```javascript
const express = require('express');
const cors = require('cors');

const authRoutes = require('./routes/auth.js');

const app = express();

const PORT = process.env.PORT || 5001;
require('dotenv').config();

app.use(cors());
app.use(express.json());
app.use(express.urlencoded({ extended: true }));

app.use('/auth', authRoutes);


app.get('/', (req, res) => {
   res.send('Hello, World!');
});

app.listen(PORT, () => {
   console.log(`listening on port ${PORT}`);
});
```

This JavaScript program uses the Express framework to create a basic web server. It includes the `cors` library to handle cross-origin requests, ensuring the server can interact with applications hosted on different domains. The server has a set of authentication routes (`authRoutes`) for handling user authentication, which are set up under the `/auth` path. The server can parse JSON and URL-encoded data, thanks to the `express.json()` and `express.urlencoded()` middleware. It responds with 'Hello, World!' when the root URL (`'/'`) is accessed. Finally, it listens on a port defined in the environment variables or defaults to port 5001, ready to handle incoming requests.

```javascript
const { connect } = require('getstream');
const bcrypt = require('bcrypt');
const StreamChat = require('stream-chat').StreamChat;
const crypto = require('crypto');

require('dotenv').config();
```

```
const api_key = process.env.STREAM_API_KEY;
const api_secret = process.env.STREAM_API_SECRET;
const app_id = process.env.STREAM_APP_ID;

const signup = async (req, res) => {
   try {
       const { fullName, username, password, phoneNumber } = req.body;

       const userId = crypto.randomBytes(16).toString('hex');

       const serverClient = connect(api_key, api_secret, app_id);

       const hashedPassword = await bcrypt.hash(password, 10);

       const token = serverClient.createUserToken(userId);

       res.status(200).json({ token, fullName, username, userId, hashedPassword,
phoneNumber });

   } catch (error) {
       console.log(error);

       res.status(500).json({ message: error });
   }
};

const login = async (req, res) => {
   try {
       const { username, password } = req.body;

       const serverClient = connect(api_key, api_secret, app_id);
       const client = StreamChat.getInstance(api_key, api_secret);

       const { users } = await client.queryUsers({ name: username });

       if(!users.length) return res.status(400).json({ message: 'User not found' });

       const success = await bcrypt.compare(password, users[0].hashedPassword);

       const token = serverClient.createUserToken(users[0].id);
       if(success) {
           res.status(200).json({ token, fullName: users[0].fullName, username,
userId: users[0].id});
       } else {
           res.status(500).json({ message: 'Incorrect password' });
       }
   } catch (error) {ads
       console.log(error);

       res.status(500).json({ message: error });
   }
};

const debug = (req, res) =>{
   res.status(200).send('hello');
};
module.exports = { signup, login , debug}
```

This code forms the backend part of a user authentication system for a chat application, handling user registration, login, and providing necessary tokens for interacting with the Stream Chat service.
Dependencies and Environment Variables:

- The code includes several libraries: `getstream` for integrating Stream Chat services, `bcrypt` for password hashing, `stream-chat` for chat functionality, and `crypto` for generating random user IDs.
- Environment variables for the Stream API (API key, secret, and app ID) are loaded using `dotenv`.

The `signup`, `login`, and `debug` functions are exported for use in route definitions.

```
const apiKey = "8gzvw3b7uqu6";
const authToken = cookies.get('token');
const client  = StreamChat.getInstance(apiKey);

if(authToken) {
   client.connectUser({
       name: cookies.get('username'),
       fullName: cookies.get('fullName'),
       id: cookies.get('userId'),
       phoneNumber:  cookies.get('phoneNumber'),
       image: cookies.get('avatarURL'),
       hashedPassword: cookies.get('hashedPassword')
   }, authToken);
}
```

The application establishes a connection between the authenticated user and the Stream Chat service, allowing for real-time messaging and interaction within the application. Managing secure authentication can be complex, especially when handling tokens and user credentials.

```
const App = () => {
         const [createType, setCreateType] = useState("");
         const [isCreating, setIsCreating] = useState(false);
         const [isEditing, setIsEditing] = useState(false);
         const [showSideBar, setShowSideBar] = useState(true);
         if(!authToken) return <Auth/>

       return (
         <div className='app__wrapper'>
             <Chat client = {client} theme='team light'>
                 <ChannelListContainer
                     showSideBar={showSideBar}
                     setShowSideBar={setShowSideBar}
                     isCreating = {isCreating}
                     setIsCreating = {setIsCreating}
                     setIsEditing = {setIsEditing}
                     setCreateType = {setCreateType}/>
                 <ChannelContainer
                     setShowSideBar={setShowSideBar}
                     isCreating = {isCreating}
                     setIsCreating = {setIsCreating}
                     setIsEditing = {setIsEditing}
                     isEditing = {isEditing}
                     createType = {createType}
                 />
             </Chat>
         </div>
       )
      }
```

This React component, `App`, is the main component of a chat application interface that uses Stream Chat for real-time messaging. The component manages several pieces of state and renders the chat UI based on the user's authentication status.

Managing the state of user authentication (logged in/out) and reflecting that accurately in the UI can be tricky, especially in a dynamic application where the state can change frequently.

```
   const getChannels = async (text) => {
              try{
                  //TODO: fetch channels

                  const channelResponse = client.queryChannels(
                      {
                          type: 'team',
                          name: {$autocomplete: text},
                          members: {$in : [client.userID]}
                      }
                  );
                  const userResponse = client.queryUsers(
                      {
                          id: {$ne: client.userID},
                          name: {$autocomplete: text}
                      }
                  );

                    const [channels, {users}] = await Promise.all([channelResponse,
userResponse]);

                    if(channels.length){
                      setTeamChannels(channels);
                    }
                    if(users.length){
                      setDirectChannels(users);
                    }



              }
              catch(error){
                  setQuery('');
              }
          };
```

This code snippet defines an asynchronous function `getChannels`, which is used for fetching channels and users from the Stream Chat service based on a search query. This is a key function for enhancing user interaction in the chat application, allowing users to efficiently search and navigate through channels and other users in a real-time messaging environment.

```
            const { username, password, phoneNumber, avatarURL } = form;

                  const URL =
'https://chat-app-react-node-ee1bc525b984.herokuapp.com/auth';
                  // const URL = 'https://medical-pager.herokuapp.com/auth';

                  const { data: { token, userId, hashedPassword, fullName } } = await
axios.post(`${URL}/${isSignup ? 'signup' : 'login'}`, {
                      username, password, fullName: form.fullName, phoneNumber,
avatarURL,
                  });
```

This code creates or logs in a user by calling the node.js server and getting the information from the form

```
const ChannelListContent = ({isCreating, setIsCreating, setIsEditing, setCreateType,
setToggleContainer, showSideBar, setShowSideBar}) => {
          const {client} = useChatContext();

          const logout = ()=> {
              cookies.remove("token");
              cookies.remove('username');
              cookies.remove('fullName');
              cookies.remove('userId');
              cookies.remove('phoneNumber');
              cookies.remove('avatarURL');
              cookies.remove('hashedPassword');

              window.location.reload();
          };
```

```
       const filters = { members: { $in: [client.userID] } };

    return (
      <>
          <SideBar logout ={logout}
              setShowSideBar={setShowSideBar}
          />
          <div className='channel-list__list__wrapper'>
              <CompanyHeader/>
              <ChannelSearch
                  setToggleContainer={setToggleContainer}
              />
              <ChannelList
                  filters ={ filters}
                  channelRenderFilterFn={customChannelTeamFilter}
                  List = {(listProps) => (
                      <TeamChannelList
                          {...listProps}
                          type="team"
                          isCreating ={isCreating}
                          setIsCreating={setIsCreating}
                          setIsEditing ={setIsEditing}
                          setCreateType={setCreateType}
                          setToggleContainer ={setToggleContainer}
                      />
                  )}
                  preview = {(previewProps)=>(
                      <TeamChannelPreview
                          {...previewProps}
                          type = "team"
                          setIsCreating={setIsCreating}
                          setIsEditing ={setIsEditing}
                          setToggleContainer ={setToggleContainer}
                      />
                  )}
              />


              <ChannelList
                  filters ={ filters}
                  channelRenderFilterFn={customChannelMessagingFilter}
                  List = {(listProps) => (
                      <TeamChannelList
                          {...listProps}
                          type="messaging"
                          isCreating ={isCreating}
                          setIsCreating={setIsCreating}
                          setIsEditing ={setIsEditing}
                          setCreateType={setCreateType}
                          setToggleContainer ={setToggleContainer}
                      />
                  )}
                  preview = {(previewProps)=>(
                      <TeamChannelPreview
                          {...previewProps}
                          type = "messaging"
                          setIsCreating={setIsCreating}
                          setIsEditing ={setIsEditing}
                          setToggleContainer ={setToggleContainer}
                      />
                  )}
              />
          </div>
      </>
    );
   }
```

Component Functionality:

- `ChannelListContent` is a React component designed to display channel lists in the chat application. It takes several props related to the state and UI manipulation, like `isCreating`, `setIsCreating`, and `showSideBar`.

- Context and Client:
  - It uses `useChatContext` from `stream-chat-react` to access the chat client (`client`). This client is used to interact with the Stream Chat API.
- Filters for Channels:
  - The `filters` object is defined to filter channels where the current user (`client.userID`) is a member. This ensures that the displayed channels are relevant to the logged-in user.
- Rendering Channel Lists:
  - The `ChannelList` component from `stream-chat-react` is used to render the list of channels.
  - It is configured with `filters` to control which channels are shown.

    - `channelRenderFilterFn` is a function (`customChannelTeamFilter`) that further filters the channels to display only those of type 'team'.
- Dynamic List Rendering:
    - The `List` prop of `ChannelList` takes a function that returns a `TeamChannelList` component, which is responsible for rendering the actual list of channels.
    - This component receives various props related to channel creation and editing states (`isCreating`, `setIsCreating`, etc.), allowing for dynamic interaction within the application.

```
const UserItem = ({user, setSelectedUsers}) => {



   const [selected, setSelected ] = useState(false);



   const handleSelect = ()=> {

       if(selected){

           setSelectedUsers((prevUsers) => prevUsers.filter((prevUser) => prevUser !==
user.id ));

       }

       else{

           setSelectedUsers( (prevUsers) => [...prevUsers, user.id]);

       }

       setSelected((prevSelected)=> !prevSelected);

   };



   return (
       <div className='user-item__wrapper' onClick={handleSelect}>

           <div className='user-item__name-wrapper'>

               <Avatar

                   image = {user.image}

                   name = {user.fullName || user.id}

                   size ={32}

               />

               <p className='user-item__name'>{user.fullName || user.id}</p>

           </div>



           {selected ? <InviteIcon />: <div className='user-item__invite-empty'/>}

       </div>

   )

};
```

- Interactive UI Elements:
  - The `UserItem` component provides an interactive element allowing users to select or deselect users from a list, which is a common requirement in chat applications for creating or managing chat groups.
- Effective State Handling:
  - The use of React's useState and functional updates demonstrates effective state handling, crucial in dynamic applications where the UI needs to respond to user interactions.

```
const TeamChannelPreview = ({setActiveChannel, setIsCreating,
setIsEditing,setToggleContainer, channel, type}) => {



   const {channel : activeChannel, client } = useChatContext();



   const ChannelPreview = () => (

       <p className='channel-preview__item'>

           # {channel?.data?.name || channel?.data?.id}

       </p>

   );



   const DirectPreview = () => {

       const members = Object.values(channel.state.members).filter(

           ({user}) => user.id !== client.userID

       )



       return (

           <div className='channel-preview__item single'>

               <Avatar

                   image = {members[0]?.user?.image}

                   name =  {members[0]?.user?.fullName}

                   size = {24}
               />

               <p>{members[0]?.user?.fullName || members[0]?.user?.id}</p>

           </div>

       )

   }




 return (

   <div className={

       channel?.id === activeChannel?.id ?
'channel-preview__wrapper__selected':"channel-preview__wrapper"

   }

   onClick={() => {

      setIsCreating(false);

      setIsEditing(false);

      setActiveChannel(channel);

      if(setToggleContainer) {

       setToggleContainer( (prevState)=>!prevState);

      }

   }}

   >

       {type === "team" ? <ChannelPreview/> : <DirectPreview/>}
   </div>

 )

}
```

    - `TeamChannelPreview` is a functional component used to display a preview of a chat channel. It receives several props, including functions for setting application state (`setActiveChannel`, `setIsCreating`, `setIsEditing`) and data about the channel and its type.
- Using Stream Chat Context:
    - The component utilizes `useChatContext` from `stream-chat-react` to access the current active channel and the chat client. This context provides essential data for determining how to render the channel preview.
- Conditional Rendering:
    - The component renders differently based on the `type` of the channel ('team' or 'direct').
    - `ChannelPreview` is rendered for team channels, displaying the channel's name.
    - `DirectPreview` is rendered for direct channels, showing the avatar URL and name of the other user in the chat (excluding the current user).
- Dynamic Styling:
    - The outer `div` has its class set dynamically. If the channel is the active channel, it uses a 'selected' style; otherwise, it uses a default wrapper style.
- OnClick Functionality:
    - When a channel preview is clicked, several state-updating functions are called. These actions reset creation and editing states and set the active channel, affecting the overall UI and state of the chat application.