Features Implemented

Appointment Booking System Improvements

- **Real Consultant Data Integration**
  - Updated the system to fetch real consultant profiles from Firebase instead of using mock data
  - Modified the `fetchExperts` function to query the `consultantProfiles` collection
  - Implemented mapping between consultant data structure and the application's Expert interface
- **Dynamic Availability Generation**
  - Created algorithm to generate available time slots based on consultant's configured days and hours
  - Converted day names (like "Mon", "Tue") to actual calendar dates
  - Calculated time slots based on appointment duration settings
- **UI/UX Enhancements**
  - Redesigned the appointment booking interface with a more compact layout
  - Implemented a two-column responsive design that adapts to desktop and mobile screens
  - Reduced unnecessary scrolling with fixed height containers and overflow management
  - Added clearer visual indicators for consultant selection
- **Consultant Display Improvements**
  - Created more compact consultant cards with essential information
  - Added visual indicators showing experience, rating, and availability
  - Implemented search and filtering in a space-efficient horizontal layout
  - Improved empty state and loading indicators
- **Appointment Scheduling**
  - Updated the scheduling process to work with the actual data structure
  - Fixed state management for appointment booking
  - Improved the date and time selection interface
  - Added a summary view of appointment details before booking

React Optimization Techniques - Applied to Chat Functionality

Common Performance Issue: Re-fetching on Component Switch

In the initial code, we observed that chat data was being re-fetched every time the user navigated between components. This is a common React performance issue that occurs when:

1. Data fetching logic is placed directly inside component effects without proper conditions

2. Components remount instead of staying preserved in the component tree
3. Global state isn't properly utilized to cache already fetched data

Optimization Solution: Centralized State Management

Here's how we would optimize chat functionality to prevent unnecessary re-fetching:

```
// src/redux/slices/chatSlice.ts
import { createSlice, createAsyncThunk } from '@reduxjs/toolkit';
import { collection, query, where, orderBy, getDocs } from
'firebase/firestore';
import { db } from '../../firebase/config';

// 1. Define a proper chat interface
interface ChatMessage {
  id: string;
  senderId: string;
  receiverId: string;
  message: string;
  timestamp: number;
  read: boolean;
}

interface ChatState {
  messages: Record<string, ChatMessage[]>; // Indexed by chatId for quick
access
  loading: boolean;
  error: string | null;
  currentChatId: string | null;
  lastFetched: Record<string, number>; // Track when each chat was last fetched
}

// 2. Define initial state with proper cache structure
const initialState: ChatState = {
  messages: {},
  loading: false,
  error: null,
  currentChatId: null,
  lastFetched: {}
};

// 3. Create a thunk that's smart about when to fetch
export const fetchChatMessages = createAsyncThunk(
  'chat/fetchMessages',
  async ({ chatId, forceRefresh = false }: { chatId: string, forceRefresh?:
boolean }, { getState, rejectWithValue }) => {
    try {
      const state = getState() as { chat: ChatState };
      const lastFetchTime = state.chat.lastFetched[chatId] || 0;
      const currentTime = Date.now();
```

```
      // Only fetch if:
      // 1. We don't have this chat's messages yet, OR
      // 2. It's been more than 30 seconds since last fetch, OR
      // 3. User explicitly requested a refresh
      if (!state.chat.messages[chatId] ||
          (currentTime - lastFetchTime) > 30000 ||
          forceRefresh) {

        // Fetch messages from Firestore
        const messagesRef = collection(db, 'chatMessages');
        const q = query(
          messagesRef,
          where('chatId', '==', chatId),
          orderBy('timestamp', 'asc')
        );

        const querySnapshot = await getDocs(q);
        const messages: ChatMessage[] = [];

        querySnapshot.forEach((doc) => {
          const data = doc.data();
          messages.push({
            id: doc.id,
            senderId: data.senderId,
            receiverId: data.receiverId,
            message: data.message,
            timestamp: data.timestamp?.toMillis() || Date.now(),
            read: data.read || false
          });
        });

        return {
          chatId,
          messages,
          fetchTime: currentTime
        };
      }

      // Return existing messages if we don't need to fetch
      return {
        chatId,
        messages: state.chat.messages[chatId] || [],
        fetchTime: lastFetchTime
      };
    } catch (error: any) {
      return rejectWithValue(error.message);
    }
  }
);

// 4. Create the slice with proper reducers
const chatSlice = createSlice({
  name: 'chat',
```

```
  initialState,
  reducers: {
    setCurrentChat: (state, action) => {
      state.currentChatId = action.payload;
    },
    markAsRead: (state, action) => {
      const { chatId, messageId } = action.payload;
      if (state.messages[chatId]) {
        const message = state.messages[chatId].find(m => m.id === messageId);
        if (message) {
          message.read = true;
        }
      }
    }
  },
  extraReducers: (builder) => {
    builder
      .addCase(fetchChatMessages.pending, (state) => {
        state.loading = true;
        state.error = null;
      })
      .addCase(fetchChatMessages.fulfilled, (state, action) => {
        const { chatId, messages, fetchTime } = action.payload;
        state.loading = false;
        state.messages[chatId] = messages;
        state.lastFetched[chatId] = fetchTime;
      })
      .addCase(fetchChatMessages.rejected, (state, action) => {
        state.loading = false;
        state.error = action.payload as string;
      });
  }
});

export const { setCurrentChat, markAsRead } = chatSlice.actions;
export default chatSlice.reducer;
```

Component Implementation with Optimizations

```
// src/components/chat/ChatContainer.tsx
import React, { useEffect } from 'react';
import { useDispatch, useSelector } from 'react-redux';
import { useParams } from 'react-router-dom';
import { RootState } from '../../redux/store';
import { fetchChatMessages, setCurrentChat } from
'../../redux/slices/chatSlice';
import ChatMessageList from './ChatMessageList';
import ChatInput from './ChatInput';

const ChatContainer: React.FC = () => {
```

```
  const dispatch = useDispatch();
  const { chatId } = useParams<{ chatId: string }>();
  const { messages, loading, error, currentChatId } = useSelector((state:
RootState) => state.chat);
  const { user } = useSelector((state: RootState) => state.auth);

  useEffect(() => {
    if (chatId && chatId !== currentChatId) {
      // Set current chat ID to track which chat we're viewing
      dispatch(setCurrentChat(chatId));

      // Fetch messages with smart caching logic
      dispatch(fetchChatMessages({ chatId }));
    }

    // Setup realtime listener for new messages
    // (This would be implemented with Firebase onSnapshot)

    return () => {
      // Clean up realtime listener
    };
  }, [chatId, dispatch, currentChatId]);

  // Use memoization to prevent unnecessary re-renders of child components
  const currentMessages = React.useMemo(() => {
    return messages[chatId] || [];
  }, [messages, chatId]);

  if (!user) {
    return <div>Please login to view your messages</div>;
  }

  return (
    <div className="flex flex-col h-full">
      {loading && <div className="loading-indicator">Loading messages...</div>}
      {error && <div className="error-message">{error}</div>}

      <ChatMessageList
        messages={currentMessages}
        currentUserId={user.uid}
      />

      <ChatInput chatId={chatId} receiverId={/* extract receiver ID */} />
    </div>
  );
};

// Use React.memo to prevent re-renders when props haven't changed
export default React.memo(ChatContainer);
```

Breaking Down the Optimization Techniques

1. **Centralized State Management**
    - Using Redux to store chat messages in a normalized structure
    - Indexing messages by chatId for O(1) access time
    - Tracking when data was last fetched to make smart decisions about refreshing
2. **Smart Fetching Logic**
    - Only fetching when necessary (not cached or stale)
    - Using conditional logic in the thunk to avoid network requests
    - Returning cached data when available instead of always fetching
3. **Component Optimization**
    - Using `React.memo` to prevent unnecessary re-renders
    - Using `useMemo` to memoize derived data
    - Proper dependency arrays in useEffect to avoid infinite loops
4. **Edge Case Handling**
    - Handling missing data gracefully (empty arrays instead of undefined)
    - Auth state validation before rendering sensitive content
    - Error states for failed fetches
    - Loading indicators for better UX

Why This Approach Prevents Re-fetching Issues

The key to preventing unnecessary re-fetching when switching components is the combination of:

1. **Persistent Global State**: Redux keeps your data even when components unmount
2. **Smart Fetch Conditions**: We check if we already have data and how fresh it is
3. **Proper Component Lifecycle**: We only trigger fetches when truly needed

When you navigate from ChatA → Settings → ChatA, the second time you visit ChatA:

- The component checks Redux for cached chat data
- Sees that it already has recent data for this chat
- Skips the network request and uses the cached data

This approach significantly reduces API calls, improves performance, and creates a smoother user experience.