
      setCurrentTime(new Date());
    }, 1000);

    return () => clearInterval(timer);
  }, [isOpen]);

  // Check if user can enter the session
  useEffect(() => {
    if (!event) return;

    const startTime = new Date(event.start);

```

```

const endTime = new Date(event.end);
const now = currentTime;

// Calculate time until session starts
const timeUntilStart = startTime.getTime() - now.getTime();
const earlyJoinThreshold = EARLY_JOIN_MINUTES * 60 * 1000; // 10 minutes in
milliseconds

if (timeUntilStart > earlyJoinThreshold) {
  // Session starts in more than 10 minutes - cannot join yet
  setSessionStatus('upcoming');
  setCanEnterSession(false);

  // Format time remaining
  const hours = Math.floor(timeUntilStart / (1000 * 60 * 60));
  const minutes = Math.floor((timeUntilStart % (1000 * 60 * 60)) / (1000 * 60));
  const seconds = Math.floor((timeUntilStart % (1000 * 60)) / 1000);

  if (hours > 0) {
    setTimeRemaining(`${hours}h ${minutes}m ${seconds}s`);
  } else if (minutes > 0) {
    setTimeRemaining(`${minutes}m ${seconds}s`);
  } else {
    setTimeRemaining(`${seconds}s`);
  }
} else if (timeUntilStart > 0 && timeUntilStart <= earlyJoinThreshold) {
  // Within early join window (10 minutes before start)
  setSessionStatus('early');
  setCanEnterSession(true);

  const minutesUntilStart = Math.ceil(timeUntilStart / (1000 * 60));
  setTimeRemaining(`${minutesUntilStart}m`);
} else if (now > endTime) {
  // Session has ended
  setSessionStatus('ended');
  setCanEnterSession(false);
  setTimeRemaining('');
} else {
  // Session is active
  setSessionStatus('active');
  setCanEnterSession(true);
  setTimeRemaining('');
}
}, [event, currentTime]);

if (!isOpen || !event) return null;

// Format dates for display
const startDate = new Date(event.start).toLocaleString();
const endDate = new Date(event.end).toLocaleString();

const handleEnterSession = () => {
  if (!canEnterSession) {

```



```

        console.log("Cannot enter session yet - not time");
        showToast.error("Cannot enter session yet. Please wait until the scheduled
time.");
        return;
    }

    // Show appropriate toast based on session status
    if (sessionStatus === 'early') {
        const minutesEarly = Math.ceil(Number(timeRemaining.replace('m', '')));
        showToast.session.earlyJoin(minutesEarly);
    } else if (sessionStatus === 'active') {
        showToast.session.joined();
    }

    navigate(`/session/${event.id}`);
    onClose();
};

const getButtonLabel = () => {
    if (sessionStatus === 'early') {
        return `Enter Session Early (Starts in ${timeRemaining})`;
    } else if (sessionStatus === 'active') {
        return "Enter Session";
    } else if (sessionStatus === 'ended') {
        return "Session Has Ended";
    } else {
        return "Session Not Started Yet";
    }
};

return (
    <div className={styles.modalOverlay} onClick={onClose}>
        <div className={styles.modal} onClick={(e) => e.stopPropagation()}>
            <div className={styles.modalHeader}>
                <h2>Session Details</h2>
                <button onClick={onClose} className={styles.closeButton}>
                    &times;
                </button>
            </div>
            <div className={styles.modalContent}>
                <div className={styles.detailRow}>
                    <strong>Title:</strong>
                    <span>{event.title}</span>
                </div>
                <div className={styles.detailRow}>
                    <strong>Created by:</strong>
                    <span>{event.creatorName}</span>
                </div>
                <div className={styles.detailRow}>
                    <strong>Start Time:</strong>
                    <span>{startDate}</span>
                </div>
                <div className={styles.detailRow}>

```

```

        <strong>End Time:</strong>
        <span>{endDate}</span>
    </div>
    <div className={styles.detailRow}>
        <strong>Status:</strong>
        <span className={styles[sessionStatus]}>
            {sessionStatus === 'upcoming' && (
                <>
                    Upcoming (Starts in {timeRemaining})
                </>
            )}
            {sessionStatus === 'early' && (
                <>
                    Starting Soon (Join {timeRemaining} early)
                </>
            )}
            {sessionStatus === 'active' && 'Active Now'}
            {sessionStatus === 'ended' && 'Session Ended'}
        </span>
    </div>
</div>
<div className={styles.modalFooter}>
    {sessionStatus === 'upcoming' ? (
        <button disabled className={styles.disabledButton}>
            Session not started yet
        </button>
    ) : sessionStatus === 'ended' ? (
        <button disabled className={styles.disabledButton}>
            Session has ended
        </button>
    ) : (
        <button
            onClick={handleEnterSession}
            className={sessionStatus === 'early' ? styles.earlyButton :
styles.enterButton}
        >
            {getButtonLabel()}
        </button>
    )}
    {isOwnEvent && onEdit && (
        <button onClick={onEdit} className={styles.editButton}>
            Edit Session
        </button>
    )}
</div>
</div>
);
};

```



```

import { useState, useRef, useEffect } from "react";
import { useUser } from "@clerk/clerk-react";
import { SignOutButton } from "@clerk/clerk-react";
import styles from "./Header.module.css";

interface HeaderProps {}

export const Header = ({}: HeaderProps) => {
  const { isLoading, isSignedIn, user } = useUser();
  const [isDropdownOpen, setIsDropdownOpen] = useState(false);
  const dropdownRef = useRef<HTMLDivElement>(null);

  const toggleDropdown = () => {
    setIsDropdownOpen(!isDropdownOpen);
  };

  useEffect(() => {
    const handleClickOutside = (event: MouseEvent) => {
      if (
        dropdownRef.current &&
        !dropdownRef.current.contains(event.target as Node)
      ) {
        setIsDropdownOpen(false);
      }
    };

    document.addEventListener("mousedown", handleClickOutside);
    return () => {
      document.removeEventListener("mousedown", handleClickOutside);
    };
  }, []);

  let displayUsername: string | null = null;
  if (isLoading && isSignedIn && user) {
    displayUsername = user.username ||
      (user.firstName && user.lastName ?
        `${user.firstName} ${user.lastName}` :
        user.emailAddresses?.[0]?.emailAddress || 'user');
  } else if (isLoading && !isSignedIn) {
    displayUsername = "Guest";
  }

  return (
    <header className={styles.header}>
      <h1>Workoutmate</h1>
      <div className={styles.userProfile}>
        <div className={styles.profileContainer} onClick={toggleDropdown}>
          <div className={styles.profileIcon}>
            {displayUsername ? displayUsername.charAt(0).toUpperCase() : "U"}
          </div>
        </div>
        {isDropdownOpen && (

```

```

        <div ref={dropdownRef} className={styles.profileDropdown}>
          <div className={styles.dropdownUserInfo}>{displayUsername}</div>
          {isSignedIn && (
            <SignOutButton>
              <button className={styles.dropdownItem}>
                Sign Out
              </button>
            </SignOutButton>
          )}
        </div>
      )}
    </div>
  </header>
);
};

```

src\components\layout\index.ts

[to top](#)

```

export { Header } from "../Header";

```

src\components\messaging\Chattsx

[to top](#)

```

import { useEffect } from "react";
import { useQuery } from "convex/react";
import { api } from "../../convex/_generated/api";
import { useAuth } from "../../contexts/AuthContext";
import { Header } from "../layout";
import { MessageInput } from "../MessageInput";
import styles from "../Chat.module.css";

interface ChatProps {
  userId: string | null;
  username: string | null;
}

export const Chat = ({ userId, username }: ChatProps) => {
  const messages = useQuery(api.chat.getMessages);

  useEffect(() => {
    // Make sure scrollTo works on button click in Chrome
    if (messages) {
      setTimeout(() => {
        window.scrollTo({ top: document.body.scrollHeight, behavior: "smooth" });
      }, 0);
    }
  }, [messages]);

  // Since ProtectedRoute ensures we always have a username, we can safely assert it's
  // non-null
  const userDisplayName = username as string;

  return (
    <div className={styles.chatContainer}>
      <Header username={userDisplayName} />
      <main className={styles.chat}>
        {messages?.map((message) => (
          <article
            key={message._id}
            className={` ${styles.messageArticle} ${message.user === userDisplayName ?
styles.messageMine : ""}`}
          >
            <div>{message.user}</div>

            <p>{message.body}</p>
          </article>
        ))}
      </main>
      <MessageInput username={userDisplayName} />
    </div>
  );
};

```

[to top](#)

```
export { Chat } from "./Chat";  
export { MessageInput } from "./MessageInput";  
export { SessionChat } from "./SessionChat";
```

src\components\messaging\MessageInputtsx

[to top](#)

```

import { useState } from "react";
import { useMutation } from "convex/react";
import { api } from "../../convex/_generated/api";
import styles from "../MessageInput.module.css";

interface MessageInputProps {
  username: string;
  sessionId?: string; // Make sessionId optional for backward compatibility
  disabled?: boolean; // Add optional disabled prop
}

export const MessageInput = ({ username, sessionId, disabled = false }:
MessageInputProps) => {
  const sendMessage = useMutation(api.chat.sendMessage);
  const [newMessageText, setNewMessageText] = useState("");

  const handleSubmit = async (e: React.FormEvent) => {
    e.preventDefault();
    if (disabled) return;
    await sendMessage({
      user: username,
      body: newMessageText,
      sessionId,
    });
    setNewMessageText("");
  };

  return (
    <form onSubmit={handleSubmit} className={styles.form}>
      <input
        value={newMessageText}
        onChange={(e) => setNewMessageText(e.target.value)}
        placeholder={disabled ? "Log in to chat" : "Write a message..."}
        autoFocus
        className={styles.input}
        disabled={disabled} // Apply disabled prop
      />
      <button
        type="submit"
        disabled={!newMessageText || disabled} // Apply disabled prop here too
        className={styles.button}
      >
        Send
      </button>
    </form>
  );
};

```



```

import { useState, useEffect, useRef } from "react";
import { useParams } from "react-router-dom";
import { useQuery, useMutation } from "convex/react";
import { useUser } from "@clerk/clerk-react";
import { api } from "../../convex";
import { MessageInput } from "../MessageInput";
import styles from "../Chat.module.css";

interface SessionChatProps {}

export function SessionChat({}: SessionChatProps) {
  const { sessionId } = useParams<{ sessionId: string }>();
  const chatContainerRef = useRef<HTMLDivElement>(null);

  // Get auth state and user details from Clerk
  const { isLoading, isSignedIn, user } = useUser();

  // Derive userId and username
  const currentUsername = isLoading && isSignedIn && user ?
    user.username ||
    (user.firstName && user.lastName ?
      `${user.firstName} ${user.lastName}` :
      user.emailAddresses?.[0]?.emailAddress || 'user')
    : "";

  // Get session details
  const session = useQuery(api.events.getEventById, { id: sessionId || "" });
  // Use session-specific messages instead of all messages
  const messages = useQuery(api.chat.getSessionMessages, {
    sessionId: sessionId || "",
  });

  useEffect(() => {
    // Scroll to bottom when messages change
    if (messages && chatContainerRef.current) {
      chatContainerRef.current.scrollTop =
        chatContainerRef.current.scrollHeight;
    }
  }, [messages]);

  if (!session) {
    return <div>Loading session information...</div>;
  }

  return (
    <div className={styles.chatContainer}>
      <main className={styles.chat} ref={chatContainerRef}>
        {messages?.length === 0 ? (
          <div className={styles.emptyState}>
            No messages yet. Start the conversation!
          </div>
        ) : (

```

```

        messages?.map((message) => (
            <article
                key={message._id}
                className={` ${styles.messageArticle} ${message.user === currentUsername ?
styles.messageMine : ""}`}
            >
                <div>{message.user}</div>
                <p>{message.body}</p>
            </article>
        ))
    )}
</main>
<div className={styles.inputContainer}>
    <MessageInput
        username={currentUsername}
        sessionId={sessionId}
        disabled={!isLoading || !isSignedIn}
    />
</div>
</div>
);
};

```

src\components\session\index.ts

[to top](#)

```

export { Session } from './Session';
export { VideoCall } from './VideoCall';

```

src\components\session\Session.tsx

[to top](#)

```

import { useParams, useNavigate } from "react-router-dom";
import { useQuery, useMutation } from "convex/react";
import { useUser } from "@clerk/clerk-react";
import { api } from "../../convex";
import { Header } from "../../layout";
import { SessionChat } from "../../messaging";
import { VideoCall } from "../../VideoCall";
import styles from "../../Session.module.css";
import { useEffect, useState } from "react";
import { showToast } from "../../utils/toast";

interface SessionProps {}

interface SessionStatus {
  canJoin: boolean;
  message: string;
  status: 'upcoming' | 'active' | 'ended' | 'loading' | 'early';
  timeRemaining?: string;
}

export function Session({}: SessionProps) {
  const { sessionId } = useParams<{ sessionId: string }>();
  const navigate = useNavigate();
  const [sessionStatus, setSessionStatus] = useState<SessionStatus>({
    canJoin: false,
    message: "Loading session information...",
    status: 'loading'
  });

  const [currentTime, setCurrentTime] = useState(new Date());
  const [hasShownJoinToast, setHasShownJoinToast] = useState(false);

  // Use Clerk's useUser hook directly for auth state and user details
  const { isLoading: clerkLoaded, isSignedIn, user } = useUser();

  // Mutation for joining the session
  const joinSessionMutation = useMutation(api.events.joinSession);

  // Get session details
  const session = useQuery(api.events.getEventById, { id: sessionId || "" });

  // Update time every second
  useEffect(() => {
    const timer = setInterval(() => {
      setCurrentTime(new Date());
    }, 1000);

    return () => clearInterval(timer);
  }, []);

  // Validate session timing
  useEffect(() => {
    // Ensure session is loaded and is actually an event document

```

```

if (!session || !("title" in session) || !("start" in session)) return;

const startTime = new Date(session.start);
const endTime = new Date(session.end || session.start); // Use start if end is
missing
const now = currentTime;

// Calculate time values
const timeUntilStart = startTime.getTime() - now.getTime();
const earlyJoinThreshold = 10 * 60 * 1000; // 10 minutes in milliseconds

if (timeUntilStart > earlyJoinThreshold) {
  // Session starts in more than 10 minutes - cannot join yet
  // Format time remaining
  const hours = Math.floor(timeUntilStart / (1000 * 60 * 60));
  const minutes = Math.floor((timeUntilStart % (1000 * 60 * 60)) / (1000 * 60));
  const seconds = Math.floor((timeUntilStart % (1000 * 60)) / 1000);

  let timeRemainingStr;
  if (hours > 0) {
    timeRemainingStr = `${hours}h ${minutes}m ${seconds}s`;
  } else if (minutes > 0) {
    timeRemainingStr = `${minutes}m ${seconds}s`;
  } else {
    timeRemainingStr = `${seconds}s`;
  }

  setSessionStatus({
    canJoin: false,
    message: `This session hasn't started yet. Please come back at the scheduled
time.`,
    status: 'upcoming',
    timeRemaining: timeRemainingStr
  });
} else if (timeUntilStart > 0 && timeUntilStart <= earlyJoinThreshold) {
  // Within 10 minutes of start time - early join is allowed
  const minutesUntilStart = Math.ceil(timeUntilStart / (1000 * 60));

  setSessionStatus({
    canJoin: true,
    message: `Session starts in ${minutesUntilStart} minute${minutesUntilStart !==
1 ? 's' : ''}. You can join early to prepare.`,
    status: 'early',
    timeRemaining: `${minutesUntilStart}m`
  });

  // Show early join toast once
  if (!hasShownJoinToast) {
    showToast.session.earlyJoin(minutesUntilStart);
    setHasShownJoinToast(true);
  }
} else if (now > endTime) {
  // Session has ended

```

```

    setSessionStatus({
      canJoin: false,
      message: "This session has ended.",
      status: 'ended'
    });
  } else {
    // Session is active
    setSessionStatus({
      canJoin: true,
      message: "Session is active",
      status: 'active'
    });

    // Show joined toast once
    if (!hasShownJoinToast) {
      showToast.session.joined();
      setHasShownJoinToast(true);
    }
  }
}, [session, currentTime, hasShownJoinToast]);

// Effect to join session when conditions are met
useEffect(() => {
  // Use clerkLoaded and isSignedIn for the check
  if (sessionStatus.canJoin && sessionId && clerkLoaded && isSignedIn) {
    console.log("Attempting to join session (Clerk auth confirmed):", sessionId);
    joinSessionMutation({ eventId: sessionId as any }) // Cast to any if Id type
    causes issues
      .then(() => {
        console.log("Successfully called joinSession mutation");
      })
      .catch((error) => {
        console.error("Error calling joinSession mutation:", error);
        showToast.error("Failed to mark you as joined in the session.");
      });
  }
}, [sessionStatus.canJoin, sessionId, joinSessionMutation, clerkLoaded, isSignedIn]);

// Derive userId and username only if signed in and loaded
const currentUserId = clerkLoaded && isSignedIn && user ? user.id : "";
const currentUsername = clerkLoaded && isSignedIn && user ?
  user.username ||
  (user.firstName && user.lastName ?
    `${user.firstName} ${user.lastName}` :
    user.emailAddresses?.[0]?.emailAddress || 'user')
  : "";

// If session data is still loading
if (!session) {
  return (
    <div className={styles.sessionContainer}>
      <Header />
      <div className={styles.errorContainer}>

```

```

        <div className={styles.errorMessage}>
            Loading session information...
        </div>
    </div>
</div>
);
}

// If session timing is invalid, show error
if (!sessionStatus.canJoin) {
    return (
        <div className={styles.sessionContainer}>
            <Header />
            <div className={styles.sessionHeader}>
                <button className={styles.backButton} onClick={() => navigate(-1)}>
                    ← Back
                </button>
                <h1>{session && 'title' in session ? session.title : 'Session'}</h1>
            </div>
            <div className={styles.errorContainer}>
                <div className={styles.errorMessage}>
                    {sessionStatus.message}
                    {sessionStatus.status === 'upcoming' && (
                        <div className={styles.timeRemaining}>
                            Starting in: <span className={styles.countdown}>
{sessionStatus.timeRemaining}</span>
                        </div>
                    )}
                </div>
                <button
                    className={styles.actionButton}
                    onClick={() => navigate('/calendar')}
                >
                    Return to Calendar
                </button>
            </div>
        </div>
    );
}

// Early join or active session - show a banner for early join
const isEarlyJoin = sessionStatus.status === 'early';

// Extract participant IDs, default to empty array
// Ensure session is an event before accessing participantIds
const participantIds = (session && "participantIds" in session &&
session.participantIds) ? session.participantIds : [];

// Log the participantIds being passed down on each render
console.log(`[Session.tsx] Rendering. Passing participantIds to VideoCall:`,
participantIds);

return (

```

```

<div className={styles.sessionContainer}>
  <Header />
  <div className={styles.sessionHeader}>
    <button className={styles.backButton} onClick={() => navigate(-1)}>
      ← Back
    </button>
    <h1>{session && 'title' in session ? session.title : 'Loading...'}</h1>
  </div>

  {isEarlyJoin && (
    <div className={styles.earlyJoinBanner}>
      <div className={styles.earlyJoinMessage}>
        {sessionStatus.message}
      </div>
    </div>
  )}

  <div className={styles.sessionContent}>
    <div className={styles.mainArea}>
      /* Use clerkLoaded and isSignedIn for the render condition */
      {clerkLoaded && isSignedIn ? (
        <VideoCall
          sessionId={sessionId || ""}
          userId={currentUserId}      // Pass derived userId
          username={currentUsername}  // Pass derived username
          participantIds={participantIds}
        />
      ) : (
        // Optionally show a loading state while auth checks
        <div>Authenticating video call...</div>
      )}
    </div>
    <div className={styles.sidebarArea}>
      /* Remove userId and username props from SessionChat */
      <SessionChat />
    </div>
  </div>
</div>
);
};

```

src\components\session\VideoCalltsx

[to top](#)

```

import { useEffect, useRef, useState, useCallback } from "react";
import { useQuery, useMutation, useConvexAuth } from "convex/react";
import { api } from "../../convex/_generated/api";
import { Id, Doc } from "../../convex/_generated/dataModel";
import styles from "./VideoCall.module.css";

// Define the structure for peer connections
interface PeerConnection {
  connection: RTCPeerConnection;
  remoteStream?: MediaStream;
}

interface VideoCallProps {
  sessionId: string;
  userId: string; // Current user's ID
  username: string;
  participantIds: string[]; // IDs of ALL participants including self
}

// Basic STUN server configuration (Google's public servers)
const stunServers = {
  iceServers: [
    { urls: "stun:stun.l.google.com:19302" },
    { urls: "stun:stun1.l.google.com:19302" },
  ],
};

export const VideoCall = ({ sessionId, userId, username, participantIds }:
VideoCallProps) => {
  const [localStream, setLocalStream] = useState<MediaStream | null>(null);
  const [remoteStreams, setRemoteStreams] = useState<Record<string, MediaStream>>({});
  // State to hold remote streams
  const localVideoRef = useRef<HTMLVideoElement>(null);
  const remoteVideoRefs = useRef<Record<string, HTMLVideoElement | null>>({}); // Refs
  for remote video elements
  const pendingCandidatesRef = useRef<Record<string, RTCIceCandidateInit[]>>({}); //
  Ref to store pending candidates
  const peerConnections = useRef<Record<string, RTCPeerConnection>>({}); // Use useRef
  for peer connections
  const makingOffer = useRef<Record<string, boolean>>({}); // Track makingOffer state
  const ignoreOffer = useRef<Record<string, boolean>>({}); // Track ignoreOffer state
  const queuedIceCandidatesRef = useRef<Record<string, RTCIceCandidate[]>>({}); //
  Queue for early candidates

  // Convex Auth state
  const { isLoading: isAuthLoading, isAuthenticated } = useConvexAuth();

  // Convex mutations and queries
  const sendSignal = useMutation(api.video.sendSignal);
  const deleteSignal = useMutation(api.video.deleteSignal);
  const signals = useQuery(api.video.getSignals, isAuthenticated ? { sessionId } :
"skip");

```



```

const otherParticipantIds = participantIds.filter(id => id !== userId);

// --- Initialize Local Media ---
useEffect(() => {
  const startMedia = async () => {
    try {
      const stream = await navigator.mediaDevices.getUserMedia({ video: true, audio:
true });
      setLocalStream(stream);
      if (localVideoRef.current) {
        localVideoRef.current.srcObject = stream;
      }
    } catch (error) {
      console.error("Error accessing media devices:", error);
      // Handle error appropriately (e.g., show message to user)
    }
  };
  startMedia();

  // Cleanup
  return () => {
    localStream?.getTracks().forEach((track) => track.stop());
    Object.values(peerConnections.current).forEach(pc => pc.close());
  };
  // eslint-disable-next-line react-hooks/exhaustive-deps
}, []); // Run only once on mount

// --- Create Peer Connection Function ---
const createPeerConnection = useCallback((targetUserId: string): RTCPeerConnection |
null => {
  if (!localStream) {
    console.error("Local stream not available to create peer connection.");
    return null;
  }
  console.log(`Creating peer connection to ${targetUserId}`);
  const pc = new RTCPeerConnection(stunServers);

  // Add local tracks
  localStream.getTracks().forEach((track) => {
    pc.addTrack(track, localStream);
  });

  // Triggered when the connection needs to negotiate (e.g., adding tracks)
  pc.onnegotiationneeded = async () => {
    // Avoid negotiation for self or if connection is closing
    if (!userId || userId === targetUserId || pc.signalingState === 'closed') {
      console.log(`Skipping negotiation for ${targetUserId} (self, closed, or no
user)`);
      return;
    }

    console.log(`Negotiation needed with ${targetUserId}, state:

```

```

    ${pc.signalingState}`);

    // Perfect Negotiation: Check flags before creating offer
    if (makingOffer.current[targetUserId] || ignoreOffer.current[targetUserId]) {
        console.log(`Negotiation needed for ${targetUserId}, but making/ignore flag
set. Skipping offer creation.`);
        return;
    }

    // Only proceed if the state is stable (or potentially closed if we want to
restart)
    if (pc.signalingState !== 'stable') {
        console.log(`Negotiation needed for ${targetUserId}, but state is
${pc.signalingState}. Skipping offer creation.`);
        return;
    }

    try {
        // Set flag before starting async operation
        makingOffer.current[targetUserId] = true;
        console.log(`Setting local description (offer) for ${targetUserId} via
negotiationneeded.`);
        const offer = await pc.createOffer();

        // Double-check state *after* offer creation, *before* setting local desc
        if (pc.signalingState !== 'stable') {
            console.warn(`State changed during offer creation for ${targetUserId}
(now ${pc.signalingState}). Aborting offer.`);
            makingOffer.current[targetUserId] = false; // Reset flag
            return;
        }

        await pc.setLocalDescription(offer);
        console.log(`Local description (offer) set for ${targetUserId}`);

        console.log(`Sending offer to ${targetUserId} via negotiationneeded`);
        sendSignal({
            sessionId,
            targetUserId: targetUserId,
            type: "offer",
            signal: JSON.stringify({ type: pc.localDescription?.type, sdp:
pc.localDescription?.sdp }),
        });
    } catch (error) {
        console.error(`Error during negotiationneeded offer for ${targetUserId}:`,
error);
    } finally {
        // Reset flag after operation completes or fails
        // Check if an incoming offer caused us to ignore ours before resetting
        if (!ignoreOffer.current[targetUserId]) {
            makingOffer.current[targetUserId] = false;
        }
    }
}

```

```

};

// Handle incoming remote tracks
pc.ontrack = (event) => {
  console.log(`Track received from ${targetUserId}`, event.streams[0]);
  const remoteStream = event.streams[0];
  if (remoteStream) {
    setRemoteStreams(prev => {
      const existingStream = prev[targetUserId];
      if (existingStream) {
        console.log(`Updating existing stream for ${targetUserId}`);
        event.track.onended = () => {
          console.log(`Remote track ended for ${targetUserId}`);
          // Optionally handle track ending, e.g., remove stream or track
        };
        return { ...prev, [targetUserId]: remoteStream };
      } else {
        console.log(`Creating new stream entry for ${targetUserId}`);
        event.track.onended = () => {
          console.log(`Remote track ended for ${targetUserId}`);
          // Optionally handle track ending
        };
        return { ...prev, [targetUserId]: remoteStream };
      }
    });
  }
};

// Handle ICE candidates
pc.onicecandidate = (event) => {
  if (event.candidate) {
    console.log(`Sending ICE candidate to ${targetUserId}`);
    sendSignal({
      sessionId,
      targetUserId,
      type: "candidate",
      signal: JSON.stringify(event.candidate),
    });
  }
};

// Handle connection state changes (optional but useful)
pc.onconnectionstatechange = () => {
  console.log(`Connection state with ${targetUserId}: ${pc.connectionState}`);
  // Can update UI or handle disconnections here
  if (pc.connectionState === 'disconnected' || pc.connectionState === 'failed' ||
pc.connectionState === 'closed') {
    // Clean up connection for this peer
    delete peerConnections.current[targetUserId]; // Use delete instead of
assigning undefined
  }
};

```

```

peerConnections.current[targetUserId] = pc;

return pc;
}, [localStream, sendSignal, sessionId]); // Dependencies

// Helper function to process queued ICE candidates for a specific peer
const processQueuedIceCandidates = useCallback(async (peerId: string) => {
  const pc = peerConnections.current[peerId];
  const queue = queuedIceCandidatesRef.current[peerId];

  // Ensure PC exists, remote description is set, and there's a queue with candidates
  if (pc && pc.remoteDescription && queue && queue.length > 0) {
    console.log(`Processing ${queue.length} queued ICE candidates for ${peerId}.
Remote desc type: ${pc.remoteDescription.type}, Signaling state:
${pc.signalingState}`);
    while (queue.length > 0) {
      const candidate = queue.shift(); // Get the oldest candidate
      if (candidate) {
        try {
          await pc.addIceCandidate(candidate);
          console.log(`Successfully added queued ICE candidate for ${peerId}:`,
candidate.candidate.substring(0, 30) + "..."); // Log partial candidate
        } catch (error) {
          // Common error: Trying to add candidate before remote description is fully
stable or in wrong state.
          // We might retry or log, depending on the specific error.
          console.error(`Error adding queued ICE candidate for ${peerId} (State:
${pc.signalingState}):`, error);
          // Optional: Put candidate back in front of queue if it's a state issue? Be
careful with loops.
          // queue.unshift(candidate);
        }
      }
    }
  } else if (queue && queue.length > 0) {
    // Log why we are *not* processing if the queue isn't empty
    console.log(`Not processing queued ICE candidates for ${peerId}. Conditions not
met: PC exists=${!!pc}, Remote desc set=${!!pc?.remoteDescription}, Queue
size=${queue?.length}. State: ${pc?.signalingState}`);
  }
  // Clear the queue if it exists, regardless of processing, to prevent reprocessing
old candidates if conditions change later
  // delete queuedIceCandidatesRef.current[peerId]; // Reconsider if clearing is
always right here.
}, []); // No dependencies needed as it uses refs

// --- Initiate Connections to New Peers ---
useEffect(() => {
  if (!localStream || isAuthLoading || !isAuthenticated) {
    console.log("Skipping peer connection initiation: Local stream or auth not
ready.");
    return; // Don't proceed if local stream or auth isn't ready
  }
}

```

```

console.log("Checking for new peers to connect to...", otherParticipantIds);

otherParticipantIds.forEach(peerId => {
  // Check if a connection already exists or is being established
  if (!peerConnections.current[peerId]) {
    console.log(`Initiating connection to new peer: ${peerId}`);

    // 1. Create Peer Connection
    const pc = createPeerConnection(peerId);
    if (!pc) {
      console.error(`Failed to create peer connection for ${peerId}`);
      return; // Skip if creation failed
    }

    // Note: Offer creation is now primarily handled by onnegotiationneeded
    // triggered by adding tracks in createPeerConnection. We might not need
    // to explicitly create/send an offer here anymore, unless onnegotiationneeded
    // doesn't fire reliably in all browsers/scenarios.
    // Let's keep the explicit offer sending for now as a fallback, but guard it.
    if (pc.signalingState === 'stable' && !makingOffer.current[peerId] &&
!ignoreOffer.current[peerId]){
      console.log(`Attempting initial offer to new peer: ${peerId} (fallback)`);
      makingOffer.current[peerId] = true; // Set flag
      pc.createOffer()
        .then(offer => {
          // Check state again before setting local description
          if (pc.signalingState !== 'stable') {
            console.warn(`State changed before setting initial offer for
${peerId}. Aborting.`);
            throw new Error(`Signaling state not stable:
${pc.signalingState}`);
          }
          return pc.setLocalDescription(offer);
        })
        .then(() => {
          console.log(`Initial local description (offer) set for ${peerId}`);
          if (pc.localDescription) {
            sendSignal({
              sessionId,
              targetUserId: peerId,
              type: "offer",
              signal: JSON.stringify({ type: pc.localDescription.type, sdp:
pc.localDescription.sdp }),
            });
          }
        })
        .catch(error => {
          console.error(`Error creating/sending initial offer to ${peerId}:`,
error);

          // Clean up potentially inconsistent state
          delete peerConnections.current[peerId];
          pc.close();

```

```

    })
    .finally(() => {
      // Reset flag if not ignored
      if (!ignoreOffer.current[peerId]) {
        makingOffer.current[peerId] = false;
      }
    });
  } else {
    console.log(`Skipping initial offer for ${peerId}:
state=${pc.signalingState}, making=${makingOffer.current[peerId]},
ignore=${ignoreOffer.current[peerId]}`);
  }
  } else {
    console.log(`Connection status for existing peer ${peerId}:
${peerConnections.current[peerId]?.connectionState}`);
    // Optional: Add logic here to re-initiate if connection failed previously
  }
});

// Optional: Clean up connections for peers who left
// Get current peer IDs from state
const currentPeerIds = Object.keys(peerConnections.current);
currentPeerIds.forEach(peerId => {
  if (!otherParticipantIds.includes(peerId)) {
    console.log(`Cleaning up connection for left peer: ${peerId}`);
    peerConnections.current[peerId]?.close();
    delete peerConnections.current[peerId]; // Use delete instead of assigning
undefined
  }
});

// Dependencies: Run when participant list, local stream, or auth state changes
// Added peerConnections to re-evaluate state, but be cautious of loops
}, [otherParticipantIds, localStream, isAuthLoading, isAuthenticated,
createPeerConnection, sendSignal, sessionId]);

// --- Process Incoming Signals ---
useEffect(() => {
  if (isAuthLoading || !isAuthenticated || !signals) {
    return; // Don't process signals if auth is loading or not authenticated
  }

  console.log("Received signals:", signals);

  signals?.forEach(async (signal) => {
    // Correct destructuring: Use userId and alias it as senderId
    const { userId: senderId, type, signal: signalData } = signal;

    // Don't process signals from self
    if (senderId === userId) return; // Compare aliased senderId with component's
userId

    // Ensure peer connection exists

```

```

const peerData = peerConnections.current[senderId];
let pc: RTCPeerConnection | null = peerData ? peerData : null;

// If connection doesn't exist for an incoming signal (e.g., offer), create it.
if (!pc && type === 'offer') {
  console.log(`Signal received from new peer ${senderId}. Creating connection.`);
  const createdPc = createPeerConnection(senderId); // Returns RTCPeerConnection
  | null
  if (createdPc) { // Check ensures createdPc is not null here
    // Update state *inside* the check where createdPc is known to be non-null
    peerConnections.current[senderId] = createdPc; // Explicitly assign non-
null connection
    pc = createdPc; // Assign the non-null connection to the loop variable 'pc'
  } else {
    console.error(`Failed to create peer connection for signal from
${senderId}`);
    // pc remains null, subsequent check will handle this
  }
}

// Check if pc is still null after potential creation attempt
if (!pc) {
  console.warn(`No peer connection found for signal from ${senderId}. Signal
type: ${type}. Ignoring.`);
  return; // Exit if no valid connection
}

try {
  const parsedData = JSON.parse(signalData);

  switch (type) {
    case "offer":
      console.log(`Received offer from: ${senderId}`);
      const offerDescription = new RTCSessionDescription(parsedData);

      // Perfect negotiation: Check makingOffer/ignoreOffer flags and signaling
state
      const isMakingOffer = makingOffer.current[senderId];
      const polite = userId! > senderId; // Determine politeness based on user ID
comparison
      const ignore = ignoreOffer.current[senderId];

      console.log(`Offer from ${senderId}: polite=${polite},
makingOffer=${isMakingOffer}, ignoreOffer=${ignore}, state=${pc.signalingState}`);

      // Condition 1: If we are making an offer and we are the impolite peer,
ignore the incoming offer.
      if (isMakingOffer && !polite) {
        console.log(`Ignoring offer from ${senderId} (impolite peer, currently
making offer)`);
        return; // Let our offer proceed
      }

```

```

        // Condition 2: Set ignore flag if we receive an offer while not stable
        and we are the polite peer
        // This prevents us from processing our own offer if it was created
        concurrently
        ignoreOffer.current[senderId] = !polite && pc.signalingState !== 'stable';

        // Condition 3: Check signaling state before setting remote description
        if (pc.signalingState !== 'stable' && pc.signalingState !== 'have-local-
offer') {
            console.warn(`Received offer from ${senderId}, but signaling state is
${pc.signalingState}. Cannot process.`);
            // If state is have-remote-offer, it's likely a duplicate, can safely
            ignore.
            return;
        }

        // Handle offer collision (glare) based on politeness
        console.log(`[Glare] Handling offer collision. My ID: ${userId}, Sender ID:
${senderId}, Polite: ${polite}`);
        if (polite) {
            // Polite peer rollback: Set remote description, create answer.
            console.log(`[Glare] Polite peer yielding to offer from ${senderId}.
Setting remote, then answering.`);
            await pc.setRemoteDescription(new RTCSessionDescription({ type:
'offer', sdp: parsedData.sdp }));
            console.log(`Remote description (offer) set for ${senderId}`);

            console.log(`Creating answer for ${senderId}`);
            if (pc.signalingState === 'have-remote-offer') { // Check state before
creating answer
                const answer = await pc.createAnswer();
                console.log(`Setting local description (answer) for ${senderId}`);
                if (pc.signalingState === 'have-remote-offer') { // Final check
before setting local answer
                    await pc.setLocalDescription(answer);
                    ignoreOffer.current[senderId] = false; // Reset flag
                    console.log(`Sending answer to ${senderId}`);
                    sendSignal({
                        sessionId,
                        targetUserId: senderId,
                        type: "answer",
                        signal: JSON.stringify({ type: answer.type, sdp: answer.sdp
}),
                    });
                } else {
                    console.warn(`[Aborting Answer] State changed to
${pc.signalingState} just before setting local description (answer) for ${senderId}.`);
                    ignoreOffer.current[senderId] = false; // Reset flag
                }
            } else {
                console.warn(`Tried to create answer for ${senderId} but signaling
state is ${pc.signalingState} (expected have-remote-offer).`);
                ignoreOffer.current[senderId] = false; // Reset flag
            }
        }
    }
}

```



```

    }
  } else {
    // Impolite peer rollback: Ignore the incoming offer for now.
    console.log(`[Glare] Impolite peer received offer from ${senderId}
while in ${pc.signalingState}. Ignoring this offer and setting ignoreOffer flag.`);
    ignoreOffer.current[senderId] = true; // Set flag to ignore subsequent
offers until this negotiation resolves
    // Do NOT process this incoming offer (no setRemoteDescription, no
createAnswer)
    // Let the negotiation initiated by this impolite peer proceed.
  }
  break;

case "answer":
  console.log(`Received answer from: ${senderId}`);
  // Recreate RTCSessionDescription from parsed data
  const answerDescription = new RTCSessionDescription(parsedData);
  // Set answer only if we are expecting one
  if (pc.signalingState === 'have-local-offer') {
    await pc.setRemoteDescription(answerDescription);
    console.log(`Remote description (answer) set for ${senderId}`);

    // Process any queued candidates for this peer
    await processQueuedIceCandidates(senderId);
  } else {
    console.warn(`Received answer from ${senderId}, but signaling state is
${pc.signalingState}. Ignoring.`);
  }

  deleteSignal({ signalId: signal._id });
  break;

case "candidate":
  console.log(`Received ICE candidate from: ${senderId}`);
  // Recreate RTCIceCandidate from parsed data
  const iceCandidate = new RTCIceCandidate(parsedData);
  // Add candidate only if remote description is set
  if (pc.remoteDescription) {
    try {
      await pc.addIceCandidate(iceCandidate);
      console.log(`Added ICE candidate from ${senderId}`);
    } catch (addError) {
      console.error(`Error adding ICE candidate for ${senderId}:`, addError);
    }
  } else {
    console.warn(`Received ICE candidate from ${senderId}, but remote
description not set yet. Queueing.`);
    // Queue the candidate if remote description isn't set
    if (!queuedIceCandidatesRef.current[senderId]) {
      queuedIceCandidatesRef.current[senderId] = [];
    }
    queuedIceCandidatesRef.current[senderId].push(iceCandidate); // Store the
candidate directly

```

```

    }

    deleteSignal({ signalId: signal._id });
    break;

    default:
      console.warn(`Received unknown signal type: ${type}`);
  }
} catch (error) {
  console.error(`Error processing signal from ${senderId}:`, signal, error);
}
});

```

// Note: Consider adding logic to clear processed signals from the Convex query if they persist.

}, [signals, isAuthLoading, isAuthenticated, createPeerConnection, sendSignal, sessionId, userId, deleteSignal, processQueuedIceCandidates]); // Added `userId` dependency

// --- Assign Remote Streams to Video Elements ---

```

useEffect(() => {
  Object.keys(remoteStreams).forEach((peerId) => {
    const stream = remoteStreams[peerId];
    const videoElement = remoteVideoRefs.current[peerId];
    if (videoElement && stream && videoElement.srcObject !== stream) {
      console.log(`Assigning remote stream from ${peerId} to video element`);
      videoElement.srcObject = stream;
    } else if (videoElement && !stream) {
      console.log(`Clearing stream for ${peerId}`);
      videoElement.srcObject = null;
    }
  });
}, [remoteStreams]); // Re-run when remoteStreams changes

```

// --- Render Component ---

```

return (
  <div className={styles.videoCallContainer}>
    <h2>Video Call - {username} ({userId.substring(0, 4)})</h2>
    <div className={styles.videoArea}>
      {/* Local Video */}
      <div className={styles.videoWrapper}>
        <video
          ref={localVideoRef}
          className={styles.videoElement}
          autoPlay
          playsInline
          muted // Mute local video playback to avoid echo
        />
        <span>You ({username.substring(0,6)})</span>
      </div>
    </div>
  </div>
)

```

```

    { /* Remote Videos */ }
    { Object.keys(remoteStreams).map((peerId) => (
      // Only render if connection exists, even if stream not yet arrived
      peerConnections.current[peerId] && (
        <div key={peerId} className={styles.videoWrapper}>
          <video
            ref={(el) => (remoteVideoRefs.current[peerId] = el)} // Assign ref
            className={styles.videoElement}
            autoPlay
            playsInline
          />
          <span>Peer: {peerId.substring(0, 4)}
            {peerConnections.current[peerId].connectionState}</span>
          { /* TODO: Get actual username for peerId */ }
        </div>
      )
    )}
  </div>
  <div className={styles.controls}>
    { /* TODO: Add call controls (mute audio, disable video, hang up) */ }
    <button onClick={() => console.log("Peer Connections:",
      peerConnections.current)}>Log Peers</button>
    <button onClick={() => console.log("Signals:", signals)}>Log Signals</button>
  </div>
</div>
);
};

```

src\contexts\AuthContexttsx

[to top](#)

```

import { createContext, useContext, useState, useEffect, ReactNode } from "react";
import { useNavigate } from "react-router-dom";
import { useClerk, useUser } from "@clerk/clerk-react";

interface AuthContextType {
  userId: string | null;
  username: string | null;
  login: (userId: string, username: string) => void;
  logout: () => void;
  isAuthenticated: boolean;
}

// Create the context with a default value
const AuthContext = createContext<AuthContextType | undefined>(undefined);

// Provider component
export function AuthProvider({ children }: { children: ReactNode }) {
  const navigate = useNavigate();
  const [userId, setUserId] = useState<string | null>(localStorage.getItem("userId"));
  const [username, setUsername] = useState<string | null>(localStorage.getItem("username"));
  const clerk = useClerk();
  const { isSignedIn, isLoading: clerkLoaded, user } = useUser();

  // Consider both local auth and Clerk auth for determining authenticated state
  const isAuthenticated = (!!userId && !!username) || (clerkLoaded && isSignedIn);

  // Listen for Clerk authentication changes
  useEffect(() => {
    const checkClerkAuth = async () => {
      // Only proceed if Clerk is loaded
      if (!clerkLoaded) return;

      console.log("Clerk auth state:", { isSignedIn, clerkLoaded });

      // If signed in with Clerk
      if (isSignedIn && user) {
        // First check if we already have local auth data
        const storedUserId = localStorage.getItem("userId");
        const storedUsername = localStorage.getItem("username");

        // Derive potential new values from Clerk
        const clerkUserId = user.id;
        const clerkUsername = user.username ||
          (user.firstName && user.lastName ?
            `${user.firstName} ${user.lastName}` :
            user.emailAddresses?.[0]?.emailAddress || 'user');

        // Check if local storage exists AND matches current state. If not, update
        // state from local storage.
        if (storedUserId && storedUsername && (storedUserId !== userId ||
          storedUsername !== username)) {

```

```

        console.log("Restoring/updating local auth state from storage");
        setUserId(storedUserId);
        setUsername(storedUsername);
    }
    // If no local storage OR Clerk data is different from current state, update
    from Clerk.
    else if (!storedUserId || !storedUsername || clerkUserId !== userId ||
clerkUsername !== username) {
        console.log("Using Clerk user information to update state");
        localStorage.setItem("userId", clerkUserId);
        localStorage.setItem("username", clerkUsername);
        // Only update state if it's actually different
        if (clerkUserId !== userId) setUserId(clerkUserId);
        if (clerkUsername !== username) setUsername(clerkUsername);
    }
    // If Clerk is signed in but state already matches, do nothing

} else if (!isSignedIn && (userId || username)) {
    // Handle case where Clerk signs out but local state still exists (optional)
    // The logout function already handles clearing state, so maybe just log
here.
    console.log("Clerk signed out, local state might still exist.");
}
};

checkClerkAuth();
}, [isSignedIn, clerkLoaded, user]));

const login = (newUserId: string, newUsername: string) => {
    console.log("Login called with", { newUserId, newUsername });
    localStorage.setItem("userId", newUserId);
    localStorage.setItem("username", newUsername);
    setUserId(newUserId);
    setUsername(newUsername);
    navigate("/");
};

const logout = async () => {
    // sign out from Clerk if there's an active session
    try {
        if (clerk.session) {
            await clerk.signOut();
        }
    } catch (error) {
        console.error("Error signing out from Clerk:", error);
    }

    // Clear local auth state
    localStorage.removeItem("userId");
    localStorage.removeItem("username");
    setUserId(null);
    setUsername(null);

```

```

    navigate("/login");
  };

  // Value to be provided to consumers
  const value = {
    userId,
    username,
    login,
    logout,
    isAuthenticated
  };

  return (
    <AuthContext.Provider value={value}>
      {children}
    </AuthContext.Provider>
  );
}

// Hook for components to use the auth context
export function useAuth() {
  const context = useContext(AuthContext);
  if (context === undefined) {
    throw new Error("useAuth must be used within an AuthProvider");
  }
  return context;
}

```

src\convexts

[to top](#)

```

import { useQuery, useMutation, useAction } from "convex/react";
import { api } from "../convex/_generated/api";

export { useQuery, useMutation, useAction, api };

```

src\maintsx

[to top](#)

```

import { StrictMode } from "react";
import ReactDOM from "react-dom/client";
import { BrowserRouter, Routes, Route } from "react-router-dom";
import "./index.css";
import App from "./App";
import { ConvexReactClient } from "convex/react";
import { ConvexProviderWithClerk } from "convex/react-clerk";
import { ClerkProvider, useAuth } from "@clerk/clerk-react";
import { Toaster } from "react-hot-toast";

const convex = new ConvexReactClient(import.meta.env.VITE_CONVEX_URL);
const clerkPubKey = import.meta.env.VITE_CLERK_PUBLISHABLE_KEY;

ReactDOM.createRoot(document.getElementById("root")!).render(
  <StrictMode>
    <ClerkProvider
      publishableKey={clerkPubKey}
      afterSignInUrl="/oauth-callback"
      afterSignUpUrl="/oauth-callback"
      signInUrl="/login"
      signUpUrl="/register"
    >
      <ConvexProviderWithClerk client={convex} useAuth={useAuth}>
        <BrowserRouter>
          <Toaster
            position="bottom-right"
            toastOptions={{
              duration: 4000,
              style: {
                background: '#333',
                color: '#fff',
              },
              success: {
                style: {
                  background: '#10b981',
                },
              },
              error: {
                style: {
                  background: '#ef4444',
                },
              },
            }}
          />
          <Routes>
            <Route path="/*" element={<App />} />
          </Routes>
        </BrowserRouter>
      </ConvexProviderWithClerk>
    </ClerkProvider>
  </StrictMode>,

```

```
);
```

`src\utils\toastts`

[to top](#)


```

import toast from 'react-hot-toast';

// Define toast notification types
export const showToast = {
  /**
   * Show a success toast notification
   */
  success: (message: string) => {
    toast.success(message, {
      icon: '👏',
    });
    console.log(`Success: ${message}`); // For debugging
  },

  /**
   * Show an error toast notification
   */
  error: (message: string) => {
    toast.error(message, {
      icon: '❌',
    });
    console.error(`Error: ${message}`); // For debugging
  },

  /**
   * Show an informational toast notification
   */
  info: (message: string) => {
    toast(message, {
      icon: 'i',
    });
    console.info(`Info: ${message}`); // For debugging
  },

  /**
   * Show a warning toast notification
   */
  warning: (message: string) => {
    toast(message, {
      icon: '⚠️',
      style: {
        background: '#f59e0b',
        color: '#fff',
      },
    });
    console.warn(`Warning: ${message}`); // For debugging
  },

  /**
   * Show a toast notification for session events
   */
  session: {

```

```
    created: () => showToast.success('Session created successfully!'),
    updated: () => showToast.success('Session updated successfully!'),
    deleted: () => showToast.success('Session deleted successfully!'),
    pastDateError: () => showToast.error('Cannot create session in the past. Please
select a future date.'),
    overlapError: () => showToast.error('Cannot create overlapping sessions. You
already have a session scheduled during this time.'),
    joined: () => showToast.success('Joined session successfully!'),
    earlyJoin: (minutesEarly: number) => showToast.info(`You've joined the session
${minutesEarly} minutes early!`),
  }
};
```

src\vite-envd.ts

[to top](#)

```
/// <reference types="vite/client" />
```